



Faculty of Science and
Bio-Engineering Sciences
Department of Computer Science
Artificial Intelligence Laboratory

On the Semi-automated Design of Reusable Heuristics

with applications to hard combinatorial optimization

Dissertation submitted in fulfilment of the requirements for the degree of Doctor of Science: Computer Science

Steven Adriaensen

Promotor: Prof. Dr. Ann Nowé (Vrije Universiteit Brussel)

© 2018 Steven Adriaensen

All rights reserved. No parts of this book may be reproduced or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the author.

Abstract

These days, many scientific and engineering disciplines rely on standardization and automated tools. Somewhat ironically, the design of the algorithms underlying these tools is often a highly manual, ad hoc, experience- and intuition-driven process, commonly regarded as an “art” rather than a “science”.

The research performed in the scope of this dissertation is geared towards improving the way we design algorithms. Here, we treat algorithm design itself as a computational problem, which we refer to as the Algorithm Design Problem (ADP). In a sense, we study “algorithms for designing algorithms”.

The main topic we investigate in this framework is the possibility of solving the ADP automatically. Letting computers, rather than humans, design algorithms has numerous potential benefits. The idea of automating algorithm design is definitely not “new”. Attempts to do so can be traced back to the origins of computer science, and, ever since, the ADP, in one form or another, has been considered in many different research communities. Therefore, we first present an overview of this vast and highly fragmented field, relating the different approaches, discussing their respective strengths and weaknesses, towards enabling a more unified approach to automated algorithm design.

One design approach, which we explore in more depth, solves the ADP by reduction to an Algorithm Configuration Problem (ACP), a practice to which we refer to as Programming by Configuration (PbC). We demonstrate PbC, using it to design a non-exact algorithm (a.k.a. a “heuristic”) for solving hard combinatorial optimization problems. While PbC has previously been applied to the design of heuristics for some specific problem, our approach differs in that we consider the automated design of domain-independent heuristics

(a.k.a. “metaheuristics”) which are reusable across a wide class of problems. As a result, we obtain a metaheuristic which we show to be competitive with the state-of-the-art across nine different problem domains.

Subsequently, we present a critical reflection on PbC, discuss its limitations, and our own work towards addressing these. In particular, we argue that PbC, in reducing the ADP to an ACP, abstracts information that can be usefully exploited to solve the ADP faster. To this end, we propose a novel performance estimation technique exploiting this information to generalize performance observations across designs. We describe a design procedure implementing our technique and validate experimentally that it improves the data efficiency (up to several orders of magnitude) in three selected scenarios.

Finally, we explore an alternative, less disruptive angle. Here, instead of aiming at replacing the (manual) process by which algorithms are currently being designed, we investigate the possibility of improving the quality of the resulting design post hoc. In particular, we focus on ‘simplicity’, a factor that is often overlooked when designing algorithms, which consequently tend to be overly complicated. To counteract this effect, we advocate Accidental Complexity Analysis (ACA), a research practice targeted at identifying excessive complexity in algorithms. We demonstrate ACA, applying it to an existing state-of-the-art metaheuristic. As an outcome of this analysis, we are able to reduce its logical complexity by a factor of eight, without loss of performance.

Samenvatting

Vandaag de dag zijn vele wetenschappelijke en technische disciplines afhankelijk van standaardisatie en geautomatiseerde tools. Het ontwerp van de algoritmen die deze tools ondersteunen, gebeurt echter nog vaak handmatig, gestuurd door de ervaring en intuïtie van de ontwerper, en wordt vaak als een “kunst” eerder dan een “wetenschap” beschouwd.

Het onderzoek uitgevoerd in het kader van dit doctoraat is gericht op het verbeteren van dit algoritmeontwerp. Hier behandelen we algoritmeontwerp zelf als een computationeel probleem, dat we het “Algorithm Design Problem” (ADP) noemen. In zekere zin bestuderen we “algoritmen voor het ontwerpen van algoritmen”.

In dit kader onderzoeken we in het bijzonder de mogelijkheid om computers, in plaats van mensen, algoritmen te laten ontwerpen. Dit biedt tal van mogelijke voordelen. Het idee om algoritmeontwerp te automatiseren is zeker niet “nieuw”. Pogingen daartoe zijn terug te vinden vanaf het ontstaan van de computerwetenschap. Sindsdien is dit probleem, in één of andere vorm, beschouwd in vele verschillende onderzoeksgemeenschappen. Daarom presenteren we eerst een overzicht van dit zeer uitgebreide en gefragmenteerde onderzoeksdomein, waarin we de verschillende benaderingen relateren, hun respectievelijke sterke en zwakke punten bespreken, om zo een meer uniforme benadering van geautomatiseerd algoritmeontwerp mogelijk te maken.

Eén specifieke benadering, die we nader onderzocht hebben, ontwerpt algoritmen met behulp van geautomatiseerde tools die bekend staan als “algoritmeconfiguratoren”. Een praktijk waarnaar we verwijzen als “Programming by Configuration” (PbC). We illustreren het gebruik van PbC in het ontwerp van een niet-exact algoritme (ook wel een “heuristiek” genoemd) voor het oplossen van moeilijke combinatorische optimalisatieproblemen. Terwijl

PbC eerder werd toegepast op het ontwerp van heuristieken voor een specifiek probleem, verschilt onze aanpak in dat we het geautomatiseerde ontwerp van domeinonafhankelijke heuristieken (ook wel “metaheuristieken” genoemd) beschouwen die herbruikbaar zijn voor een groot aantal van verschillende problemen. We tonen aan dat de op deze wijze verkregen metaheuristiek even goed werkt als de best bestaande vergelijkbare technieken voor negen verschillende probleemdomeneinen.

Vervolgens, presenteren we een kritische reflectie op PbC. We bespreken de beperkingen ervan, en ons eigen werk om deze aan te pakken. In het bijzonder stellen we dat PbC nuttige informatie abstraheert die kan worden gebruikt om het ADP sneller op te lossen. Hiertoe stellen we een nieuwe techniek voor die deze informatie gebruikt om algoritmeprestaties te schatten aan de hand van minder experimentele data. We beschrijven een ontwerpprocedure die onze techniek implementeert en tonen aan dat deze weldegelijk de gegevens efficiëntie (tot met meerdere grootteorden) verbetert in drie geselecteerde experimentele scenario's.

Ten slotte verkennen we een alternatieve, minder disruptieve invalshoek. Hier, in plaats van het (handmatige) proces waarmee de algoritmen momenteel worden ontworpen te proberen te vervangen, onderzoeken we de mogelijkheid om achteraf de kwaliteit van het resulterende ontwerp te verbeteren. In het bijzonder richten we ons op 'eenvoud', een factor die vaak over het hoofd wordt gezien bij het ontwerpen van algoritmen, die bijgevolg de neiging hebben overgecompliceerde te zijn. Om dit effect tegen te gaan, pleiten we voor Accidental Complexity Analysis (ACA), een onderzoekspraktijk gericht op het identificeren van onnodige complexiteit in algoritmen. Als demonstratie, passen we ACA toe op een bestaande metaheuristiek. Deze analyse stelt ons in staat om de complexiteit hiervan met een factor van acht te verminderen, zonder prestatieverlies.

Contents

Abstract	3
Samenvatting	5
Contents	7
1 Introduction	11
1.1 Our Critical Perspective on Algorithmics	11
1.2 Research Topic	14
1.2.1 Scope and Focus	14
1.2.2 Research Questions	15
1.3 Structure of this Dissertation	16
1.3.1 Content Outline	16
1.3.2 Reader's Guide	18
 Part 1: Background and Personal Perspectives	 19
2 Foundations of Problem-solving	21
2.1 Problems	21
2.2 Optimization Problems	23
2.2.1 Examples of Combinatorial Optimization Problems	24
2.3 Algorithms	26
2.3.1 Formalization	29
2.3.2 Computer Programs	32
2.3.3 Parametrized Algorithms	33
2.4 Problem Formulation	34
2.4.1 Reducibility	34
2.4.2 Problem-solving vs. Problem Formulation	37

2.4.3	Opinion: Importance of Natural Formulations	38
2.5	Computational Complexity	39
2.5.1	Analysis of Algorithms	39
2.5.2	Complexity of Problems	41
2.6	Beyond Effective Procedures	43
2.6.1	Randomized Algorithms	44
2.6.2	Asymptotically Correct Algorithms	45
2.6.3	Non-exact Algorithms	47
2.6.4	Contract and Anytime Algorithms	48
2.7	Search Methods for Combinatorial Optimization	49
2.7.1	Systematic Search	50
2.7.2	Heuristic Search	54
2.8	Summary	67
3	Algorithm Design	69
3.1	The Algorithm Design Problem	70
3.2	Algorithm Quality	71
3.2.1	Performance	71
3.2.2	Beyond Performance	73
3.2.3	Conflicting Criteria?	74
3.3	Who is the Designer?	80
3.3.1	The Manual Approach	80
3.3.2	Automated Algorithm Design	82
3.3.3	Semi-automated Design	86
3.4	What Information Does the Designer Use?	93
3.4.1	Analytical	93
3.4.2	Empirical / Simulation-based	93
3.4.3	Who Uses What Information?	94
3.5	When Does Design Take Place?	94
3.5.1	Offline	94
3.5.2	Online	95
3.5.3	Semi-online	98
3.6	Summary	99
4	Design by Algorithm Selection	101
4.1	Design by Per-set Algorithm Selection	102
4.1.1	Formulating the ADP as a set-ASP	102
4.1.2	Solving the set-ASP	106
4.2	Design by Per-input Algorithm Selection	117

4.2.1	The input- and subset-ASP	117
4.2.2	Formulating the ADP as a subset-ASP	118
4.2.3	Solving the subset-ASP	121
4.3	Design by Dynamic Algorithm Selection	129
4.3.1	The Dynamic ASP	130
4.3.2	Solving the Dynamic ASP	139
4.4	A Broader Perspective	147
4.4.1	Relations between the ASPs	147
4.4.2	Opinion: Relative Strengths and Weaknesses	148
4.5	Summary	150

Part 2: Algorithmic Contributions and Case Studies 151

5 Programming by Configuration 153

5.1	Programming by Optimization	154
5.2	Case Study: Designing Reusable Metaheuristics	156
5.2.1	The Metaheuristic Design Problem	156
5.2.2	Reduction to an ACP	161
5.2.3	Solving the ACP	166
5.2.4	Results	167
5.3	Fair Share Iterated Local Search (FS-ILS)	169
5.3.1	Design Decisions	169
5.3.2	Empirical Analysis	172
5.4	Critical Reflection	188
5.5	Summary	189

6 Towards a White Box Approach to Automated Algorithm Design 191

6.1	Opinion: Limitations of PbC	192
6.2	A White Box Formulation	194
6.2.1	How about the Dynamic Algorithm Selection Problem?	194
6.2.2	A White Box Variant of the Algorithm Configuration Problem	195
6.2.3	Relation to Other Formulations	197
6.3	Exploiting White Box Information	200
6.3.1	Algorithm Performance Evaluation	200
6.3.2	An Importance Sampling Approach	201
6.4	A Proof of Concept White Box Optimizer	210
6.4.1	Choice of Incumbent	210
6.4.2	High-level Search Strategy	213

- 6.5 Experimental Validation 217
 - 6.5.1 The Looping Problem: Break or Continue 218
 - 6.5.2 Static Sorting Portfolio 220
 - 6.5.3 Dynamic Metaheuristic Scheduler 223
- 6.6 Limitations and Future Research 226
- 6.7 Summary 227
- 7 Accidental Complexity Analysis 229**
 - 7.1 Simplicity in Algorithm Design 230
 - 7.2 Identifying Accidental Complexity 231
 - 7.3 Case Study: Accidental Complexity in Reusable Metaheuristics 231
 - 7.3.1 Context and Motivation 232
 - 7.3.2 Empirical Study 233
 - 7.4 Critical Reflection 240
 - 7.5 Summary 241
- 8 Conclusion 243**
 - 8.1 Research Questions (revisited) 243
 - 8.2 Limitations and Future Research 248
- A Appendix 251**
 - A.1 Benchmark Environment Description 251
 - A.2 The HyFlex Benchmark Set 252
 - A.3 Properties of $\hat{\delta}$ and Their Proofs 254
 - A.4 The Dynamic Metaheuristic Scheduling wb-ACP 258
 - A.5 AdapHH's Anatomy 262
- List of Publications 269**
- List of Acronyms 271**
- Bibliography 277**

1 | Introduction

The first chapter of this dissertation is structured as follows: First, in Section 1.1, we provide some context and motivation for the research we have performed in the scope of this dissertation. Subsequently, in Section 1.2, we give an overview of the main topics covered in this work. Finally, in Section 1.3, we explain the structure of this document, outline its content and guide the reader to the information of interest.

1.1 Our Critical Perspective on Algorithmics

In this section, we first provide a brief introduction to algorithmics. Subsequently, we express our concerns about common practices in algorithmic research. We wish to stress that this section reflects our own personal perspective on the matter, and that views differ. Also, there is no one-to-one correspondence between the issues we raise here and the specific research we have performed in this dissertation, i.e. we did not actually address all of these issues. Nonetheless, as these are the concerns which have motivated our research, we believe that the broader personal perspective we sketch here is important to understand the decisions we made and what we ultimately aim to achieve, independently of what we actually achieved within the limits of this dissertation. Also, as it is our belief that these issues impede scientific progress in algorithmics (and beyond), we feel compelled to bring these to the awareness of the reader. Please note that algorithmics is subdivided in many different subfields and that some of the issues we raise here are more strongly pronounced in some of these subfields than in others. Also, we intend to question the methods, not the researchers, who arguably do their best with the methods available to them.

What are algorithms?

Algorithmics is the science devoted to the design and analysis of “algorithms”. Algorithms describe systematic procedures for solving problems in such way that they can be executed without requiring any ingenuity from the agent executing them [Audi 2015]. While algorithmics is often regarded as a subfield of computer science, i.e. studying procedures intended to be executed by computers, the field predates computers and the conceptual notion of algorithm can be traced back to ancient times [Cooke 2011]. Therefore, for those not familiar with computer science, let us consider some examples of daily life algorithms, intended to be executed by humans:

Recipes describe how to prepare a given dish from raw ingredients.

Construction manuals provide instructions to build something from bits and pieces.

Remark that general purpose digital computers manipulate binary numbers, rather than pots and pans or hammers and nails, somewhat affecting how algorithms are described and restricting the class of problems they solve in practice.

Why algorithms?

Algorithms formalize problem-solving procedures. This formalization facilitates sharing, reproduction, and reuse. You do not bother creating recipes for a dish that is only cooked once. People create recipes to make the same dish again in the future, and to allow others to do the same. Similarly, often, what makes formalizing a solution approach as an algorithm worthwhile, is its ability to be shared with and reused by the rest of the community. In particular, algorithms enable others to solve problems, requiring merely the ability to rigorously follow simple instructions. Recipes enable lesser chefs to prepare delicious meals. Construction manuals allow not-so-handly (wo)men to build their own furniture, saving Ikea™ a lot of money. Similarly, algorithms enable computers to solve problems, i.e. they enable *automation*. Computers, given proper instructions, can solve many problems orders of magnitude faster, more accurately and cheaper than humans.

How are algorithms designed?

Algorithmics, ever since its advent, has contributed to major scientific breakthroughs and feats of engineering. These days, most scientific and engineering disciplines rely heavily on standardization and automated tools. Somewhat ironically, the design of the algorithms underlying these tools is often a highly manual, ad hoc, experience- and intuition-driven process, commonly regarded as an “art” rather than a “science” [Hunt and Thomas 2001].

As a consequence, the design of an algorithm is time-consuming and costly. Furthermore, this process is rarely documented, making it untraceable: It is often unclear what motivated certain design decisions (e.g. natural constraints, expert knowledge, intuition, experimentation, etc.) and what alternatives were considered. Not only do we, in this way, lose potentially interesting information and insights that could otherwise be used to design algorithms in the future, it also makes the process susceptible to personal bias.

Digression: Personal Biases in Algorithmics

Researchers in algorithmics are subject to numerous “perverse incentives” that possibly negatively affect the way algorithms are designed, analyzed and how these results are presented to the rest of the community. Everybody wants to be “famous”. In algorithmics, most “celebrities” have come up with new and better ways to solve a problem. It is only natural, that many researchers attempt to do the same, hoping to achieve similar fame. As not everybody can be famous, algorithm design on a community level is often a competitive rather than a collaborative effort, focused on the novelty and performance of algorithms:

Novelty in science is important, as one should not plagiarize somebody else’s work. However, novelty in some communities has become almost synonymous with proposing new algorithms. We argue that this relentless pursuit of novelty has led to a relative scarcity of critical, incremental and fundamental research, leading to new insights and a better understanding of existing techniques. It also discourages reusing or accurately relating to existing techniques, in fear of one’s contribution being regarded as less novel. Finally, it is notoriously difficult¹ to distinguish “actual” novelty (whatever form that may take...) from novelty in representation, making it in our opinion an improper measure for scientific contributions.

Performance: Empirical studies in algorithmics often more resemble track meets than they do scientific endeavors [Hooker 1995]. While such studies, if conducted fairly, tell us which algorithm performs best, they provide little insight into why this is the case, or whether these performance observations can be generalized beyond the specific setting considered in the study. The latter is particularly troublesome, as the problem instances considered by researchers often differ from those of interest to practitioners. Researchers test their methods on well-known “benchmark” problems to compare their results with those obtained in prior research. Practitioners on the other hand are interested in solving “new” problem instances of the particular problem they are faced with. Practical applications fail when methods fail to generalize to this

¹For instance, it is provably impossible to determine whether two arbitrary algorithms are functionally different.

new setting. Furthermore, to compete in “races” with state-of-the-art, often very mature (commercial) solvers, the code must be highly optimized and implemented in low-level programming languages. This takes up a tremendous amount of time and effort, results in overly complex code artifacts with limited portability and it is often unclear whether algorithmic innovation or micro-optimized implementations are to be credited for superior performance. Finally, we risk abandoning actually novel and promising ideas, just because they are not yet able to “beat” such solvers.

As many scientific fields, also algorithmics suffers from an increasing publication pressure: “publish or perish” [Sarewitz 2016], requiring the researcher to regularly publish his/her research, and this preferably in a high impact journal [Colquhoun 2003], which forces him/her to spend a lot of time on writing convincing research articles and leaving little time for actual research, let alone to perform elaborate and unbiased surveys of prior art. On the other hand, it has also lead to a proliferation of publications, which represent prior art a researcher is supposed to be aware of. While publications increase exponentially [Larsen and Von Ins 2010], funding, top-tier journals, peer-reviewers, research positions etc. do not, putting pressure on the scientific system and further increasing competitiveness. This also impacts how results are represented to the rest of the community. Negative results rarely get published [Fanelli 2011], inducing a scientific bias. Established scientific vocabulary is replaced by marketing buzzwords [Vinkers et al. 2015] and metaphors [Sörensen 2015]. Research artifacts such as data and code backing up results are often not shared [Collberg et al. 2015], algorithmic contributions are represented overly complex [Adriaensen and Nowé 2016a], unnecessarily complicating reuse. Failure to accurately relate to and reuse prior art, which results in a fragmentation of the field and reinvention [Weyland 2010]. Beyond perverse incentives, we believe that at the base often lies the lack of incentives to invest the effort and accept the risks associated with transparency.

1.2 Research Topic

1.2.1 Scope and Focus

As argued in the previous section, the contemporary approach to designing, analyzing and representing algorithmic contributions spawns a host of evils that we believe impede both scientific progress and practical applications. All research we performed in the scope of this dissertation was therefore geared towards improving the way we conduct algorithmic research, as a community. That being said, this topic is too broad to be reasonably treated in any depth in this dissertation. We therefore focused on the following more specific topics:

Semi-automated design: Given our background in artificial intelligence, one of the main topics we investigated in this framework was the possibility of (partially) automating algorithm design. Letting computers, rather than humans, design algorithms has many potential benefits: Computers are faster, cheaper and unbiased. On the other hand, we believe that fully eliminating the human researcher would be overly radical. Therefore, we focused our investigation on semi-automated approaches, where automation begins where expert knowledge ends.

Reusability is one of the main factors making the design of an algorithm worthwhile. Remark that we view reuse here not as being limited to the direct reuse of implementations in applications or analysis, but to also include re-implementations, incremental refinements, hybridizations with other algorithms, partial reuse and use as a source of inspiration. In our opinion, there is little merit in sharing your algorithms if they cannot be reused by the rest of the community. Also, from an economics perspective, reuse of an algorithm allows us to amortize the cost associated with its design. Yet, we argue that current practices in algorithmics have a tendency to produce artifacts with poor reusability. Therefore, another topic we investigated was how to improve the reusability of algorithmic contributions.

Heuristics: The main application area we considered was the design of heuristics for solving hard combinatorial optimization problems. Many interesting combinatorial optimization problems cannot be solved efficiently. A classical example is the traveling salesman problem [Jünger et al. 1995]. Difficult instances of these problems are therefore in practice typically solved using heuristics. Heuristics are non-exact problem-solving procedures which do not provide any relevant theoretical guarantees w.r.t. the quality of the solutions they return, yet in practice they often quickly find good solutions to large and otherwise intractable problems. Our interest in heuristic optimization, as a case study, stems from the fact that many of the aforementioned issues are notably pronounced in this area.

1.2.2 Research Questions

Given this focus, the central question we aimed to answer in this dissertation was

[Q.A]: *How can computers help us in designing reusable heuristics?*

That is, we investigated the use of semi-automated design approaches as an alternative for the manual, ad hoc, experience- and intuition-driven process by which heuristics are most commonly designed.

Here, our design objective was reusability across a wide variety of use cases, as opposed to classical objectives such as novelty or peak performance on a specific use case. One question we investigated in this context was

[Q.A.1]: *What makes one heuristic more reusable than another?*

The idea of automating algorithm design is definitely not “new”. Attempts to do so can be traced back to the origins of computer science, and, ever since, the problem, in one form or another, has been considered in many different research communities. Therefore, rather than inventing new ways of designing algorithms, we first examined

[Q.A.2]: *Which of the existing approaches is best suited to design reusable heuristics?*

Subsequently, we looked at the main limitations of these techniques and how they could be addressed, i.e.

[Q.A.3]: *How can we further improve this approach?*

While automation is the central theme of this dissertation, we realize that it is somewhat idealistic to assume that all researchers will suddenly adopt a semi-automated design approach. Also, all its shortcomings aside, it cannot be denied that the manual approach has resulted in highly performant algorithms. We therefore also explored a less disruptive angle where we take the design (approach) as a given, i.e.

[Q.B]: *How can we improve the reusability of heuristics post-hoc?*

We will revisit and formulate our answers to these research questions in Section 8.1.

1.3 Structure of this Dissertation

In this section, we provide a brief overview of the structure of this work to help the reader in navigating this lengthy document.

1.3.1 Content Outline

Besides this introduction (Chapter 1), it consists of six main chapters and a conclusion (Chapter 8). Each chapter is self-contained and provides a brief introduction (preceding the first subsection), presenting the context and motivation for its existence, as well as an outline of its contents. Furthermore, the final section of each of the six main chapters summarizes the content covered in each (sub)section. The main content of this dissertation is subdivided in two parts.

Part I: Background and Personal Perspectives (Chapters 2, 3 and 4)

In the first part, we introduce core concepts and an overview of the field. Here, we go well beyond the standard textbook material, and present our own perspective on prior art. As such, Part I not only makes the research in later chapters (i.e. Part II) more accessible to readers not familiar with the topic, it also provides a broader context and motivation for the specific research we performed in these chapters. It is structured as follows:

Chapter 2 provides a comprehensive introduction to algorithmics, in general, and to the subfield of heuristic optimization, in particular.

Chapter 3 introduces the “Algorithm Design Problem” (ADP) and presents a broad overview of the approaches which have been taken, thus far, to solve this problem.

Chapter 4 discusses three generic solution approaches which we regard as being prototypical for semi-automated algorithm design.

Part II: Algorithmic Contributions and Case Studies (Chapters 5, 6 and 7)

The second part is based on our publications.² In contrast to Part I, the research described in these chapters has a more specific scope, contributions are more tangible, and claims are supported by empirical evidence. It is structured as follows:

Chapter 5 explores a specific, promising, semi-automated design approach in more depth: Programming by Configuration (PbC). First, it demonstrates how PbC can be used to design a domain-independent heuristic (a.k.a. a metaheuristic): FS-ILS [Adriaensen et al. 2014a, Adriaensen et al. 2014b]. Subsequently, it analyzes FS-ILS’s reusability. In particular, it investigates its generality [Adriaensen et al. 2015], evaluating its performance across nine different classical combinatorial optimization problems.

Chapter 6 discusses the limitations of PbC and how they could be addressed. In particular, it argues that PbC abstracts information which can be usefully exploited to solve the ADP faster [Adriaensen and Nowé 2016b]. Subsequently, it describes a novel performance evaluation strategy that uses this information, discussing its merits and validating these experimentally [Adriaensen et al. 2017].

Chapter 7 argues for the importance of presenting algorithmic contributions as simple as possible and advocates the use of Accidental Complexity Analysis (ACA) to help doing so [Adriaensen and Nowé 2016a]. It demonstrates ACA, using it to analyze the presence of accidental complexity in AdapHH, a state-of-the-art metaheuristic.

²A notable exception here is our formalization of the dynamic algorithm selection problem [Adriaensen and Nowé 2016b], which is covered in Chapter 4 (see Section 4.3.1).

1.3.2 Reader's Guide

In this dissertation, we cover a wide range of different topics. Our treatment thereof is not exactly minimal, i.e. we frequently elaborate, presenting formal definitions, personal perspectives, critical discussions, etc. While we are convinced of the added value of doing so, we realize that not all of the content covered in this document will be of interest to all of our readers; or that some of our readers may not have the time to read this lengthy document entirely. In this section, we propose alternative ways of (partially) reading our work, which allow the more selective reader to adjust his/her reading according to his/her background knowledge and interests.

Your Background Knowledge?

Readers which have a background in computer science will likely be able to skip most of Chapter 2 and nonetheless understand the content of interest in later chapters (3-7). While we personally find “reading a different perspective on familiar topics” an enriching experience, we advise the more selective reader to read Section 2.8, which briefly summarizes the content covered in each subsection of Chapter 2, and to only read the subsections on those topics he/she is less familiar with.

Your Interests?

A general overview: Readers interested in a full overview of the content covered in this dissertation, but who do not have the time to read it entirely, we advise to read

1. The Abstract, for a brief summary of our research and our main results/findings.
2. Chapter 1, for additional context and motivation.
3. Sections 2.8, 3.6, 4.5, 5.5, 6.7 and 7.5, for a brief, yet complete, summary of the content covered in chapters 2, 3, 4, 5, 6 and 7, respectively.
4. Chapter 8, for a discussion of limitations and further research.

A specific part/chapter: Each chapter is self-contained and can be read independently. Frequent cross-references indicate information that may facilitate further understanding. Here again, we advise the selective reader to first read the summary at the end of the chapter, allowing him/her to read only the subsections of interest.

A specific research question: In Section 8.1, we formulate answers to our research questions. This section also summarizes and references the most relevant subsections.

Part 1:

**Background and Personal
Perspectives**

2 | Foundations of Problem-solving

In this chapter, we provide a general introduction to “problem-solving using computers”. Here, we introduce various core concepts informally, to make our work in later chapters accessible to readers less familiar with the domain, and formally, to avoid ambiguity and to support our argumentations. Specifically, in Section 2.1 we explain what *problems* are, and subsequently, in Section 2.3, we introduce *algorithms* as “effective procedures” to solve them. In Section 2.4, we discuss the interface between problems and algorithms. In Section 2.5, we present a primer to complexity theory, the theoretical study of the resources required to solve problems. In Section 2.6, we argue that the traditional definition of what constitutes an algorithm is overly restrictive, and describe various generalizations of this notion. Alongside this general introduction, we pay special attention to the combinatorial optimization problem (see Section 2.2) and how to solve it (see Section 2.7). Finally, in Section 2.8 we provide a summary of the content covered in this chapter.

2.1 Problems

In this dissertation, we frequently talk about “problems” and how to (best) solve them. World problems such as climate change, hunger, poverty, war etc., might come to mind. In what follows, we conceptually and formally define the notion of problems as considered in this dissertation.

Conceptually, we consider a problem to be any “statement demanding a solution”. Take for example the following problem: “sort $[5,3,1,5,8]$ in ascending order”; the solution to this problem is “ $[1,3,5,5,8]$ ”. Another example is the following: “find a non-trivial prime factor of 1183”; a solution to this problem is 7. More generally, we will use the term “problem” to denote any possibly infinite collection of such statements and use the term “problem *instance*” to refer to a specific element thereof. For example “sort $[5,3,1,5,8]$ in ascending order” and “sort $[2,8,4]$ in ascending order” are both instances of the more general “sort a list of integers l in ascending order” (sorting) problem. Similarly, “find a non-trivial prime factor for 1183” and “find a non-trivial prime factor for 1013” are both instances of the more general “find a non-trivial prime factor for the natural number n ” (prime factor finding) problem.

More specifically, in this dissertation and computer-science in general, we study the class of *computational problems*. Conceptually, computational problems are problems having the “appropriate form” to be solved using computation.¹ As we will see in Section 2.3, computation is commonly viewed as “deriving an output for a given input”. As such, computational problems are formalized in terms of “desired outputs” for “possible inputs”.

Definition 2.1: computational problem [Trevisan 2010]

A computational problem is a pair (X, R) , where X is the set of possible inputs and $R \subseteq X \times Y$ the problem relation, specifying admissible input/output pairs. In a computational problem (X, R) , given any input $x \in X$, we are to compute an output y satisfying $(x, y) \in R$ (or indicate that no such y exists).

Conceptually, each $(x, y) \in R$ (also denoted xRy) represents a problem instance x and a solution y thereof. Note that instances of a computational problem may have more than one admissible solution, or even none at all. We define $R(x) = \{y \mid xRy\}$ as the set of admissible outputs for an input x . Both sorting and factorization problems described above can be formalized as computational problems:

Prime factor finding corresponds to a pair $(\mathbb{N}, R_{\text{pfactor}})$ where

$$R_{\text{pfactor}} = \{(n, p) \in \mathbb{N} \times \mathbb{P} \mid 1 < p < n \wedge \frac{n}{p} \in \mathbb{N}\}.$$

Remark that $R(1183) = \{7, 13\}$ and $R(1013) = \emptyset$ (since $1013 \in \mathbb{P}$).

¹As we will see in Section 2.3.1.1, not all computational problems (as in Definition 2.1) can actually be solved using computation (as in Definition 2.8).

Sorting is described by the pair $(\mathbb{Z}^*, R_{\text{isorta}})$, where \mathbb{Z}^* is the set of arbitrary length integer lists. Let the set of permutations of $l \in \mathbb{Z}^*$ be defined as

$$P^l = \{l' \in \mathbb{Z}^{|l|} \mid \exists \text{ a bijection } p : \mathbb{N} \rightarrow \mathbb{N} \text{ such that } l'_i = l_{p(i)}, \forall i\}.$$

and $R_{\text{isorta}} = \{(l, l') \in \mathbb{Z}^* \times \mathbb{Z}^* \mid l \in P^{l'} \wedge l'_j \leq l'_{j+1}, \forall 1 \leq j < |l|\}$, i.e. relating any list of integers l to its sorted permutation l' .

Based on properties of the problem relation R , we further distinguish between two sub-classes of computational problems: Function and decision problems.

Definition 2.2: function problem

A function problem (X, R) is a computational problem where each input has exactly one admissible output, i.e. R is a function $X \rightarrow Y$.

For example, sorting is a function problem as R_{isorta} is a function.

Definition 2.3: decision problem

A decision problem (X, R) is a function problem where R is a binary function $X \rightarrow \{\text{yes}, \text{no}\}$.

Conceptually, decision problems are problems whose statements take the form of yes-no questions, e.g. “is n prime?” (primality testing); and “is a given list l sorted in ascending order?” (order testing). Remark that each computational problem (X, R) has an associated decision problem (X, R') with $R'(x) = \text{yes} \iff |R(x)| > 0$, i.e. “does problem instance x have a solution?”.

2.2 Optimization Problems

We will now describe a class of computational problems which will be the focus of application-oriented aspects of this dissertation: “combinatorial optimization problems”. First, we define the more general class of global function optimization problems:

Definition 2.4: Global Optimization Problem (GOP, maximization variant)

In a global optimization problem, given a pair $\langle f, S' \rangle$, where f is a function $S \rightarrow \mathbb{R}$ and $S' \subseteq S$, we are to find an $s^* \in S'$ such that $f(s) \leq f(s^*), \forall s \in S'$.

Here, S' is commonly called the search space and f the objective function. An element of S' we will refer to as a *candidate* solution. Remark that we have defined the global optimization problem as a maximization problem. Any problem in which we would like to minimize a cost or loss function g over S' can be formulated as a maximization problem using the negated loss function as objective, i.e. $\langle -g, S' \rangle$ and vice versa. Beyond minimization, we distinguish numerous other variants of the GOP:

black box optimization problem where one is given a procedure computing $f(s)$, $\forall s \in S'$, but f itself is not known explicitly.

unconstrained optimization problem where $S' = S$.

convex minimization problem where f is a convex function and S' a convex set; e.g. unconstrained linear and quadratic optimization problems.

stochastic optimization problem where f is a stochastic function, i.e. $f(s)$ is a random variable and the objective is to maximize $\mathbf{E}[f(s)]$.

multi-objective optimization problem where there are multiple, possibly conflicting, objectives we would like to optimize, i.e. $f : S \rightarrow \mathbb{R}^m$ is multi-dimensional. If objectives conflict, f imposes a partial rather than a total order over S' , and solutions take the form of a set $S'_{\text{Pareto}} \subseteq S'$ (a.k.a. the Pareto front) of all non-dominated (a.k.a. Pareto optimal/efficient) candidate solutions.

search problem where $f : S' \rightarrow \{0, 1\}$. In this setting f is also known as the goal test.

combinatorial optimization problem where S' is finite.

Remark that sorting a sequence l can be formulated as a GOP, with $S' = P^l$ (finite) and $f(l') = \prod_{1 \leq j < |l|} [l'_j \leq l'_{j+1}]$ (binary). In Section 2.4, we will discuss the relations between these problems and problem (re)formulations in general.

2.2.1 Examples of Combinatorial Optimization Problems

In this section, we provide some examples of well-known combinatorial optimization problems. In particular, we describe six of the nine problems we considered in our experiments in Chapters 5, 6 and 7. The remaining three are described in Section 5.3.2.4, p. 180.

MAX-SAT problem: The MAX-SAT problem is the optimization variant of the well-known Boolean Satisfiability (SAT) search problem: Given a boolean formula from propositional logic, find an interpretation (model) under which this formula is true, i.e. satisfied.

Such formula F consists of a set of possibly negated variables (i.e. literals) that are combined using (\wedge, \vee) operators, e.g. $\neg A \wedge (B \vee C)$. An interpretation assigns a truth value (true, false) to each variable (A, B, C) and is said to satisfy F if and only if F is true under these assignments. For instance, $(A \rightarrow \text{false}, B \rightarrow \text{false}, C \rightarrow \text{true})$ satisfies the formula above, while $(A \rightarrow \text{true}, B \rightarrow \text{true}, C \rightarrow \text{true})$ does not. Note that not all formulas are satisfiable, e.g. $A \wedge \neg A$, giving rise to the decision problem variant: “is F satisfiable?”. Each boolean formula can be put in a conjunctive normal form (CNF), where F is written as a conjunction (\wedge) of clauses and a clause is a disjunction (\vee) of literals. For instance, consider the following formula $\neg(A \vee B) \vee C$ (not in CNF). This formula is equivalent to $(\neg A \vee C) \wedge (\neg B \vee C)$ (CNF), having two clauses $(\neg A \vee C)$ and $(\neg B \vee C)$. In the MAX-SAT problem, the search space consists of possible interpretations of a CNF formula and the objective is to find one minimizing the number of unsatisfied clauses.

Bin packing problem: In the bin packing problem, one is given objects of different sizes (i.e. pieces) which must be packed into a number of bins with fixed capacity V as to minimize the number of bins used. The search space consists of all possible assignments of pieces to bins such that no bin overflows (i.e. content exceeds capacity) and the objective is to minimize the number of bins in the assignment.

Personnel scheduling problem: In the personnel scheduling problem, one must decide at what times and on which days (i.e. shifts) each employee should work over a given planning period. The search space consists of possible assignments of employees to a subset of all shifts in the planning period (i.e. a roster). The objective function differs greatly between different scheduling problem instances and encodes the preferences of employer and employees [Brunner 2010].

Permutation flow shop problem: In the permutation flow shop problem, one is to find an order in which a set of n jobs $\{j_1, j_2, \dots, j_n\}$ are to be processed by m machines $[M_1, M_2, \dots, M_m]$ such that the time it takes to complete all jobs (i.e. the makespan) is minimized. Each job is first processed by machine M_1 , then machine M_2 and so on. The different machines work in parallel and each job has a processing time for each machine. Machine M_j can only start processing the i^{th} job when it finished processing the $i-1^{\text{th}}$ job in the sequence and after M_{j-1} finished processing it. It is therefore important to order the jobs as to minimize intermachine waiting times. The search space are all possible permutations of n jobs and the objective is minimizing the time it takes to complete all jobs, i.e. the end-time of the last job in the permutation on M_m .

Traveling salesman problem: In the Traveling Salesman Problem (TSP, [Jünger et al. 1995]), given a set of n cities and the distances (travel time) between each pair of cities, one is to find the shortest possible route that visits each city exactly once and returns to the origin city. The search space consists of all possible cycles that visit each city exactly once and the cost function, to be minimized, is the sum of the distances between every pair of adjacent cities in this cycle. TSP is equivalent to the problem of finding an Hamiltonian cycle of minimal length in a complete weighted graph [West et al. 2001]. We discriminate different subclasses based on the properties of the weights: In the symmetric TSP the weights are symmetric. In the metric TSP, weights also satisfy the triangle inequality. In the Euclidean TSP, weights correspond to Euclidean distances between cities located on a hyperplane. Instances of the TSP, as formulated above, appear in many practical settings [Lenstra and Kan 1975], such as planning, logistics, and the manufacturing of microchips.

Vehicle routing problem: In the Vehicle Routing Problem (VRP), we are to serve a number of customers with a fleet of vehicles. We are given a set of n customers $N = (c_1, c_2, \dots, c_n)$, their locations (l_1, l_2, \dots, l_n) , a depot location l_0 , as well as distances between all these locations. A route is a sequence of locations starting and ending with the depot location, e.g. $[l_0, l_8, l_2, l_0]$ is a route serving customer c_8 followed by customer c_2 . A candidate solution is a set of m routes, where each route r_i serves customers R_i and (R_1, R_2, \dots, R_m) is a partitioning of N , i.e. a set of routes such that every customer is served exactly once. The objective is to find a solution minimizing the number of vehicles used (m), the total distance driven or a combination thereof. A lot of variants exist, adding extra features and constraints, an overview can be found in [Laporte 1992]. An example is the Capacitated Vehicle Routing Problem (CVRP), where capacity constraints are added. Here, each customer is given a certain demand, the vehicles have a given capacity C , and as a vehicle can only be loaded at the depot, the sum of demands of customers served within one route must never exceed C .

2.3 Algorithms

Thus far, we have introduced and formalized the notion of problems. Here, the crux is that while a problem specifies *what* needs to be done, it does not specify *how* this can be achieved. The field of algorithms studies systematic methods for solving problems (efficiently). While most computer scientists will agree with the above statement, defining what exactly “a systematic problem-solving method” (i.e. an algorithm) entails is non-trivial. In fact, finding a fully satisfactory, formal definition, capturing the intuitive notion of an algorithm, is commonly regarded as being an open problem [Blass and Gurevich 2003].

For now,² we consider an algorithm to be an *effective procedure* solving some computational problem exactly.

Definition 2.5: effective procedure [Audi 2015]

An effective procedure specifies how to compute a function. It has a finite description. For any input, it specifies a finite sequence of instructions that can be followed rigorously, in the real physical world, without requiring any ingenuity from the agent (man or machine) executing them, to obtain the correct output.

Definition 2.6: solving computational problems exactly

We say that an effective procedure, computing a function $a : X \rightarrow Y \cup \{\perp\}$, solves a computational problem (X, R) if and only if the following holds $\forall x \in X$:

- 1) $a(x) = y \implies xRy$ (it only outputs admissible solutions)
- 2) $a(x) = \perp \implies R(x) = \emptyset$ (it outputs $\perp \notin Y$ only if no solution exists)

Conceptually, an effective procedure is said to solve a computational problem, if it specifies how to derive an output, satisfying the problem property, for any possible input. Remark that we use $\perp \notin Y$ as the symbol returned by the algorithm to indicate that the given problem instance x has no admissible solutions. Note that, under this definition, if an effective procedure solves (X, R) , it also solves any (X', R') satisfying

$$\forall x \in X' : (x \in X) \wedge (xRy \implies xR'y) \wedge (R(x) = \emptyset \implies R'(x) = \emptyset).$$

Let us make this more concrete by describing a simple factor finding and sorting algorithm:

Trial Division Factorization (Algorithm 1) is a simple algorithm for $(\mathbb{N}, R_{\text{pfactor}})$. It tries to divide n by all integers $1 < p \leq \sqrt{n}$ in ascending order, and returns the first integer for which division succeeds and \perp if it fails for all.

Selection Sort (Algorithm 2) is a simple algorithm for $(\mathbb{Z}^*, R_{\text{isorta}})$. It can be viewed as dividing the given sequence l into a sorted $(l_{1:i})$ and unsorted $(l_{i:|l|})$ part. Initially $i = 0$ (all unsorted). Each iteration, it determines the smallest element in the unsorted part and swaps it with the leftmost element thereof, i.e. puts it in order, and increments i . After n iterations the ordered part will be $l_{1:|l|}$ (all sorted).

Procedures for solving combinatorial optimization problems are described in Section 2.7.

²In Section 2.6, we will revisit this definition and discuss possible generalizations thereof.

Algorithm 1 Pseudocode for the trial division algorithm, finding the least (smallest) non-trivial prime factor of an integer.

```

1: procedure FINDFACTOR( $n$ )
2:    $p \leftarrow 2$ 
3:   while  $p * p \leq n$  do
4:     if  $n \% p = 0$  then
5:       return  $p$ 
6:     else
7:        $p \leftarrow p + 1$ 
8:     end if
9:   end while
10:  return  $\perp$ 
11: end procedure

```

Algorithm 2 Pseudocode for the selection sort algorithm

```

1: procedure SORT( $l$ )
2:   for  $i = 1 : n - 1$  do
3:      $k \leftarrow i$ 
4:     for  $j = i + 1 : n$  do
5:       if  $l_j < l_k$  then
6:          $k \leftarrow j$ 
7:       end if
8:     end for
9:     swap  $i^{th}$  and  $k^{th}$  element in  $l$ 
10:  end for
11:  return  $l$ 
12: end procedure

```

2.3.1 Formalization

2.3.1.1 A Historical Note: Decidability

While the conceptual notion dates back to antiquity [Cooke 2011], modern attempts to formalize algorithms (\sim effective procedures) and computable functions started in 1928 when David Hilbert asked for an algorithm that would take any proposition in first order logic as input and output whether that proposition is true or false. This decision problem, known as the *Entscheidungsproblem*, is well-defined, as any statement in first order logic is either true or false. In the following decade, numerous models of computation were proposed, most notably μ -recursive functions [Gödel 1931, Kleene 1936], λ -calculus [Church 1932] and Turing machines [Turing 1937]. While the formalisms vastly differ, all three models are known to be equivalent, i.e. they describe the same class of (Turing) computable functions. In 1936, both Alan Turing and Alonzo Church (independently) proved that the (binary) function David Hilbert asked to compute is *not* Turing computable. The Church-Turing thesis (CT) asserts that the intuitive notion of effectively calculability is captured by the formal notion of Turing computability, i.e. a function can be computed (by any mathematician or physical machine) if and *only if* it can be computed by a Turing machine. As effective calculability is ill-defined, this thesis cannot be proven formally. However, as any successive attempts to define more powerful models of computation were unsuccessful, the CT thesis is generally assumed to hold true and the Entscheidungsproblem, as such, to be unsolvable. Computability theory is a branch of theory of computation that studies whether or not computational problems are solvable.

Definition 2.7: unsolvability (undecidability)

We say that a computational problem (X, R) is unsolvable (undecidable^a), if no (Turing) computable function a , satisfies (1) and (2) of Definition 2.6, $\forall x \in X$.

^aWhile, in its most literal sense, only applicable to decision problems, we use this term more flexibly in this dissertation to denote unsolvability of any computational problem.

2.3.1.2 Turing Machines

Now, we briefly introduce the Turing Machine (TM) model of computation. Remark that understanding the specifics of this formalism (and later extensions) is only required to fully appreciate our formalization of the dynamic algorithm selection problem in Section 4.3.1. Conceptually, a TM is an abstract machine that computes the value of a function.³ It does so by iteratively scanning/writing symbols from/to an infinite one-dimensional tape. This

³To be fully correct, as we will discuss at the end of this section, *not all TMs compute a function*.

tape is subdivided in cells, each containing a single symbol (or a blank). Initially, the tape contains only the input. Each time step, it either (1) scans and modifies the content of the single cell, positioned under the machine's single scanning/writing head, before moving this head to one of the adjacent cells; or (2) halts. When it halts, the content of the tape is interpreted as its output. Its behavior on an input is uniquely determined by its finite-state controller. In what follows, we formally define TMs and their operation.

Definition 2.8: Turing Machine (TM) [Hopcroft et al. 2006, pp. 327]

We define a TM as a 7-tuple^a $\langle Q, q_0, F, \Gamma, B, \Sigma, \delta \rangle$:

Q is a finite, non-empty set of *control states*.

$q_0 \in Q$ is the *start state*.

$F \subseteq Q$ is the set of final or *accepting states*.

Γ is a finite, non-empty set of *tape alphabet* symbols.

B is the *blank* symbol.

$\Sigma \subseteq \Gamma \setminus \{B\}$ is the set of *input alphabet* symbols.

δ is the *transition function*: a partial function $(Q \setminus F) \times \Gamma \rightarrow (Q \times \Gamma \times \{R, L\})$.

The arguments of $\delta(q, a)$ are the current control state q and tape symbol being scanned a . The value of $\delta(q, a)$, if defined, is a triple (p, b, d) , where

$p \in Q$ is the next state.

$b \in \Gamma$ is the symbol written in the cell being scanned, i.e. replacing a .

d is the direction in which the head moves, either L (left) or R (right).

^aWe took the liberty to reorder some of the elements as to enable a more concise definition.

The operation of a TM: We further formalize the operation of a TM. To this end, we first introduce some notation to describe its full state or configuration. We use $\alpha qa\beta$ to denote the Instantaneous Description (ID) of a TM in control state q , with tape content $\alpha qa\beta$, head pointing at symbol a . Here, α and β abstract a prefix and suffix of zero, one or more tape symbols, proceeded and followed by infinite blank symbols respectively.

A TM simulated on x starts in ID q_0x , i.e. control in the start state and head pointing at the cell containing the leftmost symbol of the input string. Let *step* be an auxiliary

function describing the ID resulting of performing a given transition in a given ID.

$$\text{step}(\alpha a q b c \beta, (q', b', d)) = \begin{cases} \alpha q' a b' c \beta & (d = L) \\ \alpha a b' q' c \beta & (d = R) \end{cases}$$

In ID $\alpha q b \beta$, the TM will move to ID $\text{step}(\alpha q b \beta, \delta(q, b))$, if $\delta(s, b)$ is defined, and will halt otherwise. Let us denote a sequence of n transitions by triples $(q, b, d) \in Q^n \times \Gamma^n \times \{R, L\}^n$, where (q_i, b_i, d_i) is the i^{th} transition. Let steps be an auxiliary function describing the ID resulting of performing a sequence of transitions starting from a given ID.

$$\text{steps}(id, (q, b, d)) = \begin{cases} id & (\text{if } |q| = 0) \\ \text{steps}(\text{step}(id, (q_1, b_1, d_1)), (q_{2:|q|}, b_{2:|b|}, d_{2:|d|})) & (\text{otherwise}) \end{cases}$$

Definition 2.9: execution of a TM on an input

We define the execution of a TM M on an input x , denoted e_x^M , as a quadruple $(a, q, b, d) \in \Gamma^* \times Q^* \times \Gamma^* \times \{R, L\}^*$ satisfying the following conditions:

1. $|a| = |q| = |b| = |d|$.
2. $(q_i, b_i, d_i) = \delta(q_{i-1}, a_i), \forall 1 \leq i \leq |a|$.
3. $\alpha q_i a_{i+1} \beta = \text{steps}(q_0 x, (q_{1:i}, b_{1:i}, d_{1:i})), \forall 0 \leq i < |a|$.
4. If $|q|$ is finite, $\alpha q_{|q|} c \beta = \text{steps}(q_0 x, (q, b, d))$ with $q_{|q|} \in F$.

Conceptually, a_i is the i^{th} tape symbol scanned and (q_i, b_i, d_i) the i^{th} transition performed when simulating M on x . Remark that (q, b, d) uniquely determines e_x^M , i.e. a is redundant. Therefore, when convenient, we will treat executions as “sequences of transitions”; e.g. we use $\text{steps}(q_0 x, e_x^M)$ to denote the ID in which TM M halts when simulated on x .

The function computed by a TM: A TM M can be viewed as describing an effective procedure computing a function $M : X \rightarrow Y \cup \{\perp\}$

$$M(x) = \begin{cases} \alpha \beta & \text{If } \alpha p \beta = \text{steps}(q_0 x, e_x^M) \text{ and } p \in F. \\ \perp & \text{Otherwise } (p \notin F). \end{cases}$$

Remark that a TM does not halt on every input x by design, i.e. it can model indefinite calculations. A TM which does halt $\forall x \in \Sigma^*$ is called *total*. Note that TMs that do not halt for x , do not return an output for x and do not solve problems with $x \in X$. As a side note, the decision problem “Does a given TM M halt on a given input x ?”, a.k.a. the *halting problem* is undecidable.

Digression: How common are undecidable problems? The existence of unsolvable problems raises the question of how prevalent such problems are. Using simple combinatorics it can be shown that while there are uncountably infinite decision problems, there are only countably infinite TMs, each solving at most one of these decision problems. Hence, there must be uncountably many more undecidable than decidable problems, i.e. if we were to pick an arbitrary problem, the likelihood of it being decidable would be infinitesimal. This result is counter-intuitive as most well-defined problems we face in practice (e.g. sorting, factorization, combinatorial optimization etc.) are decidable. The only logical explanation is that these problems are somehow not arbitrary, i.e. there exists a bias towards encountering solvable problems.

2.3.2 Computer Programs

As discussed in Section 1.1, *automation* is one of the key incentives for formalizing problem-solving methods as algorithms. Arguably, the popularity of the TM formalism, relative to the alternatives mentioned earlier, can be largely attributed to its “mechanical” nature, i.e. it is relatively straightforward to imagine (and build) a physical machine that simulates a given TM. However, unlike TMs, modern computers are not designed to solve any particular problem, rather they can be *programmed* to solve any (decidable) problem. Foundational to general purpose computers is the Universal Turing Machine (UTM), described by Alan Turing alongside his TM formalism in [Turing 1937]. The UTM is a TM which takes as input a description of an arbitrary TM M and an arbitrary input string x , i.e. $\langle M, x \rangle$, and returns $M(x)$. Conceptually, the UTM embodies the idea that rather than building a machine executing a fixed algorithm, we can build a machine (hardware) which can execute any algorithm, given a logical description thereof (software).

Definition 2.10: computer program

A description of an algorithm, in a way it can be understood by an UTM.

The act of describing algorithms as such is known as *programming*. The wording used by a *programmer* to do so, is called a *programming language*. The first programming languages were often machine specific, making it difficult to port programs across devices, e.g. requiring manual translation. In addition, manually specifying every low-level instruction to be performed by a computer was tedious and error-prone. To improve portability and programming convenience, high-level programming languages were developed. High-level programming languages can be translated automatically to their machine-specific low-level variants, using programs called compilers or interpreters. To communicate algorithms to humans, often an ad hoc, informal, programming language-like notation is used, known as

pseudocode (e.g. as we used in Algorithms 1 and 2). While pseudocode does not permit the same level of formality as e.g. TM notations, descriptions therein are more concise, require less effort to write and typically suffice to communicate broad algorithmic concepts.

2.3.3 Parametrized Algorithms

Some algorithms, next to a problem instance, also take one or more parameter values (e.g. command-line arguments) as input. We will use the terms parametrized algorithm or algorithm framework, to refer to such algorithms which are “programmable” themselves.

Following the terminology and notation of [Hutter et al. 2009], we use p_1, \dots, p_k to denote the k configurable parameters of an algorithm. Each parameter p_i can take one of a set of possible values Θ_i , called the domain of p_i and a configuration θ assigns a value $\theta_i \in \Theta_i$ to each parameter p_i . Not all configurations may be allowed, and $\Theta \subseteq \Theta_1 \times \dots \times \Theta_k$, denotes the set of valid configurations, called the *configuration space*. Remark that a parametrized algorithm can be viewed as specifying a family of parameterless algorithms, each configuration $\theta \in \Theta$ corresponding to a single algorithm instance.

We can distinguish between different types of parameters based on their domain, e.g.

categorical parameters can take on a finite set of nominal values, without natural relations between their values (i.e. Θ_i is an unstructured set).

ordinal parameters are similar to categorical parameters, but differ in that their values are ordered, i.e. one value is smaller/larger than another.

numerical parameters are integer-valued (discrete) or real-valued (continuous).

procedural parameters can take an executable subroutine, i.e. a program, as value.

Why parametrize programs/algorithms? Often the (best) choice of configuration (\sim algorithm instance) is use case dependent (see Section 3.2). As such, “hard-coding” these values would limit the reusability of the program/algorithm. Parameters may be viewed as design choices left open at *design time*, to be made at *use time*, where more information about the specific use case is available, to increase the reusability of the algorithmic framework. However, this comes at the cost of ease of use, as the user must now configure the algorithm. The latter can be ameliorated by providing one (or more) default configuration(s) (for each use case), or by using automated parameter tuning software (e.g. SMAC [Hutter et al. 2011], see Section 3.3.3, p. 90). In Section 2.7.2, we will see many examples of algorithmic frameworks in the context of solving the combinatorial optimization problem. Note that a UTM may be viewed as an extreme case of an algorithmic framework taking an arbitrary effective procedure as a single procedural parameter.

2.4 Problem Formulation

In this Section, we will discuss problem formulation, i.e. the act of expressing problems in a form such that they can be solved automatically, using computers. Problems we encounter “in the wild” rarely correspond to well-defined computational problems, rather they are inexact notions. However, many can be usefully formulated as such. The act of doing so, we will refer to as problem formalization. Thus far, we have formulated computational problems in terms of mathematical objects (e.g. ‘list’, ‘integer’, etc.) and we will continue to do so in the remainder of this dissertation. However, in this section, we wish to stress the fact that computers operate on *specific representations* of these objects. Therefore, when faced with some problem we would like a computer to solve, we must decide in which form to present problem instances as inputs to the computer and the format we expect the output to have. The form these representations can take clearly depends on the capabilities of the model of computation. For instance, a TM computes a function $\Sigma^* \rightarrow \Gamma^* \cup \{\perp\}$, and therefore to solve a computational problem (X, R) , we must encode inputs and outputs as elements in Σ^* and Γ^* , respectively.⁴ i.e. we must reformulate it as a computational problem (Σ^*, R) with $R \subseteq \Sigma^* \times \Gamma^*$.

2.4.1 Reducibility

Thus far, we have encountered various computational problems. However, not all of these are unrelated. Computational problems and their properties are one of the main subjects of study in theory of computation. Here, the branch of computability theory studies whether or not a problem can be solved using a computer (i.e. whether an algorithm exists), while the branch of complexity theory investigates the resources required for doing so (i.e. whether an efficient algorithm exists). In Section 2.3 we already introduced some core concepts in computability theory, in the Section 2.5 we will do so for complexity theory. A fundamental relationship between problems exploited in both is the notion of “reducibility”.

Definition 2.11: Turing reducibility

A computational problem R is (Turing) reducible to another R' , denoted $R \leq_T R'$, if any solver for R' can be used as a subroutine (a.k.a. an oracle) to solve R . Formally, $R \leq_T R'$ if and only if there exists an R' oracle machine^a solving R .

^aA generalization of, and operating in a similar way as, an ordinary TM defined in 2.8, but extended with the ability to query an oracle for R' .

⁴Commonly, a digital model of computation is considered, i.e. $\Sigma = \Gamma \setminus \{B\} = \{0, 1\}$.

For instance, the prime factor finding and sorting problems are Turing reducible to the primality and order testing problems, respectively. The corresponding oracle machines would enumerate all candidate solutions ($S' = \{z \in \mathbb{Z}^+ \mid z \leq \sqrt{n}\}$ and $S' = P^l$), apply the respective tests until one succeeds and return that solution.⁵ Note that $R \leq_T R'$ does not require R' to be solvable. However, if R' is solvable, we can solve R as well. Or equivalently, if R cannot be solved then R' cannot be either. The latter is how reducibility is most commonly used in computability theory to prove (by contradiction) that some problem R' cannot be solved: We take a problem R known to be undecidable and show $R \leq_T R'$. In fact, this is how Alan Turing proved the undecidability of the Entscheidungsproblem R_{decide} in [Turing 1937]. He first showed the halting problem R_{halt} to be undecidable and subsequently $R_{\text{halt}} \leq_T R_{\text{decide}}$.

Another form of reducibility, of particular interest in the context of problem formulation and complexity theory, is many-one reducibility:

Definition 2.12: many-one reducibility (for decision problems)

A decision problem (X, R) is many-one reducible, short m -reducible, to a decision problem (X', R') , denoted $(X, R) \leq_m (X', R')$, if and only if there exists a computable function $f : X \rightarrow X'$, called a reduction, satisfying

$$\forall x \in X : R(x) = \text{yes} \iff R'(f(x)) = \text{yes}.$$

Conceptually, m -reductions map instances of one problem to those of another, capturing the intuitive notion of problem reformulation. Remark that m -reducibility is only defined for decision problems. However, this notion can be generalized

Definition 2.13: many-one reducibility (for computational problems)
[Papadimitriou 1994, p. 506]

Let (X, R) and (X', R') be computational problems, with $R : X \times Y$ and $R' : X' \times Y'$. We say that $(X, R) \leq_m (X', R')$ if and only if there exist computable functions $f : X \rightarrow X' \cup \{\tau\}$ (formulation) and $g : X \times Y' \rightarrow Y$ (interpretation) such that $\forall x \in X$:

1. $\forall y' \in R'(f(x)) : g(x, y') \in R(x)$.
2. $R(x) = \emptyset \iff R'(f(x)) = \emptyset \vee f(x) = \perp$.

Note that a reduction is now defined as a pair of functions (f, g) to account for the fact that the solution (representation) space of two computational problems might differ.

⁵This oracle machine is a brute-force algorithm for combinatorial search (see also Section 2.7.1).

To make this more concrete, we show that m -reducibility may be viewed as a special case of Turing reducibility (restricting the use of the oracle to a single application):

Theorem 2.1

For any two computational problems (X, R) and (X', R') :

$$(X, R) \leq_m (X', R') \implies (X, R) \leq_T (X', R')$$

Proof. Let (f, g) be the pair of functions reducing (X, R) to (X', R') , we can construct an oracle machine for (X, R) as follows: Given any $x \in X$.

1. use f to formulate x as $x' = f(x)$ an instance of R' (if $f(x) = \perp$, return \perp).
2. use oracle a' , a solver for (X', R') , to obtain $y' = a(x')$, the solution of x' .
3. use g to interpret y' as $y = g(x, y')$ the solution of x .

The diagram in Figure 2.1 depicts the operation of this machine. □

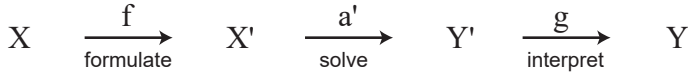


Figure 2.1: Solving problems by reformulation (*reduction diagram*)

For instance, order and primality testing are m -reducible to the sorting and prime factorization problems, respectively. Any NP-complete problem (e.g. SAT) (see Section 2.5.2) can be m -reduced to any of the six combinatorial optimization problems listed in Section 2.2.1. Remark that both types of reducibility relate computational problems in a way that is reflexive ($R \leq R$) and transitive ($R \leq R' \leq R'' \implies R \leq R''$), i.e. they induce a preorder. A symmetric preorder is an equivalence relation. If $R \leq_T R'$ and $R' \leq_T R$ we call R and R' computationally equivalent, denoted $R \equiv_T R'$, i.e. R is solvable if and only if R' is. Examples are order testing and sorting; primality testing and prime factorization. Remark that this does NOT imply that both can be solved equally efficiently. Similarly, if $R \leq_m R'$ and $R' \leq_m R$ we call R and R' m -equivalent, denoted $R \equiv_m R'$ and by Theorem 2.1: $R \equiv_m R' \implies R \equiv_T R'$. Examples of m -equivalent problems are:

- Sorting a sequence in ascending or descending order.
- The minimization and maximization variants of an optimization problem.
- The SAT and MAX-SAT problems.

Many-one reductions are of particular interest in complexity theory because most well-studied complexity classes (including P and NP, see Section 2.5) are closed under some type of m -reducibility.⁶ A special case of m -reducibility occurs when any algorithm solving (X', R') also solves (X, R) (see also Section 2.3), i.e. no reformulation is required.

Definition 2.14: problem generalization/specialization

We say that a problem (X', R') is a generalization of a problem (X, R) , or equivalently, that (X, R) is a specialization of (X', R') , denoted $(X, R) \subseteq (X', R')$, if and only if $(X, R) \leq_m (X', R')$ with $f(x) = x$ and $g(x, y) = y$.

Note that $R \subseteq R' \wedge R' \subseteq R \iff R = R'$. In this dissertation, we will use the terms specialization/generalization somewhat more flexibly, allowing trivial (computationally cheap and straightforward) reductions. Also, if we refer to “the problem an algorithm a solves” (outside the context of any particular problem) we generally refer to the most general problem it solves, i.e. $(X, R) : a \text{ solves } (X', R') \implies (X', R') \subseteq (X, R)$. For instance, selection sort (Algorithm 2) sorts a sequence of ordinals, which is a generalization of the problem of sorting a sequence of integers. Trial division (Algorithm 1) finds the least (smallest) prime factor, which is a generalization of the problem of finding a non-trivial prime factor. The convex, unconstrained, combinatorial optimization and search problems are specializations of the GOP, and the stochastic and multi-objective variants are generalizations thereof. Also, Euclidean TSP \subseteq symmetric TSP \subseteq TSP \subseteq VRP \subseteq CVRP.

2.4.2 Problem-solving vs. Problem Formulation

In practice, the form in which we present a problem to a computer will affect how efficiently it can be solved. This suggests that there is a fine line between formulating a problem and actually solving it.

“A problem well put is half solved.” - John Dewey

In what follows, we will formalize the close relationship between problem-solving and problem formulation. To this end, we first define the class of trivial problems:

Definition 2.15: trivial problem

A problem (X, R) is trivial if $\forall x \in X : xRx$, i.e. R is the identity function.

Note that solving a trivial problem requires no computation, i.e. it is effectively solved. Next, we show that the following holds:

⁶with appropriate resource restrictions on (f, g) .

Theorem 2.2

For a given computational problem (X, R) the following two statements are equivalent, i.e. $(1) \iff (2)$:

1. (X, R) is solvable (see Definition 2.7)
2. (X, R) is m -reducible to a trivial problem (as in Definition 2.15).

Proof.

- (1) \implies (2):** Let a be a solver for (X, R) . (X, R) can be trivialized using a reduction (f, g) , where $\forall x \in X : f(x) = a(x) = y$ and $g(x, y) = y$.
- (2) \implies (1):** Let (f, g) be a reduction from (X, R) to a trivial problem. We can construct an algorithm for (X, R) as follows: Given any instance $x \in X$:
1. compute $f(x)$. If $f(x) = \perp$, return \perp . Otherwise, continue.
 2. compute and return $y = g(x, f(x))$.

Intuitively, each well-defined problem implicitly encodes its solution, and computation merely decodes it [Goldreich 2008]. \square

2.4.3 Opinion: Importance of Natural Formulations

In what follows, we argue that in formulating a problem, one's primary concern should always be: accurately capturing/modeling the *actual* problem. In particular, it should be trivial to represent any natural occurrences of problem instances and their solutions, i.e. take as little as possible manual effort or expert knowledge (*natural*). This activity is orthogonal to *how* one plans to solve it and should avoid any implicit design decisions inducing a bias towards any particular solution approach (*declarative*). When contemplating alternative formulations, one should consider possible m -reductions amongst them, and search one (efficiently) reducible to all alternatives considered, i.e. maximizing possible solution approaches (*reducible*). Clearly, this natural formulation will be very specific to one's particular problem setting and therefore it is relatively unlikely that machines/programs solving the problem, in this form, out-of-the-box, exist. However, this is no reason for concern, as either one would have been unable to formulate it as such in the first place, or we know an m -reduction for doing so. In the latter case, we can describe an effective procedure for computing this m -reduction allowing it to be performed automatically. Remark that this gives rise to a perspective, where the solution approach as a whole is viewed as solving the actual problem by reduction to the problem solved by the algorithm used.

We claim that there are many benefits associated with this “problem-centric” perspective. First, explicitly formalizing “the problem we are actually trying to solve”, in such manner, enables us to study the actual problem theoretically (e.g. computability, complexity). It also makes design decisions made in further reformulations of the problem explicit, e.g. allowing us to distinguish between limitations inherent to the problem and those of a particular solution approach. As such, it also counteracts the law of the instrument [Maslow 1966, p. 15], a cognitive bias, more commonly known as the “hammer looking for a nail” phenomenon, which we believe to be ubiquitous and a major cause of fragmentation in algorithmic research. Explicitly acknowledging the fact that we are trying to solve the same problem, encourages comparative studies, evaluating quality w.r.t. solving the actual problem, knowledge transfer across communities and contributes towards a more unified solution approach, maximally exploiting the nature of the problem at hand. Note that naturality (\sim capturing all features of naturally occurring instances of a problem) is not the same as generality (\sim capturing as many as possible problem instances). While generality is a desirable feature for algorithms, we argue that it is a false incentive for problem formulations. Overly general formulations have a tendency of abstracting information which could otherwise be usefully exploited in solving the problem, i.e. they may ‘artificially’ restrict how efficiently it can be solved (have a lower reducibility).

2.5 Computational Complexity

Thus far, we have discussed computational problems and the use of algorithms to solve them (automatically). In particular, we examined *computability*: Whether a problem is solvable, and if so, whether a given algorithm solves it. Under the CT, undecidable problems cannot be solved in practice. However, the inverse is often not the case as resources in our physical world are limited, while theoretical models of computation (e.g. Turing machines) assume unbounded resources (e.g. infinite tape). Put differently, while algorithms solving some problem may exist, actually executing them may require a finite, yet intractable amount of resources. Computational complexity theory is a branch of theory of computation that studies how difficult computational problems are to solve and classifies them according to their “complexity”, i.e. the resources required to solve them.

2.5.1 Analysis of Algorithms

Analysis of algorithms is concerned with the study of the resources a given algorithm requires to solve a given problem. In what follows, we will focus on time complexity.⁷

⁷Space complexity is bounded by time complexity. Intuitively, because it takes time to use space.

Definition 2.16: worst-case time complexity

Let t_x^a denote the time it takes an algorithm a to solve a problem instance x of a computational problem $p = (X, R)$. Let $\|\cdot\|$ be a measure of the size of inputs. Let $X_n = \{x \in X \mid \|x\| = n\}$ be the set of inputs of size n . The worst-case time complexity of a on problem p is given by $t_p^a(n) = \max_{x \in X_n} t_x^a$

Remark that computational complexity is expressed as a function of the size of inputs, as the resources required naturally grow as a function of input size. For example, a measure of the size for the sorting problem could be the number of elements in the sequence l . Alternative measures could be used, hence computational complexity depends on our choice thereof. In Section 2.3 we mentioned that “what can be computed” does not depend on the model of computation used. However, “the time a computation takes” does. Hence, computational complexity must be studied in the context of a model of computation. To avoid that time complexity becomes dependent on details of the choice of model (or measure of input size), and as it is often difficult to determine $t_p^a(n)$ exactly for all possible n , asymptotic complexity bounds are often used instead in theoretical analysis, commonly expressed using Bachmann-Landau notations:

Definition 2.17: Big O(micron), Omega and Theta notations [Knuth 1976]

Let $f(n)$ and $g(n)$ be two functions. One writes

$f(n) \in \mathcal{O}(g(n))$ if and only if there exist positive constants C and n_0 such that $\forall n \geq n_0 : f(n) \leq C * g(n)$ holds.

$f(n) \in \Omega(g(n))$ if and only if there exist positive constants C and n_0 such that $\forall n \geq n_0 : C * g(n) \leq f(n)$ holds.

$f(n) \in \Theta(g(n))$ if and only if $f(n) \in \mathcal{O}(g(n)) \wedge f(n) \in \Omega(g(n))$ holds.

For example, assuming comparing and swapping integers takes constant time, i.e. is $\Theta(1)$, selection sort (see Algorithm 2) has a worst case time complexity of $\mathcal{O}(n^2)$ for sorting arbitrary sequences of integers. As the number of comparisons performed is independent of the values of elements, it is also $\Omega(n^2)$ and therefore $\Theta(n^2)$. Assuming the modulo operator is $\Theta(1)$, trial division (see Algorithm 1) takes $\mathcal{O}(2^n)$ time to find a factor for arbitrary integers,⁸ with n the number of digits used in the positional (e.g. binary) representation

⁸Worst case complexity occurs when given a prime number.

of the integer. However, it is $\Omega(1)$ as it finds a factor of an even number (i.e. 2) in a single iteration, independent of the size of that number.

2.5.2 Complexity of Problems

Complexity theory is concerned with the complexity of problems, rather than that of any specific algorithm. Conceptually, the complexity of a problem is defined as the complexity of the most efficient algorithm solving it. For example, under aforementioned assumptions, the complexity of sorting arbitrary sequences of ordinal elements is $\mathcal{O}(n \log(n))$. Complexity theory aims to classify computational problems w.r.t. their complexity. Two fundamental classes for decision problems are P and NP.

2.5.2.1 P

Definition 2.18: P (complexity class)

P is the set of all decision problems decidable in polynomial time by a TM (see Definition 2.8), i.e. \exists TM M deciding $(X, R) : t_{(X,R)}^M(n) \in \mathcal{O}(n^k)$, for some finite k .

Sorting is in P.⁹ The primality testing problem, long believed to be outside of P, was shown to be in P as well [Agrawal et al. 2004]. In complexity theory, polynomial-time algorithms are commonly regarded as those that can be executed efficiently in practice (also known as Cobham's thesis [Goldreich 2008]). It turns out that in the definition of P, the TM could be replaced by any contemporary practical, universal, deterministic model of computation and P would remain invariant.

2.5.2.2 NP

Before defining NP, we first introduce a generalization of the TM model of computation:

Definition 2.19: non-deterministic Turing Machine (NTM)

A non-deterministic Turing machine is defined as a 7-tuple $\langle Q, q_0, F, \Gamma, B, \Sigma, \delta \rangle$, with $Q, q_0, F, \Gamma, B, \Sigma$ as in Definition 2.8 and δ is the *transition relation*, a binary relation $((Q \setminus F) \times \Gamma) \rightarrow 2^{(Q \times \Gamma \times \{R, L\})}$, where $\delta(q, a)$ specifies the set of possible moves the NTM can perform in a non-final control state q on scanning tape symbol a .

⁹Technically incorrect, as P by definition only contains decision problems.

A NTM operates as follows: When $|\delta(q, a)| \leq 1$ a NTM behaves as an ordinary TM. There are two equivalent perspectives on how a NTM resolves $|\delta(q, a)| > 1$:

- A NTM performs these in parallel, as soon as any one of the possible computations halts, the NTM halts and returns the result of that computation.
- A NTM performs the computation requiring the least time, i.e. the least moves.

From this definition, it should be clear that while it is relatively easy to conceive a TM, whether or not it is possible to physically construct an arbitrary NTM, remains an open question [Aaronson 2005]. Now, NP is defined as follows:

Definition 2.20: NP (complexity class)

NP is the set of all decision problems decidable in polynomial time by an NTM.

From a computability perspective, a NTM is not more powerful than a TM, i.e. both models describe the same set of computable functions. However, whether they describe the same set of efficiently computable functions, i.e. $NP = P$, is one of the most important open problems in mathematics and computer science [Fortnow 2009]. It is easily shown that $P \subseteq NP$, since a TM is a special case of a NTM where the relation δ is a partial function. However, the proposition $NP \subseteq P$ has thus far famously eluded formal (dis)proof. This question is of interest, because there are numerous important problems in NP, for which no polynomial-time algorithm (TM) is known (yet). Generally, $P \subset NP$ is assumed, implying that there are problems in NP that cannot be solved efficiently using contemporary deterministic computation technology.

2.5.2.3 NP-hard

A subclass of problems that are particularly difficult to solve are NP-complete problems. Informally, NP-complete contains the set of problems which are “at least as difficult as” any other in NP. To define this subclass, we must first formalize this relation:

Definition 2.21: efficiently reducible (“at least as difficult as”)

A problem p is efficiently reducible^a to another p' , denoted $p \leq_m^P p'$, iff there exists a pair of polynomial-time computable functions (f, g) that m -reduce p to p' .

^aIn the context of decision problems also known as Karp reducible.

Now, we define NP-complete as:

Definition 2.22: NP-complete (complexity class)

A problem p is NP-complete, if $p \in \text{NP}$ and $p' \leq_m^P p, \forall p' \in \text{NP}$.

The first problem found to be NP-complete was SAT (see Section 2.2.1) [Cook 1971]. Given an NP-complete problem p (e.g. SAT), it is sufficient to show p to be efficiently reducible to p' , in order to show that p' , too, is NP-complete. This way numerous problems have been shown to be NP-complete [Karp 1972]. If for any one of these a polynomial-time algorithm were to be found, this would imply $\text{NP} = \text{P}$. As mentioned before, the notion of NP-completeness is only defined for decision problems in NP. A broader notion which applies to computational problems in general is NP-hardness:

Definition 2.23: NP-hard (complexity class)

A problem p is NP-hard, if $p' \leq_m^P p, \forall p' \in \text{NP}$.

Thus NP-hardness drops the requirement that a problem should be in NP. Note that NP-hard, under this definition, includes undecidable problems. As SAT can be m -reduced to MAX-SAT in $\Theta(1)$, the latter is NP-hard. In fact, all six combinatorial optimization problems discussed in Section 2.2.1 are known to be NP-hard.¹⁰ For these problems, under $\text{P} \neq \text{NP}$, we will worst case have to evaluate a super-polynomial (e.g. exponential $\mathcal{O}(2^n)$, factorial $\mathcal{O}(n!)$, etc.) number of candidate solutions, rendering relatively small instances already intractable in practice. As we will discuss in Section 2.6.3, a practical way to deal with NP-hard problems is to not actually solve them in the traditional sense (Definition 2.6), but rather approximate the solution to the problem.

2.6 Beyond Effective Procedures

In practice, problems are often being solved using methods which do not correspond to algorithms as we have defined them in Section 2.3 (\sim effective procedures, Definition 2.5). For example, they may (with some likelihood) output a solution which is not admissible, fail to return a solution even though one exists, or require the execution of an infinite number of instructions. Why would we use such methods? As we have seen, for many interesting problems, no (efficient) algorithms are known, or may exist, e.g. they are undecidable (Definition 2.7) or NP-hard (Definition 2.23). The general workaround to “solve” such problems, in practice, is to extend our notion of what exactly constitutes an algorithm,

¹⁰Remark that whether prime factorization is NP-hard is an important open question. RSA, a common cryptographic system, critically relies on the difficulty of this problem.

beyond effective procedures computing a deterministic function. We will still use the term “algorithm” to refer to this wider notion of problem-solving procedures/methods, combined with the adjective “exact” to refer to those satisfying the more narrow definition. In what follows, we will briefly characterize different kinds of problem-solving procedures which are not “effective” in the traditional sense.

2.6.1 Randomized Algorithms

Many problem-solving procedures incorporate randomness as part of their logic, i.e. their execution requires the agent to flip a coin, roll a dice etc. To model such procedures, we introduce the following generalization of the TM model of computation:

Definition 2.24: Probabilistic Turing Machine (PTM, [Santos 1969])

A probabilistic Turing machine is defined as a 7-tuple $\langle Q, q_0, F, \Gamma, B, \Sigma, \delta \rangle$, with $Q, q_0, F, \Gamma, B, \Sigma$ as in Definition 2.8 and δ the *transition probabilities*, $((Q \setminus F) \times \Gamma) \times (Q \times \Gamma \times \{R, L\}) \rightarrow [0, 1]$, where $\delta((q, a), (p, b, d))$ specifies the likelihood of performing a transition (p, b, d) in a non-final control state q , on scanning tape symbol a .

Recall that deterministic algorithms M uniquely determine the sequence of instructions to be executed for any given input x (e_x^M , see Definition 2.9). The same does not hold for a randomized algorithms.

Definition 2.25: execution of a PTM on an input

A PTM M , for each input x , specifies a distribution over a set of possible executions E_x^M where each $(a, q, b, d) \in E_x^M$ satisfies conditions (1), (3) and (4) from Definition 2.9 and

$$2. \delta(q_{i-1}, a_i, (q_i, b_i, d_i)) > 0, \forall 1 \leq i \leq |q|.$$

The likelihood that simulating M on input x results in execution $e = (a, q, b, d) \in E_x^M$, denoted $\text{pr}(e|M, x)$, is given by

$$\text{pr}(e|M, x) = \prod_{i=1}^{|q|} \delta((q_{i-1}, a_i), (q_i, b_i, d_i)). \quad (2.1)$$

In general, a halting PTM M computes a stochastic function, i.e. $M(x)$ is a random variable, with $\text{Pr}(M(x) = y) = \sum_{e \in E_x^M} \text{pr}(e|M, x) [\text{steps}(q_0 x, e) = \alpha p \beta \wedge y = \alpha \beta]$.

However, a randomized algorithm may still compute a deterministic function, i.e. solve a problem according to Definition 2.6. An example is the quicksort algorithm with random pivot selection. Also, a TM is a special case of a PTM where δ is a degenerate distribution. In the remainder of this subsection, we will focus on randomized algorithms which are exact in this sense, and postpone the discussion of various non-exact procedures to later subsections. Obviously, as the instructions executed by a PTM vary, so will the time it takes to do so. We define the time complexity of PTMs accordingly:

Definition 2.26: expected (worst-case) time complexity

Let $t(e)$ denote the time it takes to perform an execution. Let $X_n \subset X$ be the set of inputs of size n . The expected (worst-case) time complexity of PTM M on problem p is given by $t_p^M(n) = \max_{x \in X_n} \sum_{e \in E_x^M} \text{pr}(e|M, x) * t(e)$.

Note that quicksort with deterministic pivot selection has a time complexity of $\mathcal{O}(n^2)$, while its variant with random pivot selection has an expected time complexity of $\Theta(n \log(n))$. A question central to the theoretical study of randomized algorithms, is whether a PTM is more powerful than a TM. From a computability perspective, the answer is clearly no, in the sense that any deterministic function computable using a PTM, can also be computed by a TM. Whether they are equivalent from a computational complexity perspective is an open question. An orthogonal open problem is whether a PTM can actually be realized in practice, as doing so would require access to a source of *true randomness* [Stipčević and Koç 2014], something which may not exist in our physical universe. In practice, “randomized” algorithms most commonly base their execution on pseudo-random numbers, generated using deterministic procedures known as pseudo-random number generators. Others inherit their stochastic behavior from interactions with the execution environment.

2.6.2 Asymptotically Correct Algorithms

A procedure, to be effective, must return the correct answer *after a finite number of steps* (see Definition 2.5). Asymptotic correctness relaxes this constraint, allowing procedures to only return the correct answer after a possibly infinite number of steps:

Definition 2.27: asymptotically correct

Let (X, R) be a function problem. Let h be a bounded function $\Gamma^* \times \Gamma^* \rightarrow \mathbb{R}$ satisfying $h(x, y) = 0 \iff xRy$ and $h(x, y) > 0$ otherwise. Let y_n be the content of the tape after executing a PTM for n steps.^a

We say that a PTM is asymptotically correct, w.r.t. (X, R) and h , if the following holds $\forall x \in X$:

$$\exists n_0 \in \mathbb{N} : \Pr(h(x, y_n) < \epsilon) > \delta \quad (\forall n > n_0, \forall \epsilon \in \mathbb{R}_0, \forall \delta : 0 \leq \delta < 1).$$

^aIf it halts after $m < n$ steps, $y_n = y_m$.

A particular subclass of asymptotically correct stochastic procedures are *Las Vegas Algorithms* (LVAs). An LVA is a randomized algorithm which always returns the correct output and whose expected runtime is finite. Note that the “finite *expected* runtime” requirement allows indefinite computation (albeit with an infinitesimal likelihood), i.e. the runtime of an LVA may not be bounded by a function of its input size. Take for example the problem of finding the position of an element b in a sequence with n elements, guaranteed to contain b exactly once. An LVA could iteratively check whether b is at a position i selected uniformly at random from $[1, n]$. If so, it returns i and continues otherwise. The (independent) probability of finding b each iteration is $\frac{1}{n}$ and hence the expected number of iterations is n . The likelihood of not having found b after k iterations is $(\frac{n-1}{n})^k$ which is > 0 for any finite k .

Remark that also deterministic procedures exist which are asymptotically correct, yet not effective. Examples are commonly encountered in the context of numerical search, optimization and approximation problems. Take for instance the problem of finding a root of a computable, continuous, real-valued function f in a given range $[a, b]$ with $f(a) * f(b) < 0$. A well-known algorithm solving this problem is the bisection method (Algorithm 3), which has the property that after k iterations, the absolute distance of c to a root (h) is bounded by $\frac{b-a}{2^{k+1}}$, i.e. it is asymptotically correct (for this h).

Algorithm 3 Pseudocode for the bisection method

```

1: procedure FINDROOT( $f, a, b$ )
2:   repeat
3:      $c \leftarrow (a + b)/2$ 
4:     if  $f(a) * f(c) > 0$  then
5:        $a \leftarrow c$ 
6:     else
7:        $b \leftarrow c$ 
8:     end if
9:   until  $f(c) = 0$ 
10:  return  $c$ 
11: end procedure

```

2.6.3 Non-exact Algorithms

In contrast to (asymptotically) correct procedures, non-exact algorithms are guaranteed to halt, but the result they return is not guaranteed to be correct. We discriminate between different types of non-exact algorithms based on the guarantees they do provide. While the specific guarantees can take many forms and are often problem-specific in nature, they typically involve some bound on the likelihood of failure, the size of the error ($\sim h$), or a combination of both. In what follows, we give two examples.

Monte Carlo algorithms: A useful guarantee for non-exact stochastic procedures, a.k.a. Monte Carlo algorithms, in the context of solving computational problems, is

Definition 2.28: probably solving computational problems

We say that a PTM M probably solves a computational problem (X, R) , if and only if, $\forall x \in X$, M is more likely to return the correct answer than a wrong one. Formally, if and only if it satisfies $\Pr(M(x) \text{ is correct}) > 0.5, \forall x \in X$, where

$$\Pr(M(x) \text{ is correct}) = \begin{cases} \Pr(M(x) = \tau) & \text{If } R(x) = \emptyset \\ \sum_{y \in \Sigma^*} \Pr(M(x) = y) * [xRy] & \text{If } |R(x)| > 0 \end{cases}$$

For instance, if we would terminate the uniform random search LVA, we described in the previous section, after $n - 1$ iterations, returning a random position, the likelihood error would be $(\frac{n-1}{n})^n < 0.5$. BPP is the class of decision problems for which there exists a PTM probably solving it, in polynomial-time. While in practice an algorithm with an error-rate of $0.499 \dots$ might not be acceptable, we can solve the problem in polynomial time (w.r.t. the input size) with an arbitrarily large probability by repeatedly executing the algorithm on the same input and returning the answer which was returned most frequently. Therefore, BPP (as opposed to P) is also sometimes regarded as the class of problems which can be solved efficiently in practice. Whether $\text{BPP} = P$ is an open question. For a long time, the most prominent problem, known to be in BPP, but not yet in P, was primality testing. For example, the Fermat test is a non-exact randomized procedure for primality testing. Even though in meantime exact polynomial-time primality tests are known [Agrawal et al. 2004], variations of the Fermat test are to date still used due to their relative efficiency and simplicity. The following fragment from [Abelson et al. 1996, pp. 54] summarizes why “probably solving” is a sufficient guarantee: *“In testing primality of very large numbers chosen at random, the chance of stumbling upon a value that fools the Fermat test is less than the chance that cosmic radiation will cause the computer to make an error in carrying out a ‘correct’ algorithm. Considering an algorithm to be inadequate for the first reason, but not for the second, illustrates the difference between mathematics and engineering.”*

Approximation algorithms are a special type of non-exact optimization methods:

Definition 2.29: approximately solving optimization problems

An algorithm a α -approximately solves a given optimization problem (X, R) if $\alpha(x) * f(a(x)) \leq f^*$ holds $\forall x = (S', f) \in X$, where $f^* = f(s^*)$ with $s^* \in R(x)$.

In optimization, we are often mainly interested in finding a good way to do things. Whether it is (provably) “the optimal way” is often of lesser importance. Conceptually, approximation algorithms obtain solutions which are provably “near-optimal”. A well-known example is the Christofides algorithm for the TSP [Christofides 1976], which guarantees a constant approximation factor of $\alpha(x) = \frac{3}{2}$.

Definition 2.30: heuristic algorithms

Heuristic algorithms (short heuristics) are non-exact problem-solving procedures that do not provide any relevant, provable guarantees w.r.t. the quality of the solution they obtain (if any); and/or the resources they require for doing so.

Heuristics algorithms, despite lacking theoretical guarantees, often require orders of magnitude less resources to return solutions which are “good enough” *in practice*. In optimization, the use of such methods can also be motivated theoretically. Let PTAS be the class of problems for which a Polynomial Time Approximation Scheme exists, i.e. a procedure that solves it α -approximately in polynomial time for arbitrary $\alpha > 1$. Let APX be the class of problems that can be solved β -approximately in polynomial time, where β is some constant factor. Whether $\text{PTAS} = \text{APX}$ is an open problem. However, it is known that $\text{PTAS} = \text{APX}$ implies $\text{P} = \text{NP}$ and some problems (e.g. MAX-SAT, TSP) are known to be APX-hard, i.e. *difficult to approximate*, unless $\text{P} = \text{NP}$.

2.6.4 Contract and Anytime Algorithms

Note that asymptotic correctness, by itself, is not a particularly useful property, in practice, as we cannot execute an algorithm indefinitely. However, non-exact variants can be obtained by preliminarily terminating such procedures. More generally, many procedures possess a property known as *monotonicity*: When given more time, the quality of the results they return will tend to be higher. While the algorithm designer could hard-code this trade-off between quality and resource usage, a better practice would be to leave such design choices open, presenting monotonic procedures as algorithmic frameworks parametrized by these preferences, a.k.a. **contract algorithms**. One type of contract algorithms, which we will refer to as *fixed budget* methods, take a given computation budget as input and are guaranteed to terminate and return the best solution found within the given

budget. For instance, a fixed budget variant of the bisection method in Algorithm 3 would take a maximum number of iterations k as input, after which c is returned. Another kind are *fixed confidence* methods, which take a bound on the probability and/or size of error as input, and terminate as soon as they can provide this guarantee. A fixed confidence variant of the bisection method would take a maximum distance to a root ϵ as input and return c as soon as $\frac{b-a}{2} < \epsilon$. One downside of contract algorithms is that the user has to specify the contract prior to execution. It may be difficult to predict beforehand the resources available/required to obtain a solution of a certain quality. A particularly useful class of methods, in this context, are so-called **anytime algorithms** [Dean and Boddy 1988]. Such procedures can be queried for a solution at any point of their execution. The bisection method is naturally anytime, with c being its *anytime solution*.

2.7 Search Methods for Combinatorial Optimization

In this section, we will describe and discuss procedures for solving combinatorial optimization problems. Here, our focus will be on *domain-independent* procedures, i.e. which can be used to solve any combinatorial optimization problem (e.g. all those described in Section 2.2.1), as opposed to any single one of them.

More specifically, we will consider *anytime search algorithms*. These methods can be viewed as iteratively evaluating the objective function value of candidate solutions in S' , each time retaining the best found thus far (\sim anytime solution). All methods described below essentially differ in their choice of candidate solution to evaluate next.

In the context of solving combinatorial optimization problems, one typically classifies search procedures as follows:

Exact methods are effective procedures that solve the problem exactly according to Definition 2.6, i.e. they are guaranteed to return an optimal solution, after a finite number of iterations.

Approximation methods are effective procedures that solve the problem approximately (see Definition 2.29), i.e. they are guaranteed to return a solution, whose objective function value is provably close to optimal, after a finite number of iterations.

Heuristic methods are procedures which are neither exact nor approximate, i.e. they provide no guarantees about the quality of the solution they return, if any.

Our focus, in this section and dissertation in general, will be on the latter. However, for the sake of completeness, we first discuss exact and approximation methods, collectively also known as systematic search methods.

2.7.1 Systematic Search

In this subsection, we will discuss how to solve any given combinatorial optimization problem. Remark that, by nature, these problems are all rather different and devising a single algorithm which can deal with such heterogeneity is troublesome to say the least. Therefore, the procedures we will describe here do not solve these problems in their natural form, rather they solve a problem which most combinatorial optimization problems can be reduced to, i.e. reformulated as (see also Section 2.4). Equivalently, these procedures may be viewed as algorithmic frameworks (see Section 2.3.3) which leave domain-specific design choices open and must be configured appropriately for a particular problem domain. Note that there may be many possible reductions/configurations and while we focus on different algorithmic frameworks here, the specific instantiation thereof is often at least as important in practice.

2.7.1.1 The Graph Search Reduction

Combinatorial optimization problems are commonly solved by reducing them to problems of the following form:

Definition 2.31: (lazy) graph optimization problem

Let

V be a finite set of vertices

$e : V \rightarrow \mathbb{R}$ be the evaluation function.

$V_0 \subseteq V$ be the non-empty set of initial vertices.

$E : V \rightarrow 2^V$ be the successor function.

The (lazy) graph optimization problem is defined as follows:

Given $\langle e, V_0, E \rangle$, find a vertex v^* such that $e(v) \leq e(v^*), \forall v \in V'$, where V' is the subset of vertices reachable from V_0 following the edges in E , i.e.

$$V' = \{v \in V \mid \exists p \in V^* : (p_1 \in V_0) \wedge (p_{|p|} = v) \wedge (p_{i+1} \in E(p_i), \forall 1 \leq i < |p|)\}.$$

Remark that every instance of the combinatorial optimization problem $\langle f, S \rangle$ can be reformulated as such by choosing $e = f$, $V_0 = S$ and $E(v) = \emptyset, \forall v \in V$. However, this is not the only possible reduction and typically better alternatives exist.

Remark that, in this naive reduction, all candidate solutions (S) are passed in an explicit form (e.g. as a collection of elements) to the solver. However, for the problems of interest in this dissertation, i.e. hard combinatorial optimization problems, the search space S is, *by nature*, specified in a relatively concise abstract mathematical form (e.g. see Section 2.2.1). Furthermore, for non-trivial problem instances, S is extremely large and generating such a collection, a priori, would require a lot of time and an intractable amount of space. For instance, the search space of the permutation flow shop problem consists of all possible permutations of n jobs. Assuming we choose $V_0 = S$, the set being passed to the search algorithm would contain $n!$ elements. For 15 jobs, any explicit representation would already require several terabytes. To overcome this issue, one uses *lazy* data-structures (e.g. lazy graph), which consist of a subset of the data (e.g. V_0) as well as a procedure (e.g. E) which can be used to recover/generate the remaining data (e.g. $V' \setminus V_0$). Remark that, in order to avoid losing optimality, V_0 and E must be chosen such that at least one vertex corresponding to an optimal candidate solution is reachable. Note that it is not always trivial to tell whether this is the case or not (e.g. 15-puzzle [Johnson et al. 1879]).

Beyond reachability, the choice of E is often also crucial for the efficiency with which the resulting problem can be solved. Here, E is commonly chosen such that every successor $v' \in E(v)$ corresponds to a minor variation of v . This way, E can typically be computed efficiently and sometimes it even allows us to also compute $e(v')$ more efficiently as a function of the $e(v)$, a practice known as delta-evaluation. E also provides an elegant way of excluding “candidates” that cannot possibly be optimal, or that do not satisfy some hard constraints. On the other hand, not every vertex has to correspond to a candidate solution or multiple vertices may correspond to the same. These redundant vertices are used to further structure the search space and enable more efficient generation/evaluation. In combinatorial optimization, candidate solutions can often be thought of as being composed of a set of components. A particularly prominent representation of the search space in this context is as a *construction tree*: Here, the root v_0 ($V_0 = \{v_0\}$) corresponds to the empty solution, leaves to the candidate solutions and each path from root to leaf can be viewed as a procedure constructing a candidate solution by iteratively adding components. As such, interior vertices correspond to “candidate solutions” having one or more components left unspecified, and are therefore also commonly referred to as *partial candidate solutions*. For example, for the permutation flow shop problem, the root could be $[\]$, and each vertex v could have $n - |v|$ successors, each appending an unscheduled job.

Finally, often $e \neq f$. For instance, because f is difficult to compute exactly. Also, f may not provide sufficient information about the quality of suboptimal solutions. For instance, a naive way to formulate the search problem variant of SAT would be to use $e = 1$ (when satisfied), $e = 0$ (otherwise). However, the standard reduction of the problem, i.e. MAX-SAT, uses a more informative e instead, equal to the number of satisfied clauses.

2.7.1.2 Systematic Graph Search Procedures

In this section, we will give a brief overview of general-purpose exact (and approximation) algorithms for the graph optimization problem (see Definition 2.31). This overview is in no way an extensive survey of exact search algorithms, as we merely aimed to provide an introduction to search and a context for the heuristic methods which we will discuss in Section 2.7.2. For the state-of-the-art in exact hard combinatorial optimization, we refer the interested reader to one of the many surveys on the topic (e.g. [Woeginger 2003]). We follow a similar approach as in [Russell et al. 1995, part II]. The main difference is that [Russell et al. 1995] introduces search algorithms in the context of an agent searching for a sequence of actions to achieve a goal, while we will introduce variants of these search algorithms used in the context of solving combinatorial optimization problems.

The most elementary search methods for solving the graph optimization problem can be viewed as instances of the tree optimization framework (TreeOpt) listed in Algorithm 4, which takes a single procedural parameter *select* as an argument. Central to the TreeOpt framework (and its many variants) is a collection, called the frontier, holding the set of vertices which have been generated, but not yet evaluated or expanded. Initially, the frontier contains V_0 . Each iteration, *select* determines which vertex v is selected to be removed from the frontier, evaluated, expanded, and have its successors (if any) added to the frontier. This method continues until the frontier becomes empty.

Algorithm 4 Tree optimization framework (TreeOpt)

```

1: procedure FINDBEST( $\langle V_0, E, e \rangle, \text{select}$ )
2:   frontier  $\leftarrow V_0$ 
3:    $e_{\text{best}} \leftarrow -\infty$ 
4:    $v_{\text{best}} \leftarrow \perp$   $\triangleright v_{\text{best}}$  is the anytime solution
5:   while frontier  $\neq \emptyset$  do
6:      $v \leftarrow \text{select}(\text{frontier})$ 
7:     frontier  $\leftarrow \text{frontier} \setminus \{v\}$ 
8:      $e_v \leftarrow e(v)$ 
9:     if  $e_v > e_{\text{best}}$  then
10:       $e_{\text{best}} \leftarrow e_v$ 
11:       $v_{\text{best}} \leftarrow v$ 
12:     end if
13:     frontier  $\leftarrow \text{frontier} \cup E(v)$ 
14:   end while
15:   return  $v_{\text{best}}$ 
16: end procedure

```

The following holds, independent of our choice of *select*:

1. When (V', E) is a tree with root v_0 and $V_0 = \{v_0\}$, TreeOpt will terminate and return the optimal solution after exactly $|V'|$ iterations.
2. When (V', E) is a Directed Acyclic Graph (DAG), TreeOpt will solve the problem, but may need $\mathcal{O}(|V'|^2)$ iterations, i.e. it may evaluate vertices multiple times.
3. When (V', E) contains cycles, the frontier will never become empty and TreeOpt never terminates.

While (3) implies that Algorithm 4 does not solve the general graph optimization problem, it can be extended in various ways to handle cyclic graphs. These extensions are known as the graph search/optimization framework. Here, the general trick is to “remember” (a subset of) the vertices we have evaluated thus far in order to avoid evaluating/expanding them multiple times. Remark that this trick also improves time complexity in (2). However, it comes at the cost of increasing the space complexity.

In the remainder of this subsection, we will restrict our discussion to (1). Recall that the search space in combinatorial optimization is commonly represented as a construction tree, making (1) a common case. In (1), the essential difference between different instantiations of the TreeOpt framework, i.e. choices of *select*, lies in the order in which vertices are evaluated. While time complexity does not depend on this order, space complexity and anytime performance do. Two canonical selection strategies, used in this framework, are:

Breadth First Selection (BFS), selecting the earliest generated vertex from the frontier.

Here, the frontier may be viewed as a standard First In, First Out (FIFO) queue.

BFS evaluates all vertices at depth (distance to root) i , before those at depth $i + 1$.

Depth First Selection (DFS), selecting the last generated vertex from the frontier. Here, the frontier may be viewed as a Last In, First Out (LIFO) queue, a.k.a. a stack.¹¹

In the context of combinatorial optimization, the latter is typically preferred. First, the space complexity of TreeOpt using BFS is $\mathcal{O}(b^d)$, where d is the depth of the tree and b is the branching factor (the maximum # successors of any vertex). While, using DFS, TreeOpt has a space complexity of only $\mathcal{O}(d \cdot b)$. Second, from an anytime perspective, whether BFS or DFS performs best will depend on whether high-quality candidates are to be found close or further away from the root. In a construction tree, all candidate solutions are in the leaves of the tree, i.e. far away from the root.

Another well-known strategy, targeted at improving anytime performance, is

¹¹This strategy is actually known as pseudo-DFS. *True* DFS (a.k.a. the backtracking algorithm) will evaluate/expand a node as soon as it is generated (i.e. before generating any other successors). Note that the space complexity of *True* DFS is $\mathcal{O}(d)$, with d the depth of the tree.

Best first selection(p), selecting a vertex $v \in \arg \max_{v' \in \text{frontier}} p(v')$ from the frontier. Here, the frontier may be viewed as a priority queue.

Note that best first selection is itself parametrized by the priority function p . Remark that if (V', E) is a tree, each vertex v can be viewed as the root of a subtree with vertices $V'_v \subseteq V'$, and the set of sets $\{V'_v \mid v \in \text{frontier}\}$ as a partitioning of the vertices still be evaluated. In this perspective, $p(v)$ can be viewed as predicting the desirability of exploring the subtree rooted at v .

Finally, remark that all search strategies described above halt after exactly $|V'|$ iterations. As $|V'|$ is often huge, running TreeOpt till termination may be intractable. We will conclude this section by discussing so-called Branch & Bound (B & B) methods [Lawler and Wood 1966], extending TreeOpt such that it may halt earlier, yet guarantee optimality. The idea is the following: Given we are able to compute an upper bound $h(v) \geq e(v')$, $\forall v' \in V'_v$, if a certain vertex v has $h(v) \leq e_{\text{best}}$, expanding it cannot possibly lead to a new v_{best} and we can cut this branch without loss of optimality. Therefore, we could extend TreeOpt to take such h as an argument, apply it to each evaluated vertex v , and expand v only if $h(v) > e_{\text{best}}$.¹² Note that this B & B framework can be extended to a fixed confidence framework (parametrized by α , using $h(v) > \alpha * e_{\text{best}}$ as criterion) guaranteed to solve the graph optimization problem α -approximately. A well-known instantiation of this framework is the A* algorithm [Hart et al. 1968], which combines B & B with a best first strategy using priority function $p = h$.¹³ It is worth noting that in adding h as a parameter, we do not lose any generality, nor do we make the framework more difficult to use, as we could pick $h(v) = +\infty$ by default. Clearly, the performance gain, if any, crucially depends on our choice of h : While tighter bounds may allow us to prune a larger part of the search space, they typically come at a cost of being more expensive to compute (and design).

2.7.2 Heuristic Search

In this section, we will describe heuristic methods used for solving (combinatorial) optimization problems, a practice which we will refer to as *heuristic optimization*. Unlike the systematic methods we have discussed thus far, heuristic search methods do not keep track of all yet unexplored options to guarantee that the solution returned is (approximately)

¹²Note that h is also commonly called a “heuristic”. This is consistent with our “definition” of a heuristic in Section 2.6.3, in that h can be viewed as a heuristically solving the function problem $(V, (v, \arg \max_{v' \in V_v} (e(v')))) \mid v \in V)$, i.e. $h(v)$ guesses the value of the best solution in V_v . When $h(v)$ is always an overestimate, h is called an “admissible heuristic”, and the B & B method using it exact. More generally, heuristic subroutines are often used to improve the efficiency of exact methods.

¹³Technically, A* does not cut any branches during the search, but rather implicitly cuts all branches at the end of the search by terminating when $h(v) \leq \alpha * e_{\text{best}}$, i.e. before the frontier becomes empty.

optimal. Clearly, in practice, we are interested in procedures that do obtain reasonable quality solutions in an acceptable time, i.e. perform well, despite their lack of theoretical guarantees. Complexity theory continuously tries to bridge the gap between procedures that perform well in practice and those which provably do so. As such, rather than viewing heuristics as a necessary evil, which is the popular perspective, one may alternatively view them as artifacts of the limits of modern complexity theory.

This lack of analytical knowledge about the performance of a heuristic procedure makes it intrinsically difficult to predict, *a priori*, whether a given heuristic will perform well, better/worse than another, etc. This, in turn, has a profound impact on how these procedures are designed: Lacking analytical knowledge, the designer must resort to empirical knowledge, i.e. try out various alternatives to find one that works well. To decide which alternatives to try (first), the designer relies heavily on intuition and experience.

In the last few decades, heuristic search procedures, of all sorts and flavors, have been successfully applied to countless hard combinatorial optimization problems. While the actual heuristic optimization methods are necessarily problem-specific, they can be viewed to follow a high-level search procedure which is more widely applicable. To avoid fragmentation, it has become standard practice to conceptually separate this reusable high-level search template, commonly referred to as a metaheuristic, from its problem-specific instantiation. While the meaning of the term “metaheuristic” has changed over time [Sorensen et al. 2017] and is to date ambiguous, of late, the following definition is gaining traction:

Definition 2.32: metaheuristic (framework) [Sørensen and Glover 2013]

A metaheuristic is a high-level problem-independent algorithmic framework that provides a set of guidelines or strategies to develop heuristic optimization algorithms.

Note that the term “metaheuristic” is also commonly used to refer to problem-specific instantiations of the framework. In this dissertation, we will use the terms “metaheuristic *framework*” and “metaheuristic *method*” to disambiguate the two uses.

In the remainder of this section, we will provide some examples of metaheuristic frameworks. This overview is by no means exhaustive and as a single method can be viewed as an instantiation of many different frameworks, our particular choice of metaheuristics is necessarily somewhat arbitrary. Furthermore, the sets of guidelines or strategies two metaheuristics represent may not be mutually exclusive, i.e. they can be combined, giving rise to “hybrids”. For an up-to-date, chronological overview we refer the reader to [Sorensen et al. 2017], for a more in depth treatment see [Hoos and Stützle 2004] and [Luke 2009].

2.7.2.1 Constructive Heuristics

This class of metaheuristics is naturally viewed as performing a non-systematic search in a construction tree. More specifically, as anytime performance is crucial and candidates in the construction tree are to be found in the leaves, these procedures will typically do so in a depth-first manner. No frontier is kept and as such they do not guarantee that an (approximately) optimal solution is found after any finite time. In addition, they may optimistically cut branches and therefore not even be approximately correct.

Simple construction heuristic: The simplest framework follows a single path in the tree and subsequently returns the candidate solution corresponding to the leaf. At each step, this heuristic can be viewed as cutting all but a single branch, determined by a possibly stochastic selection rule (*select*) passed as an argument to the framework. An example of an instantiation of this framework is the nearest neighbor heuristic for the TSP. Here, we select an arbitrary city as a starting point. Subsequently, we iteratively visit the unvisited city closest to our current location next. While efficient, the quality of the solutions obtained by such simple construction heuristics are typically not competitive with the state-of-the-art. However, the solutions generated by these heuristics are often used as starting points in more complex metaheuristics.

Algorithm 5 Repeated construction framework

```

1: procedure FINDBEST( $\langle v_0, E, e \rangle$ , select, terminate)
2:    $e_{\text{best}} \leftarrow e(v_0)$ 
3:    $v_{\text{best}} \leftarrow v_0$ 
4:   repeat
5:     children  $\leftarrow E(v_0)$ 
6:     while children  $\neq \emptyset$  do ▷ Constructs a solution
7:        $v_{\text{inc}} \leftarrow \text{select}(\text{children})$ 
8:       children  $\leftarrow E(v_{\text{inc}})$ 
9:     end while
10:     $e_{\text{inc}} \leftarrow e(v_{\text{inc}})$ 
11:    if  $e_{\text{inc}} > e_{\text{best}}$  then
12:       $e_{\text{best}} \leftarrow e_{\text{inc}}$ 
13:       $v_{\text{best}} \leftarrow v_{\text{inc}}$ 
14:    end if
15:  until terminate is satisfied
16:  return  $v_{\text{best}}$ 
17: end procedure

```

Repeated construction: Simple construction heuristics have a fixed runtime. When given less time, they do not return any solution, when given more, the solution will not be better, i.e. they are not anytime, nor asymptotically correct. A general strategy to overcome such weakness, assuming the subordinate heuristic is stochastic, is simply executing it repeatedly until some termination criterion `terminate` is satisfied. Hybridizing this strategy with simple construction heuristics gives rise to the repeated construction metaheuristic listed in Algorithm 5, where lines 5-9 correspond to a simple construction heuristic. Remark that repeated (construction) procedures can be viewed as iteratively sampling the search space according to a certain distribution. This distribution may be stationary, i.e. remain the same for each iteration (e.g. GRASP [Feo and Resende 1995]) or it may change over time as to increase the likelihood of generating high-quality candidates (e.g. cross-entropy [De Boer et al. 2005], ant colony optimization [Dorigo et al. 2006]).

2.7.2.2 Local Search Heuristics

The second class of metaheuristics we will discuss can be viewed as instantiations of the framework listed in Algorithm 6. They start from a given vertex v_0 . Each iteration, a given, potentially stochastic, step function (a.k.a a perturbation, mutation or successor function) is applied to the incumbent vertex v_{inc} . This process is repeated until some termination criterion `terminate` is satisfied. Local search heuristics can be seen as iteratively sampling candidates from a distribution conditioned on the last generated candidate, i.e. performing a form of correlated sampling.

Algorithm 6 Local search framework

```

1: procedure FINDBEST( $\langle v_0, \text{step}, e \rangle, \text{terminate}$ )
2:    $e_{\text{best}}, e_{\text{inc}} \leftarrow e(v_0)$ 
3:    $v_{\text{best}}, v_{\text{inc}} \leftarrow v_0$ 
4:   repeat
5:      $v_{\text{inc}} \leftarrow \text{step}(v_{\text{inc}})$ 
6:      $e_{\text{inc}} \leftarrow e(v_{\text{inc}})$ 
7:     if  $e_{\text{inc}} > e_{\text{best}}$  then
8:        $e_{\text{best}} \leftarrow e_{\text{inc}}$ 
9:        $v_{\text{best}} \leftarrow v_{\text{inc}}$ 
10:    end if
11:  until terminate is satisfied
12:  return  $v_{\text{best}}$ 
13: end procedure

```

Neighborhood search: Instantiations of Algorithm 6 can be viewed as performing a non-systematic search in a graph (V', E) , where V' is the set of vertices reachable from v_0 through E , with E a set of edges satisfying $\{(v, v') \mid \Pr(\text{step}(v) = v') > 0\} \subseteq E$. Neighborhood search explicitly decomposes step in terms of a successor relation E (a.k.a. neighbor relation) and a *pivot rule*. The former determines the modifications considered at each step, while the latter determines the likelihood any of these will actually be applied.

Some examples of neighborhoods are:

- **MAX-SAT:** The flip-one or one-exchange neighborhood consists of all models differing only in the truth assignment of a single variable.
- **TSP:** The 2-opt neighborhood consists of all tours formed by removing any pair of two non-adjacent edges $\{(c_i, c_j), (c_k, c_l)\}$ and replacing them by $\{(c_l, c_j), (c_k, c_i)\}$.

Some examples of common simple pivot rules are:

- **random:** a neighbor is selected uniformly at random, i.e. $\text{step}(v) \sim \mathcal{U}(E(v))$.
- **first-improvement:** an improving neighbor v' , i.e. $e(v') > e(v)$, is selected uniformly at random, or v if no such v' exists.
- **best-improvement:** a neighbor v' satisfying $e(v') \geq e(v''), \forall v'' \in E(v)$ is selected uniformly at random, or v if no such v' exists.

Hill climbing: We call a step function

- **greedy:** if $\forall v, v' \in V : \Pr(\text{step}(v) = v') > 0 \implies e(v') \geq e(v)$
- **strict greedy:** if $\forall v, v' \in V : \Pr(\text{step}(v) = v') > 0 \implies e(v') > e(v) \vee v = v'$.

Examples of greedy/strict-greedy step functions are those using a best/first improvement pivot rules. Instantiations of local search using a greedy step function are known as Iterative Improvement (II) or hill climbing heuristics.

While the II scheme is intuitive and often quickly finds high-quality solutions, it is not guaranteed to find the optimal vertex in (V', E) , i.e. it is not asymptotically correct. Figure 2.2 illustrates the problem.

First, typically only a small fraction of neighbors are improving, i.e. $E_{\text{greedy}} \subset E$, and therefore only a limited set of vertices $V'_{\text{greedy}} \subset V'$ may be reachable from v_0 by an ascending path (through E_{greedy}). For instance, in Figure 2.2, the optimal vertex v_3 cannot be reached from v_0 by a hill climbing procedure.

Second, II gets stuck (v_{inc} remains invariant) in a

- local optimum of E : $v \in V : e(v') \leq e(v), \forall v' \in E(v)$ (using strict greedy step)
- strict local optimum of E : $v \in V : e(v') < e(v), \forall v' \in E(v)$ (using greedy step)

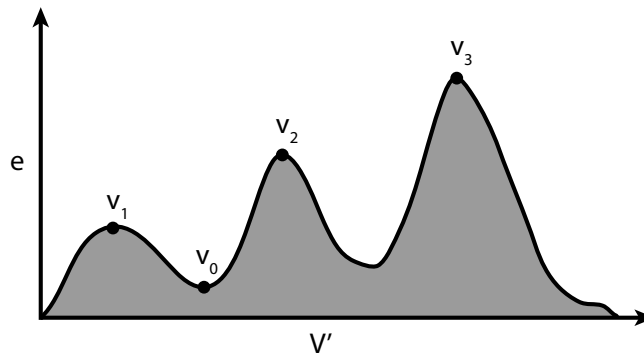


Figure 2.2: The fatal attraction of hill climbers to local optima.

For instance, in Figure 2.2, v_1 , v_2 and v_3 are all local optima. However, v_3 is the only global optimum. Note furthermore that the (quality of the) solution obtained depends on whether we by chance performed the first step to the left (v_1) or right (v_2).

Many more advanced metaheuristics may be viewed as extending (or hybridizing) II to overcome these limitations. Here, one typically distinguishes between strategies targeted at exploring distant regions in the search space and avoid getting stuck (called *diversification*) and mechanisms to quickly find high-quality solutions, such as II (called *intensification*). A careful balance between both is crucial for achieving satisfactory anytime performance.

Greedy Randomized Adaptive Search Procedures (GRASP): In analogy to repeated construction, we can repeat a randomized II procedure multiple times to guarantee that we eventually find the best reachable local optima (v_2 in Figure 2.2). However, typically not all vertices are guaranteed to be reachable from v_0 and this trick alone may not be sufficient to achieve asymptotic correctness (e.g. reach v_3 in Figure 2.2). This is typically addressed by varying v_0 each iteration; e.g. using v_0 's generated by a randomized simple construction heuristic. Methods hybridizing II and iterated construction in this fashion are known as Greedy Randomized Adaptive Search Procedures (GRASP) [Feo and Resende 1995]. GRASP can be viewed as iteratively drawing independent and identically distributed (i.i.d.) samples according to some distribution over local optima.

Simulated Annealing (SA): An alternative scheme to avoid getting stuck in a local optimum is to (with some likelihood) perform worsening steps, i.e. go downhill. Here, it is common practice to decompose the step function into the application of a non-greedy stochastic perturbation operator, followed by an acceptance criterion which determines

whether the “proposed” candidate is accepted as new incumbent or not. The former is the source of diversification, while the latter controls it. A prominent example is the Exponential Monte Carlo (EMC) criterion [Ayob and Kendall 2003], which accepts a worsening solution with probability $e^{\frac{e(v_{\text{prop}}) - e(v_{\text{inc}})}{T}}$, where T is a positive real-valued parameter called the temperature. Using this criterion, the greater the worsening proposed, the smaller the likelihood the proposal will be accepted. More specifically, the likelihood of accepting n times a worsening of a is equal to that of accepting a worsening of $n * a$ once. The EMC criterion was popularized in heuristic optimization in the form of Simulated Annealing (SA) [Kirkpatrick et al. 1983]. The criterion used in SA is a variant of EMC lowering the temperature over time. Its name and inspiration come from annealing in metallurgy, a technique involving heating and controlled cooling of a material to increase the size of its crystals and reduce their defects. SA starts at high temperatures and slowly lowers the temperature over the duration of the run. When cooling is sufficiently slow, the search process provably ends up in a global optimum. However, the practical relevance of this phenomenon is small. First, the required cooling rate is too slow for efficient search. Second, ending up in a global optimum is not required as optimization methods retain the best solution they come across. Nonetheless, the SA criterion remains popular and many more complex cooling (and reheating) schemes have been devised [Koulamas et al. 1994].

Iterated Local Search (ILS) combines the idea of repeated II used in e.g. GRASP, with the controlled diversification strategy used in e.g. SA. ILS [Lourencco et al. 2003] first applies II to locally optimize v_0 . Subsequently, it iteratively applies a non-greedy perturbation operator to escape the local optimum, followed by II, and finally uses an acceptance criterion to decide whether to use the newly generated local optimum as the starting point of next iteration. Similar to GRASP, ILS can be viewed as iteratively sampling the space of local optima. It differs in that it does so in a correlated, rather than i.i.d. fashion. Remark that ILS is an instance of the framework listed in Algorithm 6, where the step function is a composition of a non-greedy perturbation operator (diversification), followed by II (intensification) and an acceptance criterion (control diversification), i.e. ILS performs a local search in a graph of local optima.

Tabu Search (TS): Local optima are often strong attractors: many worsening steps might be required to avoid ending up in the same local optimum a few improving steps later. A powerful method to effectively escape local optima is Tabu Search (TS) [Glover 1989]. TS makes local optima or certain solution components of a local optimum “tabu” for some time (called *tabu tenure*). By not accepting/proposing candidate solutions with these attributes it effectively avoids ending up in recently visited states. TS is particularly successful in search spaces with large plateaus; e.g. MAX-SAT [Mazure et al. 1997].

Dynamic Local Search (DLS) uses an alternative to accepting worsening solutions, to escape a local optimum v . DLS changes the evaluation function such that v is no longer a locally optimal. This is typically realized by penalizing some of the attributes of the locally optimal solutions (similar to TS), lowering the quality of the local optimum, making further improvement possible. However, care must be taken as attributes common in local optima, are likely also part of a globally optimal solution. A well-known DLS framework is guided local search [Voudouris and Tsang 2003].

Very large-scale neighborhood search (VLSNS) procedures, rather than trying to escape local optima, solve a formulation of the problem where local optima are rare and/or of sufficiently high quality. VLSNS is based on the observation that larger neighbor relations E tend to have fewer and higher quality local optima. In the extreme, one could choose E such that every local optimum is also globally optimal and apply II to find one. However, such neighborhoods are typically exponential in size and finding the best or an improving neighbor itself a challenging optimization problem. VLSNS is a collective name for methods recursively applying advanced (heuristic) search procedures to do so efficiently nonetheless. A famous example is the Lin-Kernighan (LK) heuristic [Lin and Kernighan 1973], which is widely considered to be one of the best local search procedures for TSP.

Variable Neighborhood Search (VNS) [Mladenović and Hansen 1997] combines multiple smaller, rather than one large neighbor relation, to obtain similar results. Often when formulating a problem as a neighborhood search problem, multiple choices can be made for E . As opposed to picking any single one of them, VNS combines multiple, motivated by the observation that a vertex v may be locally optimal w.r.t. one, but often not w.r.t. (all) others. An example of a VNS framework is the Variable Neighborhood Descent (VND) method. VND orders a sequence of greedy step functions by the size of their respective neighborhoods and iteratively applies the smallest one for which v_{inc} is no local optimum. This process continues until v_{inc} is locally optimal w.r.t. all neighbor relations.

2.7.2.3 Population-based Heuristics

All heuristic frameworks discussed above are single point methods: they retain a single solution v_{inc} in memory after each iteration. Retaining multiple candidate solutions, is a general way to achieve diversification, i.e. not betting on a single horse. For instance, while a single point II scheme gets stuck when the incumbent solution is a local optimum, a population-based II would only get stuck when all members of the population are locally optimal. Also, populations allow one to compare the relative quality of, and similarities

between, members to perform a more directed form of diversification and focus computational efforts on the more promising. Note that “retaining a population” is just another idea, which can be combined/hybridized with the frameworks described earlier.

Local Beam Search (LBS) is a basic population-based framework. LBS starts off with a population of k candidate solutions. Subsequently, a *step* function is applied to each member of the population and from these $2k$ candidates the k best candidate solutions are retained. This process continues iteratively until some termination criterion is met. Note that the local search framework is a special case of LBS with $k = 1$.

Evolutionary Algorithms (EA) are a prominent class of frameworks loosely based on the process of biological evolution [Back 1996]. In the context of discrete optimization, the term Genetic Algorithm (GA) is also commonly used. Each iteration (“generation”), the population is updated using the following evolutionary operators:

- A **mutation** operator modifies a single member of the population (\sim *step* function). Typically, the effect of a mutation is small, changing only a few solution components (“genes”) in a stochastic and non-greedy fashion.
- A **crossover** (a.k.a. recombination) operator takes two members (“parents”) as input and generates a single individual (“offspring”), combining genes from both.
- A **selection** operator selects one or more members of the population for survival, reproduction or mutation. This is done such that fitter (\sim higher evaluation function value) individuals are more likely to be selected. Common examples are
 - *roulette wheel selection* selecting individuals proportional to their fitness value.
 - *tournament selection* selecting the fittest of k randomly selected individuals.

While these operators can be combined in many different ways, a canonical GA framework is listed in Algorithm 7. It starts with a population of k individuals. Each generation, offspring are produced through recombinations of l parents selected for reproduction. Subsequently, additional candidates are generated as mutations of m individuals selected from both the original population and the newly generated offspring. Finally, k individuals are selected for survival (from the $k+l+m$ candidates) and the process is repeated until some termination criterion is met. Note that LBS is a special case of the GA with $l = 0$ and $m = k$, i.e. only applying the mutation/selection operators (\sim asexual reproduction). GAs may suffer from a lack of intensification, and are therefore often combined with II, resulting in a hybrid known as Memetic Algorithms (MAs) [Moscato and Cotta 2002]. In MAs, the initial population, as well as the offspring in each generation, are locally optimized. As such an MA can be viewed as evolving a population of local optima.

Algorithm 7 Genetic algorithm framework

```

1: procedure FINDBEST( $V_0$ ,mutate,recombine,select, $l,m,e$ ,terminate)
2:    $V_{\text{inc}} \leftarrow V_0$  ▷ Initial population is passed as argument to the framework
3:   repeat
4:      $V_{\text{xo}} \leftarrow \emptyset$ 
5:     for  $t = 1 : l$  do ▷ Generate offspring by recombination
6:        $(v_i, v_j) \leftarrow \text{select two candidates from } V_{\text{inc}}$ 
7:        $V_{\text{xo}} \leftarrow V_{\text{xo}} \cup \{\text{recombine}(v_i, v_j)\}$ 
8:     end for
9:      $V_{\text{mut}} \leftarrow \emptyset$  ▷ Generate offspring by mutation
10:    for  $t = 1 : m$  do
11:       $v_i \leftarrow \text{select a candidate from } V_{\text{inc}} \cup V_{\text{xo}}$ 
12:       $V_{\text{mut}} \leftarrow V_{\text{mut}} \cup \{\text{mutate}(v_i)\}$ 
13:    end for
14:     $V_{\text{prop}} \leftarrow V_{\text{inc}} \cup V_{\text{xo}} \cup V_{\text{mut}}$ 
15:     $v_{\text{best}} \leftarrow \arg \max_{v \in V_{\text{prop}}} e(v)$ 
16:     $V_{\text{inc}} \leftarrow \text{select } k = |V_0| \text{ candidates from } V_{\text{prop}}$  ▷ Survival of the fittest
17:  until terminate is satisfied
18:  return  $v_{\text{best}}$ 
19: end procedure

```

Ant Colony Optimization (ACO) [Dorigo 1992] is a population-based repeated construction procedure inspired by the foraging behavior observed in ant colonies. Ants looking for food will explore the environment around their nest and when they find a source of food, they carry some to the nest and deposit pheromones on the way back. These pheromones are used by the ant and other ants to find their way back to the food source. As the paths taken differ slightly for each ant and shorter paths will get higher frequencies of pheromones (due to more frequent deposits and fixed evaporation), the ants will learn the shortest path from the nest to the food source [Goss et al. 1989]. ACO uses the same principle to construct good candidate solutions to NP-hard problems (e.g. TSP [Stützle and Dorigo 1999]). It operates as follows: Each edge in the construction tree is given a certain initial pheromone level (often the same). Subsequently, each of the k ants in the colony constructs a candidate solution performing a simple construction heuristic, where the likelihood of selecting any given successor depends on the amount of pheromones deposited on the associated edge. After constructing k candidate solutions, the pheromone levels are updated, reducing the original pheromone levels (evaporation) and depositing pheromones along the edges followed by the ants. Here, the amount of pheromones deposited is proportional to the quality of the constructed solution. This process is repeated until some termination condition is met. As for EAs, the construction process alone of-

ten leads to insufficient intensification and therefore II is often applied to the solutions constructed by the ants.

2.7.2.4 Inspired Frameworks

As discussed before, lacking analytical knowledge, the design of new heuristic frameworks often strongly relies on inspiration. Here, common sources of inspiration are “how humans would solve similar problems” and “natural processes with seemingly desirable characteristics”. Thus far, we have discussed the following prominent examples of the latter: Simulated Annealing (SA), Evolutionary Algorithms (EAs) and Ant Colony Optimization (ACO). While interesting and original, we argue that the aesthetically pleasing origin of these frameworks has led to the frequent, yet often poorly motivated use thereof. Similar concerns, for EAs in particular, have been expressed in [Russell et al. 1995, pp. 129]. Furthermore, the popularity (and success) of these frameworks has, in turn, prompted a proliferation of nature or otherwise “inspired” frameworks [Sörensen 2015]. Frequently cited¹⁴ examples draw inspiration from the behavior of various organisms, ranging from bacteria [Passino 2002], weeds [Roy et al. 2011], insects (e.g. bees [Karaboga 2005], fruit flies [Pan 2012], fireflies [Lukasik and Zak 2009]), glow worms [Krishnanand and Ghose 2009], frogs [Eusuff and Lansey 2003], cuckoos [Yang and Deb 2009], bats [Yang 2010], monkeys [Mucherino and Seref 2007], wolves [Mirjalili et al. 2014] to whales [Mirjalili and Lewis 2016]. Next to the behavior of organisms of all sorts and sizes, also various other natural (e.g. flower pollination [Yang 2012], flow of rivers [Shah-Hosseini 2009], hydrological cycle [Eskandar et al. 2012], chemical reactions [Lam and Li 2010], electromagnetism [Birbil and Fang 2003], gravity [Rashedi et al. 2009]) and even man-made (e.g. music [Geem et al. 2001]) phenomena have inspired metaheuristics.

Recently, some of these “inspired” metaheuristics have received strong criticism for their lack of scientific rigor [Sörensen 2015].¹⁵ In summary, the main criticism is that these “inspired” frameworks are commonly described and justified solely in terms of the novel metaphors that inspired them, abstracting how “new” and/or “beneficial” the underlying procedures actually are. In addition, these frameworks have a tendency to be overly complex, as they introduce certain mechanisms to support the analogy with the metaphor that inspired them, rather than based on empirical evidence that these mechanisms actually

¹⁴According to Google scholar™, all papers referenced here, have been cited over 100 times, multiple of which received over 1000 citations. These as such only represent the proverbial tip of the iceberg.

¹⁵We refer here to [Sörensen 2015] as it presents the most extensive argument thus far. However, [Sörensen 2015] is not isolated, e.g. recently the Journal of Heuristics (JoH), one of the major journals in the field, has updated its policy statement, to echo this critique, explicitly stating that it “fully endorses the view presented in [Sörensen 2015]”.

contribute to performance.¹⁶ Their high complexity, combined with the use of metaphor-specific terminology to explain them, makes it unnecessarily difficult to understand and relate them to other frameworks, resulting in fragmentation and reinvention.¹⁷ For these reasons, we will not describe any of these frameworks in this dissertation. However, a description/discussion of some prominent examples can be found in [Yang 2014].

2.7.2.5 Hyper-heuristics

Despite the numerous successful applications of metaheuristics to a wide variety of combinatorial optimization problems, they are not readily applied to newly encountered problems. That is, the design of a metaheuristic procedure for some combinatorial optimization problem of interest remains challenging, is often done largely from scratch, in an ad hoc fashion, strongly relying on human intuition, experience and labor, making it a costly process.

We distinguish two ways¹⁸ in which the community has tried to address aforementioned challenges, i.e. reduce the cost associated with applying metaheuristics:

1. **Off-the-shelf metaheuristics:** The community has continued to search for new metaheuristic frameworks and new ways of combining existing ideas that work well across a wide range of problems, i.e. achieve acceptable performance, independent of their problem-specific configuration.
2. **Design automation:** The community has embraced the possibility of letting computers, rather than humans, design heuristic procedures (see Section 3.3.2).

Hyper-heuristics [Burke et al. 2003] is a term which has become associated with some of these efforts. This term was first used in [Cowling et al. 2000] to mean “heuristic to choose heuristics”, which was later generalized as follows:

Definition 2.33: hyper-heuristic (HH) [Burke et al. 2010b]

A hyper-heuristic is a procedure for selecting or generating heuristics to solve hard computational search problems.

While there exists a rough consensus on the definition of this term, what exactly it entails, and its relation to metaheuristics, in particular, is a subject of confusion. Hyper-heuristics typically distinguish themselves from metaheuristics in that they are said to operate on

¹⁶A matter which we discuss extensively in Chapter 7.

¹⁷A well-known example of this is harmony search [Geem et al. 2001], which was shown to be a special case of the established evolutionary strategies metaheuristic [Weyland 2010].

¹⁸In Chapter 5, we investigate the possibility of combining both approaches: the automated design of off-the-shelf metaheuristics (see Section 5.2.1).

a higher level of abstraction by searching in the space of heuristics as opposed to meta-heuristics which search directly in solution space. However, what exactly it means to “search in a space of heuristics” remains somewhat underdefined. We argue that the underlying cause of this confusion is that the term “hyper-heuristic” is used to describe two rather different concepts, as distinguished in [Burke et al. 2010b, Burke et al. 2013] and corresponding to instances of the two general responses described above respectively:

1. **Selection hyper-heuristics** are heuristic procedures that search the solution space by selecting and applying search operators (called *low-level heuristics*) from a given set of alternatives (called the *heuristic set*). A selection hyper-heuristic is in essence just another metaheuristic framework.
2. **Generation hyper-heuristics** are procedures that automate the design of heuristic search procedures. In essence any of the techniques we will discuss in the next chapter, applied to this design problem, could be called a generation hyper-heuristic. However, predominantly genetic programming techniques are used within the hyper-heuristics community (see Section 3.3.2, p. 83).

In summary, the former returns a solution for an instance, while the latter returns a heuristic search algorithm. Beyond the common goal of reducing manual effort, these at first sight have little in common. We will revisit this discussion in Section 3.5.2.1 (p. 97).

In the remainder of this section, we will focus on selection hyper-heuristics. As discussed before, metaheuristics require users to specify various problem-specific search operators. For instance, to use an EA, we must specify how to initialize, mutate and recombine members of the population. Typically, many alternatives exist and the best choice is often problem *instance* dependent. In addition, it may be beneficial to combine multiple, i.e. change the operator used dynamically over the course of the run. The idea underlying selection hyper-heuristics¹⁹ is to allow users to pass these as a set, rather than forcing them to choose one a priori. Note that this framework is closely related to VNS, differing only in that it combines multiple operators, rather than multiple neighbor relations.

On the reusability of HHs: Are selection hyper-heuristics indeed more *off-the-shelf*, i.e. easier to (re)use, than “ordinary” metaheuristics? At first sight, specifying a set of operators H may seem to require more effort than specifying a single operator h . However, as the heuristic set may be a singleton, i.e. $H = \{h\}$, this is not the case. Using selection hyper-heuristics, we do not have to specify multiple operators, rather we are allowed to specify multiple if we have difficulty choosing. Nonetheless, there are a few pitfalls here.

¹⁹And closely related fields such as reactive search [Battiti et al. 2008], adaptive operator selection [DaCosta et al. 2008], parameter control [Eiben et al. 1999], etc.

First, the performance of a hyper-heuristic may still be sensitive to the choice of H [Mısırlı et al. 2012], leaving the user with the difficult choice of whether to include a possible operator h in H or not. Second, it shifts the challenge of choosing “which operator to use when”, from the framework user to the framework (designer). State-of-the-art selection hyper-heuristic frameworks are often highly complex, making them difficult to understand, implement and maintain (see also Section 7.3.1).

2.8 Summary

In this chapter, we have introduced various core concepts of algorithmics in general and metaheuristics in particular. Below, we summarize the content covered in each section.

In Section 2.1, we explained what “problems” are in the context of computer science and introduced the sorting and prime factorization problems, which were used as running examples throughout this chapter.

In Section 2.2, we introduced a class of computational problems known as optimization problems. Here, we first defined the global function optimization problem and subsequently characterized variants thereof such as the combinatorial optimization problem. Here, the latter is of particular interest as the design of general solvers for this problem is the main application area we considered in the scope of this dissertation. In Section 2.2.1, we described six examples of specific combinatorial optimization problems (e.g. TSP), which we will be using in our experiments in Chapters 5, 6 and 7.

In Section 2.3, we introduced algorithms as effective procedures for solving problems. In Section 2.3.1.2, we continued to formalize what exactly constitutes an “effective procedure”, providing a historical context, introducing the notion of undecidability and the Turing machine formalism. Subsequently, in Section 2.3.2, we discussed the relevance of algorithms in the context of automation, introducing the Universal (Turing) Machine and related concepts such as computer programs, programming languages and pseudo-code. Finally, in Section 2.3.3, we introduced “configurable” algorithms and motivated their use.

In Section 2.4, we discussed problem formulation, i.e. the interface between problems and solvers. In Section 2.4.1, we introduced the concept of reducibility and argued that m -reductions formalize the intuitive notion of problem (re)formulation. In Section 2.4.2, we discussed the relationship between solving and formulating a problem and presented a formal argument that they are two sides of the same coin. In Section 2.4.3, we discussed the practical implications hereof and argued for the merits of formulating a problem in its most natural, declarative and reducible form, and the pitfalls of implicit reductions.

In Section 2.5, we provided an introduction to complexity theory. In Section 2.5.1, we discussed the efficiency of algorithms and introduced the Bachmann-Landau notation. In Section 2.5.2, we studied the difficulty of problems, defining canonical complexity classes such as P, NP, NP-complete and NP-hard. Here, we argued that many interesting problems are NP-hard, i.e. no effective procedure exists that solves them efficiently (unless $P = NP$).

In Section 2.6, we argued that procedures used to “solve” problems in practice often do not adhere to the traditional notion of algorithms as defined in Section 2.3. Here, we characterized the nature and motivated the use of such methods, from a practical and theoretical point of view, touching upon some more advanced topics in complexity theory. In Section 2.6.1, we considered randomized algorithms. In Section 2.6.2, we introduced the notion of asymptotic correctness. In Section 2.6.3, we discussed non-exact algorithms, discriminating between approximation and heuristic procedures. In Section 2.6.4, we introduced contract and anytime algorithms.

In Section 2.7, we described anytime search procedures for solving combinatorial optimization problems. In Section 2.7.1, we first examined how such problems are commonly formulated and subsequently described simple exact and approximation algorithms for solving a problem in this form. In Section 2.7.2, we consider heuristic procedures, which are of particular interest as many combinatorial optimization problems (e.g. all those described in Section 2.2.1) are hard to solve exactly (NP-hard), or even approximately (APX-hard). Here, we introduced the notion of a metaheuristic and presented numerous classical examples thereof. In Section 2.7.2.4, we discussed the polarizing topic of nature-inspired metaheuristics. Finally, in Section 2.7.2.5, we examined some obstacles in the use/design of metaheuristics and discussed hyper-heuristics as a response.

3 | Algorithm Design

Thus far, we have introduced computational problems and algorithms as systematic procedures to solve them. In the remainder of this dissertation, we will discuss how to design algorithms for a given computational problem. In the next two chapters, in particular, we aim to give our reader an overview of how algorithms are currently being designed, in order to put our own research in this area, and that of others, into a wider perspective. In this chapter, we focus on presenting a high-level discussion of a wide variety of different design approaches, while postponing a more in depth treatment to the next chapter. This chapter is outlined as follows: In Section 3.1, we introduce the Algorithm Design Problem (ADP) as the problem of finding the best algorithm to solve a given computational problem. In Section 3.2, we take a step back, discussing what attributes make one algorithm “better” than another and examining the tension between them. Subsequently, we turn towards how the ADP is being solved. In Section 3.3, we discriminate different solution approaches based on the role the human designer is envisioned to play in the process. Here, we briefly discuss the manual design process, as a prelude to design automation, where we further distinguish between fully and semi-automated design approaches. In Section 3.4, we distinguish between pure analytical and empirical approaches, based on whether the designer uses only a priori information (\sim what is known beforehand) or also a posteriori information (\sim experience obtained through experimentation). In Section 3.5, we consider “when” algorithm design takes place, distinguishing between offline (“design before use”) and online (“design during use”) approaches, discussing their respective strengths, weaknesses and the possibility of combining the best of both worlds in a semi-online approach. Finally, in Section 3.6 we provide a summary of the content covered in this chapter.

3.1 The Algorithm Design Problem

In Chapter 2, we have seen that algorithms formalize problem-solving procedures such that they can be executed automatically. Therefore, when faced with a problem we would like to solve automatically, we are inevitably faced with another, associated problem: we need an algorithm to do so.¹ In what follows, we briefly characterize this problem, which we will be referring to as the Algorithm Design Problem (ADP). We will use the term “target problem” to refer to the problem we want an algorithm for.

The ADP is *the problem underlying algorithm design*, i.e. it is the problem algorithm design (attempts to) solve. It is a concept of our own, which we introduce in function of presenting our own personal perspective on algorithm design in this chapter. A key feature of this perspective is that we treat algorithm design within the standard framework of algorithmics, i.e. we view algorithm design as solving a problem: the ADP.

Let us try to make this notion somewhat more concrete. Consider one would like to solve some problem (e.g. sorting). There are numerous ways (\sim algorithms) to do so, some of which are better (e.g. exact vs. non-exact, faster, stable vs. unstable, in-place vs. out-of-place, etc.) than others. In algorithm design, we are interested in finding the “best” way of solving a target problem (e.g. the best algorithm for sorting sequences). Conceptually, the ADP is *the problem of how to best solve problems*.

In the remainder of this section, we define the ADP as a computational problem.

Definition 3.1: Algorithm Design Problem (ADP)

Let A_U be the set of all algorithms.^a In the algorithm design problem, given a preorder^b \preceq over A_U , we are to find a design $a^* \in A_U : a^* \not\preceq a, \forall a \in A_U$.

^aNote that A_U is not restricted to the set of procedures solving some specific target problem, but rather is a universal set containing “any procedure that solves some problem”. Further formalization of this notion is hindered by the lack of a generally accepted, formal definition of what “an algorithm” is (see Section 2.3). Limiting oneself to a Turing machine model of computation (see Section 2.3.1.2), one could formalize A_U to be the set of all total TMs.

^bA binary relation having the reflexive and transitive property.

Remark that we essentially defined the ADP as a subclass of optimization problems with search space A_U . In the context of the ADP, we will also call A_U the *design space* and elements thereof *candidate designs*. Note that instances of the ADP solely differ in \preceq , i.e. the order they impose on the design space. While the target problem does not appear explicitly in this definition, it is typically one of the factors inducing \preceq , i.e. we prefer

¹[Rodriguez et al. 2007] called this problem the “associated problem” in the context of hyper-heuristics.

algorithms that perform well for some problem of interest (see Section 3.2). For instance, if our target problem is sorting, algorithms that actually sort sequences, e.g. selection sort (see Algorithm 2) will be preferred over algorithms that do not, e.g. trial division (see Algorithm 1), the bisection method (see Algorithm 3), etc. Formally, we have $\forall a, a' \in A_U : a \in A_{\text{isorta}} \wedge a' \notin A_{\text{isorta}} \implies a' \prec a$, where $A_{\text{isorta}} = \{a \in A_U \mid a(x) = R_{\text{isorta}}(x)\}$.

Finally, remark that the ADP, as formulated above, attempts to model “the problem we are actually trying to solve when designing algorithms”, independently of whether we, in practice, know what that problem is, i.e. the actual \preceq is typically not known to the designer, at least not in an explicit form. As such, this is not the form in which we will be solving it using computers (see Section 2.4). Rather, automated design approaches can be viewed as solving the actual ADP by reducing it to some other computational problem.

3.2 Algorithm Quality

As we see it, algorithm design is essentially about “finding the best algorithm to do something” (see Section 3.1). This raises the question: *What makes one algorithm better than another?* Clearly, this is to some extent inherently use case dependent; e.g. if we want to sort sequences, we prefer sorting algorithms. In Definition 3.1, we did not restrict these preferences in any way, allowing \preceq ’s to be arbitrary preorders. As a consequence, when looking at preferences across all instances of the ADP (\sim use cases), one could argue that “all algorithms are equally good” (\sim No Free Lunch, see also Section 3.2.3.3), since every algorithm will be the best/worst for as many ADP instances as any other algorithm.

However, we argue that for the instances of the ADP we encounter in practice, \preceq is not arbitrary at all, i.e. algorithms exist which are preferred in more practical use cases than others. Remark that this directly relates to our “reusability” objective (see Section 1.2), as these algorithms can be reused in more different settings than others. Next, we discuss some attributes that make one algorithm more reusable than another (answer Q.A.1).

3.2.1 Performance

An important reason to prefer one algorithm over another is because it “performs” better. In this section, we examine what exactly this entails. Here, we first briefly discuss performance on a single input, and subsequently discuss performance across multiple inputs.

Efficiency: For a given input x , each algorithm a (TM) determines a sequence of instructions e_x^a to be executed to obtain an output (see Definition 2.9). Two different algorithms a and b may have $e_x^a \neq e_x^b$ and the time it takes to execute these instructions may vary as well,

i.e. $t(e_x^a) \neq t(e_x^b)$. More generally, a randomized algorithm a (PTM, see Section 2.6.1) can have a set of possible executions E_x^a on a given input x (see Definition 2.25). Its runtime RT_x^a is a random variable with $\Pr(\text{RT}_x^a \leq t) = \sum_{e \in E_x^a} \Pr(e|a, x)[t(e) \leq t]$ indicating the likelihood that a solves x within a time budget T , a.k.a. its Run-Time Distribution (RTD, [Hoos and Stützle 1998]). All other things being equal, we prefer algorithms which require less resources. On x , we prefer an algorithm a over another b , if it probabilistically dominates the other: $\Pr(\text{RT}_x^a \leq t) \geq \Pr(\text{RT}_x^b \leq t), \forall t$. If neither algorithm dominates the other on x , preferences may be determined by some property of interest of the respective RTDs. Here, the minimal expected runtime is commonly used, i.e. $\mathbf{E}[\text{RT}_x^a] \leq \mathbf{E}[\text{RT}_x^b]$.

Efficacy: While efficiency relates to the amount of resources an algorithm requires to produce outputs, efficacy relates to the desirability of these outputs. Different executions may lead to different outputs, not all of which may be equally desirable. For example, non-exact procedures (see Section 2.6.3) return solutions which may be incorrect. In addition, not all incorrect answers may be equally bad, e.g. in optimization settings (see Section 2.2) the quality of suboptimal solutions (e.g. # unsatisfied clauses in MAX-SAT) may differ. Specifically, for a randomized, non-exact, optimization procedure a , the quality of the solution obtained SQ_x^a (on some input x) is a random variable with $\Pr(\text{SQ}_x^a \geq q) = \sum_{e \in E_x^a} \Pr(e|a, x)[f(y(e)) \geq q]$, a.k.a. the Solution Quality Distribution (SQD), where f is the objective function to be maximized.

Per-input performance: All other things being equal, we prefer algorithms which obtain higher quality solutions, faster: We prefer an algorithm a over another b on input x if

$$\forall q : \Pr(\text{SQ}_x^a \geq q) \geq \Pr(\text{SQ}_x^b \geq q) \quad \wedge \quad \forall t : \Pr(\text{RT}_x^a \leq t) \geq \Pr(\text{RT}_x^b \leq t).$$

Clearly, this may not hold for every pair of algorithms. For instance, a heuristic optimization procedure will quickly return a “good” solution, while exact optimizers typically take longer but are guaranteed to return an optimal solution (see Section 2.7). In general, we will use $p : X \times A \rightarrow \mathbb{R}$ to denote some quantification of per-input performance such that $p(x, b) \leq p(x, a)$ if and only if algorithm a performs better than b on x . Remark that p captures, amongst others, our relative preferences between efficiency and efficacy.

Generality: The generality of an algorithm is its ability to perform well across a wide range of problem instances. Thus far, we have discussed performance on a single input. While, an algorithm which performs well on (only) a single input may definitely be of interest in certain settings, e.g. to automatically find a prime larger than $2^{74207281} - 1$,²

²At the moment of writing, the largest known prime number.

or to automatically generate a proof for $P \neq NP$, it loses its value as soon as its execution has completed, i.e. the output is obtained. We will therefore also call such algorithm “disposable”.³ Generality allows an algorithm to be *reused*. As motivated in Section 1.2, our focus in this dissertation will be on “reusable” algorithms.

How do we compare algorithms across multiple problem instances X , i.e. for a given computational problem (\sim target problem)? All other things being equal, we prefer an algorithm a over another b if it performs better on all inputs, i.e. $p(x, b) \leq p(x, a), \forall x \in X$. Here again, an algorithm a may not dominate another b across all inputs and for such pairs, our preferences may be determined in some other way. One measure, which we already discussed in Section 2.5.1 is worst-case performance, i.e. $\min_{x \in X} p(x, b) \leq \min_{x \in X} p(x, a)$. A downside of this measure is that while it tells you a (theoretical) limit on how bad performance may be, it may not be representative of an algorithm’s actual performance in practice. Also, estimating worst-case performance empirically is troublesome. An alternative for worst-case performance is average-case performance:

Definition 3.2: average-case performance

Let X be a set of possible inputs and $\mathcal{D} : X \rightarrow [0, 1]$ a distribution over X . The average-case performance of an algorithm a is given by $o(a) = \mathbf{E}_{x \sim \mathcal{D}} p(x, a)$, where $p(x, a)$ is some measure of performance of algorithm a on input x .

Note that while X specifies the set of possible problem instances, \mathcal{D} specifies the likelihood of actually receiving any of these as input, in practice. A computational problem augmented with a distribution over its inputs is also known as a *distributional problem* [Ben-David et al. 1989]. In this work, we will use average-case performance as our measure of generality.

3.2.2 Beyond Performance

Performance is important. However, in what follows, we would like to argue that it is not “the only thing that matters”. Imagine you are given a choice between (1) a black box solver and (2) an algorithm, having the same “problem-solving capabilities”, i.e. performance. Clearly, one would, in most settings, prefer the algorithm, i.e. one is not merely interested in solving a problem, but also in the underlying logic used to solve it. This is particularly true in the context of computer science, where one is concerned with understanding *why* something performs well/poorly. For instance, to assess the significance of an algorithmic contribution, not only performance matters: achieving such performance in a new way (through an untrodden path) has more potential, may lead to novel insights, inspire others, while doing so reusing prior art provides further evidence for the merits of an

³Adopting terminology introduced in [Bader-El-Den and Poli 2007] in the context of hyper-heuristics.

existing approach. Also, while our black box solver may perform well for some particular problem, it does not provide any insights which can be used to further improve it or to solve other problems, i.e. it does not allow (partial) reuse of ideas. Finally, it does not permit theoretical analysis (of correctness, worst-case performance, etc.).

In [Adriaensen and Nowé 2016a], we argued for the importance of *simplicity*. In presenting algorithmic contributions overly complex, the benefits of having the algorithm (rather than merely a black box solver) are largely lost, as the only way to deal with this complexity, is to treat such algorithms as black boxes in practice. In Chapter 7, we will discuss this matter in more depth and present a simple approach to eliminate unnecessary complexity.

3.2.3 Conflicting Criteria?

To summarize, we prefer simple algorithms which require little time (and memory) to obtain high-quality solutions for a wide range of problems. However, these objectives are commonly viewed as being strongly conflicting. Indeed, we find that algorithms are rarely both efficient, effective, general and simple. However, we wish to stress that this does not imply that such algorithms do not exist. Instead, using contemporary design techniques, it may simply be extremely difficult to find them (which would imply that we need better design techniques, or try harder...) In what follows, we have a closer look at the tension between our objectives. At the end of this section, we summarize our findings.

3.2.3.1 Time vs. Space Efficiency

Efficiency is related to the resources required to execute an algorithm. Thus far we have focused on one resource: time. However, executing an algorithm requires more than just time, e.g. it also requires space to store the data to be used in further calculations. In practice, memory capacity may be limited, or adding additional memory may come at a cost. Furthermore, it is often possible to trade space usage for computation time and vice versa, also known as *the space-time trade-off* [Hellman 1980]. A typical example is *memoization*, i.e. the practice of storing and reusing the output of calculations which would otherwise have to be performed multiple times during a run. Do note that modern computers tend to need more time to execute memory intensive computations (e.g. due to the memory hierarchy, [Alpern et al. 1994]) which somewhat counteracts this trade-off. Obviously, the fact that frequently the fastest known algorithm does not use the least memory, does obviously not preclude the existence of such algorithm. Let us consider this problem theoretically: Let P be the class of decision problems which can be solved

time efficiently (see Definition 2.18), let polyL^4 be the class of decision problems which can be solved memory efficiently and finally let SC be the class of problems for which a single algorithm exists which is both time and space-efficient [Cook 1979]. One may wonder whether there exists a problem for which a time-efficient algorithm and a space-efficient algorithm exists, but no single algorithm that is both time and space-efficient, i.e. $P \cap \text{polyL} \neq \text{SC}$? This is an open problem and it being false would *not* imply $P = \text{NP}$. Put differently, time and space efficiency may not have to be strongly conflicting.

3.2.3.2 Efficiency vs. Efficacy

As we already discussed in Section 2.6.4, there is often a trade-off between the quality of the results obtained by non-exact algorithms and their runtime. Unless we put lower bounds on quality, this trade-off is clearly inherent, as an algorithm which simply returns its input (TM with $q_0 \in F$) requires minimal resources (no time and only n space). The notions of non-exact problem-solving discussed in Section 2.6.3, give rise to the complexity classes BPP and APX, corresponding to the sets of problems which can be efficiently probably solved (see Definition 2.28) and approximated up to a constant factor (see Definition 2.29) respectively. Here again, $P \subset \text{BPP}$ and $P \subset \text{APX}$ are open problems. Note that unlike the space-time trade-off, a negative answer here would imply $P = \text{NP}$. On the other hand, given positive answers, optimal anytime solvers (see Section 2.6.4) could still exist that at any point in time return the best solution that can be obtained in that time.

3.2.3.3 Generality vs. Peak Performance

When it comes to human abilities, one often distinguishes between specialists who have mastered a limited set of skills, and generalists who have dabbled with a wide range of skills, without really excelling at any one of them, as is for instance captured in the figure of speech: “Jack of all trades, master of none?”. However, there seem to be exceptions to this rule, so-called polymaths, or universal (wo)men, who have mastered an exceptionally diverse skill set. The quintessential polymath is arguably Leonardo da Vinci.

How about algorithmics? As with humans, there is no lack of examples where algorithms designed for/applicable to specific problems (*made-to-measure*), outperform more general (*off-the-shelf*) alternatives. For instance, to sort sufficiently long sequences of integers within a limited range, counting sort algorithms (e.g. radix sort) will typically outperform the more general comparison-based methods (e.g. merge sort). Similarly, convex optimization problems are typically solved more efficiently using dedicated convex optimization

⁴ polyL is the class of decision problems solvable by a deterministic TM (see Definition 2.8) using at most polylogarithmic space, i.e. $\mathcal{O}(\log(n)^k)$ cells, where n is the input-size and k some finite constant.

methods (e.g. interior point methods [Nesterov and Nemirovskii 1994]) than using more general black box optimizers (e.g. CMA-ES, [Hansen and Ostermeier 1996]). More generally, when comparing a limited sample of different algorithms for a given problem, one often finds that no single algorithm is non-dominated on all problem instances.

What if we could compare all possible algorithms? Does good performance on some problem instances necessarily come at the cost of performing worse on other instances? Or does a universally optimal solver exist? It is easy to see that the latter cannot exist: Any *finite* subset of problem instances can be solved in $\mathcal{O}(n)$, where n is the (input+output)-size, by an algorithm encoding a table of correct (input, output) pairs.⁵ However, as an implication of the time hierarchy theorems [Žák 1983], not all solvable problems can be solved in $\mathcal{O}(n)$. For instance, the problem of deciding whether any given deterministic TM M on any given input x , halts after performing at most $2^{|x|}$ moves can be shown to require worst-case super-polynomial time ($\notin P$).

The No Free Lunch Theorem: What if we were to restrict our target problems to those with an infinite number of instances? What can be said about average-case performance? In particular, we will discuss the following (as it is particularly relevant to our scope):

“Does there exist a search procedure which, on average, performs better than any other for the general combinatorial optimization problem?”

The No Free Lunch (NFL) theorem for optimization [Wolpert and Macready 1997] is widely considered to answer this question negatively. The name of the theorem is derived from the popular expression “There ain’t no such thing as a free lunch”, used to indicate that nothing actually comes for free and that there are always hidden costs. In what follows, we first state the NFL theorem, followed by a critical discussion of its significance.

Theorem 3.1: The No Free Lunch theorem (for optimization)

In the context of this theorem, let

- “optimization algorithm” denote a deterministic search procedure, which evaluates all candidate solutions exactly once (*complete* and *non-repeating*).
- “optimization problem” denote any combinatorial *black box* optimization problem and without loss of generality, we represent an instance $\langle S, f \rangle$ as a sequence $f \in \mathbb{R}^{|S|}$ such that every f_i corresponds to an $f(s)$ for a unique $s \in S$.

⁵Remark that this also implies that, from a theoretical perspective, there is no such thing as unsolvable or intractable problem *instances*. For example, for any finite set of TSP instances (e.g. TSPLIB, [Reinelt 1991]), there exists an algorithm which will produce the optimal tour in $\mathcal{O}(n)$.

- $Y(f, m, a)$ denote the sequence of objective function values observed (\sim *solution quality trace*) during the first m evaluations performed by an optimization algorithm a on the optimization problem instance f .

The NFL theorem^a states that, for any pair of optimization algorithms a and b , for any optimization problem F , for any (per-input) performance measure p , for any $1 \leq m \leq |f|$ and for any $k \in \mathbb{R}$, the following holds:

$$H_{F,m}(k) = \sum_{f \in F} [k = p(Y(f, m, a))] = \sum_{f \in F} [k = p(Y(f, m, b))] \quad (\text{NFL property})$$

$$\Longleftrightarrow$$

F is closed under permutation (c.u.p.), i.e. $\forall f \in F : P^f \subseteq F$ (c.u.p. property)

^aThere are many variants of the NFL theorem. Here, we state the “sharpened” version of the theorem proven by [Schumacher et al. 2001], in the form used in [Igel and Toussaint 2005].

The NFL theorem states that $H_{F,m}$, i.e. the histogram of anytime performance measurements made after m evaluations on *all* instances of an optimization problem F , is independent of the optimization algorithm used, if and only if F is c.u.p. This theorem implies that if F is c.u.p. then according to any preference relation which is solely a function of $H_{F,m}$, every algorithm is equally desirable. In particular, it implies that (assuming F is c.u.p.) the average solution quality obtained by two optimization algorithms a and b after m iterations is equal when averaged uniformly across all instances in F .

Now, we will discuss the relevance of the NFL theorem w.r.t. solving the *general* combinatorial optimization problem. Here, there are two key observations to be made:

First, the NFL theorem *only* applies to combinatorial *black box* optimization problems, i.e. the only information passed to the solver for an instance is the search space S and some means of evaluating f for any $s \in S$. For naturally occurring combinatorial optimization problems, we often have additional information, e.g. we may know f to be a member of a given family of functions, which could be passed to/exploited by the solver. At most, the NFL theorem implies that all solvers treating f as a black box are equally good/bad.

Second, while the NFL-property holds for the general black box optimization problem (since $F = \mathbb{R}^*$ is c.u.p.), the latter does *not* directly imply that the average-case performance of any black box optimizer is equal. Note that general black box optimization is a *computational problem*, while average-case performance is defined for *distributional problems* (see Definition 3.2), i.e. beyond $H_{F,m}$, it also depends on the likelihood of actually being asked to maximize f in practice (i.e. \mathcal{D}). The NFL as formulated in [Wolpert and Macready 1997] only trivially implies equal average-case performance (\sim no free lunch),

under the assumption that \mathcal{D} is uniform ($\mathcal{D} = \mathcal{U}(F)$), i.e. each f is equally likely.⁶ Otherwise, algorithms can achieve superior performance by trading on performance for f 's which are less likely to be encountered in practice. If one assumes uniformity for the general black box optimization problem, i.e. $D = \mathcal{U}(\mathbb{R}^*)$, one may equally reasonably assume uniformity for decision problems. In Section 2.3.1.2, p. 32, we discussed that the latter implies that the likelihood of encountering a decidable problem would be infinitesimal.

In summary, the relevance of the NFL theorem w.r.t. solving the general combinatorial optimization problem is arguably limited. Even if we were to (artificially) restrict ourselves to the black box optimization setting, using the NFL to motivate trade-offs between generality and peak performance is only reasonable under strong a priori assumptions. If one were to generalize such assumptions to other problem classes (e.g. decision problems), one could equally (un)reasonably argue that it is not worth trying to solve these problems in the first place, as they are unlikely to be solvable anyway.

3.2.3.4 Simplicity vs. Performance

In this section, we distinguish and discuss the relation between two kinds of complexity:

Computational complexity: complexity in the sense of “efficiency” (see Section 2.5).

Logical complexity: complexity in the sense of “understandability”.

When discussing the topic of logical complexity with colleagues, we found that while most agreed with the premise that simplicity is desirable, many were also skeptical when it came down to actively pursuing simplicity. In general, there was a concern that in reducing logical complexity, we would lose something “more important”: performance. In what follows, instead of discussing the relative importance of performance and simplicity, we will examine the tension between both objectives.

Admittedly, there is no shortage of examples where simple algorithms perform worse than more complicated alternatives. Take for instance sorting: While sequential algorithms (e.g. selection sort, insertion sort, bubble sort, etc.) are simpler than divide & conquer sorting algorithms (e.g. merge sort, quicksort, etc.), the latter tend to perform better for all but very small input sequences. More complicated hybrids such as Tim sort,⁷ tend to perform even better on average. However, as discussed before, this alone does not preclude the existence of simple and highly performant algorithms which have yet to be discovered.

⁶[Igel and Toussaint 2005] examined possible relaxations of this uniformity assumption and concluded that distributions satisfying the necessary conditions are rare for reasonably large problems.

⁷Tim sort combines merge sort and insertion sort. It is named after Tim Peters who implemented it in 2002 for use in the Python programming language. It is currently the default algorithm for sorting lists in Python (`list.sort`, as of version 2.3) and Java (`Collections.sort` as of SE 7).

What can be said theoretically about “the computational power of simple algorithms”, i.e. what problems can be solved (efficiently) by simple algorithms? Before we can answer this question, we must first decide what it means for an algorithm to be “simple”. Here, we will be using the number of control states of a TM ($|Q|$) as a measure of logical complexity, even though the argument below is equally valid for alternative logical complexity measures such as description length. We call a TM C -simple if it has less than C control states, where C is some finite constant. A key observation here is that the number of C -simple TMs is finite, for any C . Combining this with the fact that every TM solves at most one decision problem, only a finite number of problems are solvable by C -simple TMs, for any C . As P is infinite, there are decision problems which can be solved in polynomial time (\sim efficiently), but not by any C -simple TM. In summary, there exist problems for which algorithms with an arbitrarily high logical complexity are needed to solve them (efficiently).

At first sight, this sketches a rather grim picture. However, before jumping to conclusions, we wish to stress that the result above only implies that there is such a thing as “essential” complexity. It does not imply that all complexity is essential. On the contrary, we could complicate any algorithm without affecting its function.⁸ Also, we would like to point out that the above result relies on a counting argument similar to the one we used in Section 2.3.1.2 (p. 32) to show that uncountably many problems cannot be solved by a TM. While the latter is well-known, it has not led people to question the practical relevance of the TMs. Therefore, we see no reason to doubt the practical relevance of C -simple TMs either. The crux is that it is unclear whether problems requiring high logical complexity are actually encountered *in practice*. For many seemingly complicated problems, relatively simple, highly performant algorithms have been found; and it is our opinion that we should continue to search for them, while being wary of possible theoretical limitations, i.e.

“Everything should be made as simple as possible, but not simpler” - Albert Einstein

3.2.3.5 Summary

While it is trivial to find examples of trade-offs between our objectives (time/space efficiency, efficacy, generality and simplicity) in known algorithms, whether such observations can be generalized is largely an open problem. Put differently, we did not find conclusive evidence that precludes the existence of algorithms which are non-dominated for all aforementioned objectives, for any problem of practical interest.

As such, we would like to stress that while it is commonly accepted that one should be wary of claims of having found (or being able to find) a truly optimal algorithm, one

⁸In Chapter 7, we will discuss a specific technique for identifying/eliminating “non-essential” complexity.

should be similarly cautious of motivating trade-offs based on a *conjectured* theoretical impossibility (e.g. NFL theorem), i.e.

“Keep your eyes on the stars, and your feet on the ground.” - Theodore Roosevelt

3.3 Who is the Designer?

Thus far, in this chapter, we have introduced the Algorithm Design Problem (ADP) as the problem of finding the best algorithm to solve a given problem (see Section 3.1). Subsequently, in Section 3.2, we discussed attributes that make one algorithm better than another, and whether such thing as “the best” algorithm exists. We now turn towards describing how algorithms are designed, i.e. how the ADP is currently being solved.

As discussed in Section 3.1, instances of the ADP are faced whenever we encounter problems we would like to solve automatically, using computers. As such, ADPs are ubiquitous in computer science, and it should come as no surprise that they are being solved in many different ways. In the remainder of this chapter, we will attempt to present an overview of this wide array of different design approaches. Here, we will each time discuss their relative strengths and weaknesses, and emphasize in subsequent sections those we deem to be the most promising in the context of the design of reusable heuristics (answer Q.A.2). In this section, we will start by roughly discriminating different approaches based on the role of the human designer in the design process.

3.3.1 The Manual Approach

To date, algorithms for many real-world problems are predominantly designed *manually*. In what follows, we will make a brief attempt to describe “how” humans design algorithms.

Note that we will not cover best practices here, nor will we present tips and tricks which can be used to become a better algorithm designer. Various books (e.g. [Knuth 1968, Skiena 1998]) have been written on algorithm design which address these aspects.⁹ Clearly, there is no single “manual approach”, different people will approach ADPs differently and we merely aim to present a rough characterization of a process which we hope most readers will at least partially recognize as having gone through. We most certainly have. Remark that while the literature typically describes algorithms (i.e. the outputs) in great detail, it rarely documents how these algorithms were obtained (i.e. the process). Therefore,

⁹In Section 2.7, we did present templates (metaheuristic frameworks, see Definition 2.32) which can be used to design heuristic search procedures for tackling hard combinatorial optimization problems.

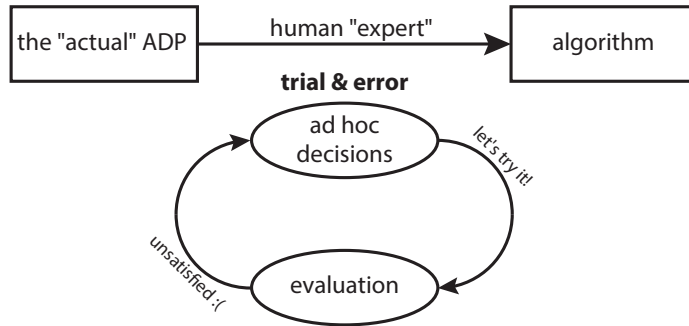


Figure 3.1: An illustration of the manual, trial & error modus operandi by which algorithms are commonly designed.

the characterization we present below is mainly based on our own experiences, discussions with colleagues, and is to some extent inherently speculative and personal.

When tackling a problem, the human designer typically possesses some *a priori*, analytical knowledge that allows him to exclude certain solution approaches, and/or he/she might have a set of algorithmic concepts in mind which could be used to solve it. However, this knowledge alone is often insufficient to solve the ADP, i.e. to derive a design which is as good as any other. This is especially true when considering the design of heuristics (see Definition 2.30) and/or when tackling real-world problems in their natural form (see Section 2.4.3), rather than well-understood academic problems (e.g. sorting a sequence of numbers). As a consequence, when designing an algorithm, the human designer is faced with decisions he/she does not know how to make, which we will collectively refer to as “difficult” design choices. To make such decisions, the human designer relies on intuition, mixed with various degrees of *a posteriori*, empirical knowledge obtained through trial & error experimentation: He/she will first decide to implement some particular algorithm instance, as a best guess. Subsequently, he/she will evaluate the resulting implementation experimentally, and iteratively refine it until satisfactory performance is observed. This trial & error modus operandi is depicted in Figure 3.1. Remark that this process resembles a manual search in design space and we will also refer to it as *graduate student search*.¹⁰

Despite the fact that much of the research we performed in the scope of this dissertation

¹⁰The term “graduate student descent” was quoted by Holger Hoos to describe a similar modus operandi in the context of parameter tuning during a tutorial at IJCAI in 2017 (slide 41: <http://www.cs.ubc.ca/~hoos/Pb0/Tutorials/IJCAI-17/ijcai-17-tutorial-slides.pdf>). A term we believe to be in turn derived from “the graduate student algorithm”, originating from the data compression community [Nelson and Gailly 1995], and which essentially boils down to solving a problem by locking a graduate student up inside a room, and having him try things out until the problem is solved.

was devoted to investigating “how we can do better” (see Section 1.2), we wish to stress the fact that the methodology described above has produced highly performant solvers for many real-world problems. However, we conjecture that underlying these successes is often a tremendous amount of manual effort (or luck), making this *modus operandi* tedious, time-consuming and costly. In Section 7.1, we will argue that it also has a tendency to produce methods which are overly complex. Furthermore, the process itself is rarely documented, making it untraceable: It is often unclear what motivated certain design decisions (natural constraints, expert knowledge, intuition, experimentation, etc.) and what alternatives were considered. Not only do we, in this way, lose potentially interesting information and insights that could otherwise be used to design algorithms in the future, it also makes the process susceptible to personal bias (see Section 1.1, p. 13).

3.3.2 Automated Algorithm Design

Letting computers, rather than humans, design algorithms has numerous potential benefits: Computers are faster, cheaper and unbiased. However, as discussed in Section 2.3.2, computers can only solve problems if they are programmed to do so, i.e. given an algorithm (\sim Lady Lovelace’s Objection [Turing 1950, Section 3.6.6]). As such, having computers design their own algorithms might come across as somewhat paradoxical. The crux is that, rather than providing a computer with an algorithm to solve the target problem, we treat the ADP itself as a computational problem and we provide the computer with “an algorithm to design algorithms”. There are many ways to formulate the ADP as a computational problem. To deal with this polymorphism, we adopt a problem-centric perspective (see Section 2.4.3), where we will view automated algorithm design as solving the “actual ADP” by reduction, which we will visualize as in Figure 3.2.

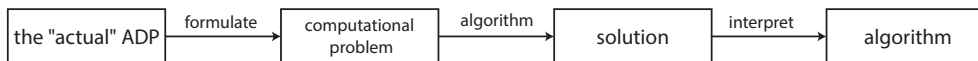


Figure 3.2: Reduction diagram illustrating the automated algorithm design approach.

The idea of automating algorithm design is certainly not new and can be traced back to the origins of the field of Artificial Intelligence (AI) in the 1950s. Ever since, this problem, in one form or another, has been considered in many different communities. In what follows, we will briefly describe the approaches taken in some of these communities.

Program Synthesis (PS) is the task of automatically constructing a program that satisfies a given high-level specification [Gulwani et al. 2017]. The “inception” of PS is

commonly attributed to Alonzo Church, who in 1957 formulated the problem of generating a circuit from mathematical specifications [Church 1963]. In PS, we are usually given a formal specification of the target problem in some logical calculus as a starting point, from which an exact algorithm (commonly expressed in a functional or logical programming language) is automatically derived by means of deduction [Manna and Waldinger 1980]. This deductive approach exploits the analogies between program verification (the task of automatically verifying whether a given program admits to a given specification) and program synthesis. In logical programming languages [Flach 1994] (e.g. Prolog), it is well-known that the logic used for proving that t has some property p , i.e. answer a query of the form $?- p(t)$, can also be used for generating a t with property p , i.e. answer a query of the form $?- p(T)$, where T is a free or unbound variable. As such, the logic for proving that an algorithm a solves a problem p , i.e. answer a query $?- solves(a, p)$ (*program verification*), can be used for generating an algorithm which solves p , i.e. answer the query $?- solves(A, p)$ (*program synthesis*). Intuitively, we start proving the correctness of *any* algorithm A . In order to apply rules of the proof theory, we must make assumptions about A , effectively constraining the set of algorithms the proof thus far is valid for. This process continues until we find a proof of correctness and any algorithm satisfying the set of assumptions made in the process is by construction correct. If we fail to find an admissible proof, no provably correct algorithm exists. Note that an exact algorithm may nonetheless exist, and failure to find it a limitation of the proof theory. See [Basin et al. 2004] for a more in-depth discussion of PS and concrete examples.

In a sense, PS can be viewed as reducing the ADP to a program verification problem (as illustrated in Figure 3.3); which, in turn, is commonly solved by reduction to a graph search problem (see Section 2.7.1.1); e.g. a construction tree, where each successor extends a partial proof by a single step, applying one of the applicable rules.

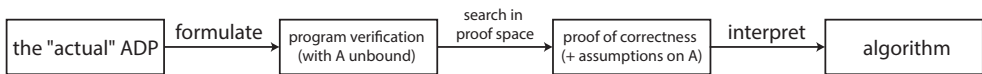


Figure 3.3: Diagram illustrating the Program Synthesis (PS) by deduction approach.

Note that while this deductive approach to PS returns an algorithm which is provably correct, this algorithm is in no way optimized for efficiency. Also, the synthesized programs may be highly complex and difficult to interpret.

Genetic Programming (GP) is another major approach to automated algorithm design in which one “breeds” a population of computer programs for solving a given target problem, using the principles of Darwinian natural selection and biologically inspired operations.

Computer simulations of evolution, with individuals modeled as computer programs, were already performed in the artificial life community in the 1940s [Langton 1997]. In 1950, Alan Turing, in his seminal paper on “Intelligent Machines” [Turing 1950, Section 7], alluded to the possibility of using an evolution-like process to evolve such machines. However, GP as a research community only really took off in 1985 when the first international conference on genetic algorithms (ICGA) was organized in Pittsburgh and Michael Cramer presented what is widely regarded as the first instance of “modern” GP [Cramer 1985]. An approach which was later extended and popularized by John Koza [Koza 1992].

Dropping the metaphors, GP in essence solves the ADP by reducing it to a global function optimization problem $\langle S, f \rangle$ (see Definition 2.4), where the search space S consists of all valid programs in some programming language, and f is the objective function quantifying the quality of any program. This optimization problem is subsequently solved using a Genetic Algorithm (GA, see Algorithm 7). Figure 3.4 illustrates this approach.

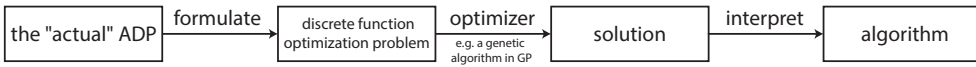


Figure 3.4: Diagram illustrating the genetic programming approach.

A GA takes numerous parameters. For instance e , i.e. the procedure to be used to evaluate the quality of an algorithm f . In Section 2.7.1.1, we already briefly discussed the choice of e in general. In Section 4.1.2 (p. 106), we will discuss the choice of e in the context of the ADP. Beyond e , a GA takes numerous other parameters which describe the search space S ; e.g. the initial population V_0 (or alternatively an initialization operator) and evolutionary operators such as mutation (mutate), recombination (recombine) and selection (select). In the context of GP, the operators used are often an integral part of the GA, which operates on specific program representations. To use such a GA, we must thus represent our candidate algorithms accordingly. Here, tree-based representations [Cramer 1985] are most common, which lend themselves naturally to representing programs in functional programming languages. If this language is Turing complete, S is countably infinite, and $\langle S, f \rangle$ a discrete, but no combinatorial, optimization problem. However, despite us having introduced GAs in the context of combinatorial optimization, GAs do not rely on S being finite and can be applied nonetheless. Remark that the same holds for virtually any of the heuristic search techniques discussed in Section 2.7.2 and while the GP community has historically been biased towards using evolutionary inspired methods, any metaheuristic could in principle be applied to $\langle S, f \rangle$. A concrete example of this is “Ant Programming” [Boryczka 2002], using Ant Colony Optimization (ACO) instead of GAs.¹¹

¹¹We are skeptical of this metaphoric terminology (see also Section 2.7.2.4), as similar conduct in the

Finally, we briefly contrast GP with PS (by deduction). In GP one performs a direct search in program space, while PS does so indirectly by searching the proof space instead. The objective function f in GP can capture a much wider notion of quality (\preceq) than mere correctness. As a consequence, GP is applicable to a wider range of ADPs than PS. Note that this includes the design of heuristics, since the latter are non-exact by definition.¹² Finally, note that GP can also obtain provably exact algorithms ($e \equiv$ program verification). That being said, performance is normally measured empirically in GP; e.g. by executing the candidate program on a given set of inputs and verifying whether its outputs are correct.

Supervised Machine Learning (ML) is the task of inferring a function $f : X \rightarrow Y$ from labeled training data $D \subseteq X \times Y$ [Mohri et al. 2012]. Here, D is typically a set of correct input/output pairs, i.e. $D \subseteq \{(x, y) \mid x \in X \wedge f(x) = y\}$. This task is also known as classification/regression if the range of f is finite/real (\mathbb{R}) respectively. ML techniques typically take D as input and return a predictive model $M : X \rightarrow Y$ minimizing some notion of error w.r.t. f . Supervised ML is naturally viewed as solving an ADP with target problem (X, f) (i.e. a function problem, see Definition 2.2), where the predictive model corresponds to an algorithm (prediction \equiv problem-solving), as illustrated in Figure 3.5.

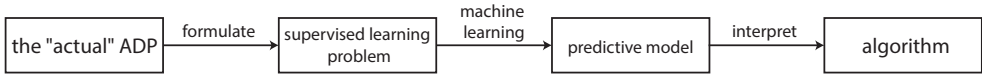


Figure 3.5: Diagram illustrating the supervised machine learning approach.

In practice, supervised ML is not commonly considered an algorithm design technique. This because traditional ML techniques (e.g. support vector machines, random forests, neural networks, etc.) are arguably not well-suited to “learn” the problem relation of typical computational problems (e.g. sorting, prime factorization, etc.). To see why, we briefly characterize how ML techniques work and their limitations. The crux of predictive models is their ability to generalize, i.e. predict $f(x)$ for $(x, f(x)) \notin D$. To do so, these models necessarily make assumptions about f which may cause an inductive bias if they are wrong [Mitchell 1980]. For instance, *parametric* ML approaches (e.g. linear regression) explicitly assume that f is a member of some parametrized family of functions F (e.g. linear functions). Models making stronger a priori assumptions about f will typically need less data to obtain similarly accurate estimates and be less sensitive to the particular subset used for training, i.e. have lower variance. This tension between both sources of prediction errors is

future will doubtlessly lead to terms like Bee, Firefly, . . . Programming, causing unnecessary fragmentation.

¹²The term (generation) hyper-heuristics [Burke et al. 2013, Section 5] has recently become associated with applications of GP in this context.

known as the bias-variance trade-off. When looking at classical ML models, we find that most make relatively strong assumptions, i.e. are highly specialized. For instance, most ML models will make a *smoothness assumption* of some sort, i.e. f maps similar inputs to similar outputs, for some notion of similarity. While this assumption is reasonable for most classical machine learning tasks (e.g. predicting the value of a house based on features such as its age, # rooms, # floors, surface area, etc.), the same does not hold for typical computational problems (e.g. finding the least prime factor), at least not for any straightforward measure of similarity. Due to these difficulties, the supervised ML community has, in the past, rarely targeted typical computational problems. However, recent advancements in representation learning [Bengio et al. 2013], suggest that this may be about to change. So-called “deep learning” techniques [LeCun et al. 2015] have recently been shown capable of automatically inferring complex similarity relations on natural representations in various real-world application areas. While at first sight diverse, closer inspection reveals that most successes involve applications of convolutional/recurrent Neural Networks (NNs) to problems where inputs exhibit spatial/temporal patterns (e.g. images, signals, etc.). More relevant in the context of the ADP are *Neural Turing Machines* (Neural TMs) [Graves et al. 2014]. Conceptually, this model assumes that f is computable by a (simple) Neural TM: a type of recurrent NN having access to an external memory source (\sim TMs). A key feature of Neural TMs is that they are differentiable end-to-end (\sim NNs), allowing them to be trained efficiently (using e.g. gradient descent). Intuitively, we can derive analytically how to modify any Neural TM to improve its accuracy.

Finally, we briefly relate this approach to GP and PS. While in deductive PS we are given a formal specification of our target problem as input, ML methods are (only) given a set of correct input/output pairs and are in that sense more similar to “inductive” PS approaches, e.g. as proposed in [Hamfelt et al. 2001]. While inductive approaches are easier to apply to complex problems, allow us to optimize fuzzy notions of efficacy, they cannot guarantee correctness for target problems with infinite instances. Similar to GP, training a Neural TM using gradient descent can be viewed as a direct search in program space. Unlike GP, which uses a GA, gradient descent is a hill climber whose step function exploits analytical gradient information. While the latter is typically a superior intensification mechanism, it may get stuck in local optima. See Section 2.7.2.2 for a discussion of this issue and how it can be addressed.

3.3.3 Semi-automated Design

In the automated approaches we discussed thus far, the ultimate goal was to minimize the role the human plays in the design process. A computer is to design algorithms from scratch, i.e. all decisions on “how to solve the problem” are left to the computer, and the







Who (human/computer) makes which decisions in each approach?		design approaches		
		manual	automatic	semi-automatic
how much expert knowledge is available to make the design decision?	little			
	plenty			

Figure 3.6: The role of the human/computer designer in the different design approaches.

design space consists of all possible algorithms. Beyond the obvious benefits of reducing human effort and its associated costs; another motivation is that this freedom allows computers to discover truly novel and optimal algorithms, without letting limitations of human creativity/knowledge bias or restrict the computer’s creativity/abilities [Colton and Wiggins 2012]. However, these fully automated approaches have thus far failed to scale up to complex problems. Also, in fully eliminating the human designer, they lose any knowledge he/she might possess w.r.t. solving the problem, which he/she uses to guide his/her manual search in program space, e.g. exclude certain solution approaches.

These observations have given rise to what we will call *semi-automated* approaches to algorithm design. Semi-automated approaches distinguish themselves in that they attempt to maximally utilize expert knowledge to prune the design space a priori. Put differently, *only* difficult design decisions are left open and made by (with the help of) computers. This approach was recently independently proposed/advocated in different articles [Ansel et al. 2009, Hoos 2012, Adriaensen et al. 2014a] as an alternative to, and golden mean between the manual and fully automated approaches. Figure 3.6 contrasts the three approaches. In semi-automated design “automation begins where expert knowledge ends” and it is this use of human “expert” knowledge that allows it to solve problems traditionally out of reach of fully automated approaches.

In the remainder of this section, chapter, and dissertation, we will therefore focus on semi-automated methods. However, this is a rather superficial restriction, as “when exactly a design decision is considered difficult” is a fuzzy concept, i.e. there are many levels of automation (e.g. [Hoos 2012] distinguishes five) ranging from manual to fully automated

design. In addition, while we discussed GP and PS in their original fully automated philosophy, aforementioned scalability issues have given rise to semi-automated variants which support the injection of expert knowledge. More generally, we resort to human intelligence to compensate for the contemporary limitations of artificial intelligence. Note that this is a trend we also observed in other areas of AI, e.g. the Reinforcement Learning (RL) community [Brys 2016]. On the other hand, we also observed how advancements in hardware and/or AI technology can cause shifts in the opposite direction, e.g. deep learning which led to Neural TMs discussed in the previous section.

From a reduction perspective, semi-automated approaches can be viewed as solving an ADP by manually reducing it to an easier problem which is solved automatically. This reduction often involves constraining the design space in one form or another, where the human designer will attempt to retain high quality algorithms, while eliminating inferior ones. In what follows, we will describe some communities solving such “constrained” ADPs, i.e. considering only a subset of all algorithms ($\subset A_U$) as candidate designs.

Algorithm Selection (AS): When considering a limited sample of algorithms A solving some target problem, we find that often no single algorithm instance performs best on all problem instances (see also Section 3.2). This observation gives rise to the following problem: “Given A , which algorithm instance should we use to solve which problem instances?”. This problem is known as the *Algorithm Selection Problem* (ASP) and was first formalized as a computational problem in [Rice 1976, pp. 3]. In a nutshell, in the ASP we are given (some means of collecting) performance observations for each algorithm instance $a \in A$ on a subset of problem instances $x \in X$ and are to find a function of the form $X \rightarrow A$, called a *selection mapping*, relating each problem instance x with the algorithm instance performing best on x . Remark that in the context of the ADP, the resulting selection mapping s is subsequently interpreted as a *portfolio solver* [Leyton-Brown et al. 2003] which given an input x , first computes $a = s(x)$, subsequently computes $y = a(x)$ and finally returns y . Despite being formalized over four decades ago, it is only in the last two decades that the ASP has received widespread attention.¹³ Why? First, the algorithm selection problem was at that point “independently” rediscovered by [Fink 1998]. Second, [Leyton-Brown et al. 2003] brought the ASP (as formulated by John Rice in 1976) to the attention of a wider AI audience, and proposed the use of traditional ML techniques to tackle it. This approach led to numerous successes, most notably SATzilla [Xu et al. 2008], a portfolio solver selecting between various state-of-the-art solvers for the boolean satisfiability problem (SAT, see Section 2.2.1), which won multiple (gold) medals

¹³At the time of writing, Google Scholar™ lists 740 citations of [Rice 1976], with only 24 thereof by articles published before 1998.

at the 2007 and 2009 SAT competitions.¹⁴ Recently, [Bischl et al. 2016] presented the Algorithm Selection library (ASlib), which facilitates comparisons between different ASP solvers (also called *portfolio builders*) and which we believe will stimulate future research.¹⁵ Note that the research in this area generally treats algorithms as black boxes, assumes no a priori knowledge about the relations between different algorithm instances and considers scenarios in which we are to choose between a few, seemingly unrelated solvers.¹⁶ Also, as (efficiently) predicting the best algorithm *a priori* is challenging, one typically settles for non-exact solutions and compares the performance of the portfolio solvers obtained to that of the *Virtual Best Solver (VBS)*, i.e. an oracle solver executing the best algorithm on each instance.

Algorithm Scheduling: When considering a set of fixed budget, or anytime algorithms (see Section 2.6.4), we are not restricted to selecting any one of these, i.e. we can allocate resources more flexibly than is considered in the ASP. For instance, assume we are given 100 seconds to solve our problem and two algorithms a_1 and a_2 . In the ASP, we would run either a_1 or a_2 for 100 seconds. However, one could also run both for 50 seconds. In fact, if a_i is randomized, it may make sense to run a_i twice for 50 seconds. We will call such allocation of resources to algorithms an *algorithm schedule*,¹⁷ and the problem of determining the best schedule for our target problem, the *algorithm scheduling problem*. Remark that while the best algorithm schedule never outperforms the best algorithm instance on any given input, it may, however, have superior average-case performance across all instances. This problem was first extensively treated in [Gomes and Selman 2001] and subsequently the community developed largely in parallel with the ASP community. The main difference is that the former focuses on selecting average-case optimal schedules (as opposed to that of single solvers) for a target distributional problem (as opposed to on a per-instance basis). However, research has been performed at the intersection of both communities [Lindauer et al. 2016], conditioning resource allocations on features of the specific problem instance that needs to be solved, i.e. solving the hybrid per-instance algorithm scheduling problem. Remark that since the best algorithm schedule never outperforms the best algorithm instance on any given input, a portfolio solver selecting algorithm schedules will never outperform the VBS. However, in many

¹⁴<http://www.satcompetition.org/>

¹⁵ASlib supported the ICON Challenge on Algorithm Selection (in 2015) and the Open Algorithm Selection Challenge (OASC) in 2017, which were the two first general competitions for portfolio builders.

¹⁶The number of alternatives ranged from 5 to 31 in the 11 scenarios used during OASC 2017.

¹⁷In the community, algorithm schedules are also commonly referred to as *algorithm portfolios*, due to the similarities between the concept of investment portfolios in economics [Huberman et al. 1997]. However, to avoid possible confusion with the term portfolio solver [Leyton-Brown et al. 2003] used in the ASP community, we will refrain to do so.

scenarios, it is difficult to accurately predict which algorithm instance (or schedule) will perform best on a given input. For instance, we may know too little about our input, i.e. sufficiently discriminative features might not be available, or they might be too expensive to compute. Remark that this uncertainty is also inherent in scenarios where solvers exhibit stochastic behavior (e.g. randomized algorithms). While the same holds for selecting algorithm schedules, in executing multiple algorithms, we hedge our bets, reducing the cost associated with making wrong decisions. These challenges have also motivated *dynamic* algorithm scheduling approaches where, instead of committing to a certain schedule prior to execution (i.e. *statically*), we start execution and make future allocation decisions based on the observations made during the execution thus far (i.e. *dynamically*).¹⁸ The crux is that during execution more information becomes available which can be used to make more informed decisions. In scenarios with inherent uncertainty, an algorithm executing the best dynamic schedule may very well outperform the VBS. For instance, [Horvitz et al. 2001] monitored a set of features during the initial phase of the execution of a solver and showed how this information can be used to train a classifier to accurately classify runs of that solver as long or short. In [Kautz et al. 2002] a dynamic restart schedule based on this model was presented. Another example is [Gagliolo and Schmidhuber 2006] which divides execution in multiple time frames and dynamically decides, for each time frame, about the relative allocation of resources to the algorithms in the portfolio.

Algorithm Configuration (AC): In algorithm selection (scheduling) we consider the following scenario: We are to solve a (distributional) target problem and are given multiple solvers for doing so. A closely related scenario is encountered when we are given a single solver, but that solver has multiple configurable parameters/components, e.g. a metaheuristic framework. As discussed in Section 2.3.3 such algorithmic framework can be seen as a family of algorithms, each configuration corresponding to a single algorithm instance. To use such framework, we must configure it appropriately, i.e. choose a configuration. This problem is known as the “Algorithm Configuration Problem” (ACP). Arguably, the first treatment of the general ACP as a computational problem was [Birattari et al. 2002], proposing the F-Race procedure as a solver (called a *configurator*). Subsequently, a variety of different approaches to the ACP were explored, resulting in a number of prominent configurators, e.g. ParamILS [Hutter et al. 2007b, Hutter et al. 2009], iRace [Balaprakash et al. 2007, López-Ibáñez et al. 2011], GGA [Ansótegui et al. 2009] and SMAC [Hutter et al. 2011] (see Section 4.1.2, p. 115, for a more detailed discussion). Remark that the configuration setting can be viewed as a special case of the algorithm selection setting, where algorithm instances are represented as configurations. Vice versa, the latter can

¹⁸Some authors use the terms offline/online instead of statically/dynamically. However, as these terms refer to very different concepts in the ML community (see Section 3.5.2), we will refrain to do so.

be viewed as configuring a single categorical (procedural) parameter. One property that makes the configuration setting particularly challenging is the fact that the configuration space Θ grows exponentially in terms of the number of parameters. Furthermore, a single parameter with a continuous domain gives rise to an uncountably infinite algorithm space. Therefore, the ACP community, unlike the algorithm selection community, has focused on techniques capable of handling large/infinite algorithm spaces. To do so, these techniques rely on the assumption that the performance of two similar configurations (e.g. having similar values for most parameters) are positively correlated. On the other hand, the ACP community has traditionally been concerned with finding a single configuration maximizing average-case performance across a set of problem instances, as opposed to selecting the best configuration on a per-instance basis, as is considered in the ASP community. However, here again, research has been performed at the intersection of both communities [Hutter and Hamadi 2005, Kadioglu et al. 2010, Xu et al. 2010], conditioning the choice of configuration on features of the specific problem instance that needs to be solved, i.e. solving the hybrid per-instance algorithm configuration problem. In analogy to our discussion in the previous paragraph, it may be difficult to predict which configuration performs best prior to execution. In addition, it has been shown that varying the value of certain parameters over the course of an execution may be beneficial (e.g. lowering the temperature parameter in the Simulated Annealing (SA) framework). This has motivated *parameter control* strategies [Eiben et al. 2007, Karafotias et al. 2014] which, instead of fixing the values of parameters before execution (i.e. statically), vary the value of parameters during the run of the algorithm (i.e. dynamically).

Optimizing Compilers: As discussed in Section 2.3.2, these days we describe algorithms in high-level programming languages, e.g. Java, C++, Python, etc. While easier to write and portable; an algorithm in this form (*source code*) cannot be understood by any (physical) machine, i.e. it must be translated to *machine code*. This translation is commonly performed automatically using tools called compilers. Typically, there are many possible translations to machine code, not all of which are as good. In what follows, we will use the term *optimizing compiler* to refer to those compilers that not merely translate source to machine code, but actively try to optimize the quality of the resulting executable in the process. Clearly, such *optimizing compiler* can be viewed as automatically solving a constrained ADP. This also implies that, unless one programs in machine code, the predominant “manual” design approach described in Section 3.3.1 actually involves the use of automated tools. However, these days their use is rarely perceived as design automation. Perhaps because these tools are integrated into the programmer’s workflow to the extent he/she is no longer aware of using them. In [Parnas 1985] it has been noted that what is considered *automatic programming* (design automation) has changed over time. In the

1940s the term was used to describe automation of the manual process of punching paper tape. Later it was used to refer to high-level programming languages¹⁹ and currently the term is predominantly associated with fully automated approaches such as PS and GP. These observations led David Parnas to conclude that "automatic programming" has always been a euphemism for programming in a higher-level language that was available to the programmer at the time. Remark that the semi-automatic programming paradigm, proposed independently in [Ansel et al. 2009, Hoos 2012, Adriaensen et al. 2014a], does not (only) advocate increased automation, but rather (to prioritize) the automation of these design decisions which are difficult for the human designer to make. More generally, they attempt to give the programmer full control over which decisions he wants to make or not. For instance, a system expert may be able to make "low-level" design decisions which are made by most modern compilers (e.g. whether to inline a certain function call), while he/she may be uncertain about "high-level" design decisions (e.g. choice of sorting algorithm) which contemporary compilers cannot make. To better understand the limitations of traditional programming languages, we will briefly discuss how their optimizing compilers operate. Optimizing compilers are instances of Search-based Software Engineering (SBSE), a term coined in [Harman and Jones 2001] to denote the collection of techniques applying search procedures (see Section 2.7) to automate part of the software engineering process. SBSE also includes other forms of program optimization, e.g. automated refactoring. These can be viewed as solving program optimization problems by reducing them to graph search problems (see Section 2.7.1.1) where vertices correspond to programs (v_0 the original program) and edges to program transformations. Here, traditionally, only transformations are applied which (1) retain functionality (i.e. same output for any given input) and (2) improve the efficiency of the program. In standard optimizing compilers, whether a transformation satisfies (1) and (2) is evaluated analytically, e.g. through a combination of static code analysis and hard-coded transformation rules, and does not involve actually executing the program. As discussed before, it is difficult to determine a priori whether a pair of programs is functionally equivalent and if so, which of the two is more efficient. This results in a sparsely connected graph, where the set of reachable programs is small. Also, functionality preserving transformations are overly restrictive; e.g. do not permit other correct outputs, or optimizing the efficacy of non-exact procedures. It is our belief that the key to enabling a more flexible, higher-level of programming is complementing this analytical with empirical information, i.e. compilers that collect and learn from performance data. An initial example of such a compiler can be found in Petabricks [Ansel et al. 2009], a programming language extension (for C++ and C) which provides language constructs that allow programmers to express design choices (and their alternatives) in code, i.e. making algorithmic choice a first-class citizen. Note that such

¹⁹In fact, one of the first high-level programming languages was called "Autocode".

specification enables transformations beyond simple functionality preserving ones. Next to this program, the user provides the compiler with a performance measure and a set of representative training problem instances. The compiler will subsequently determine experimentally which decisions perform best for which input sizes.²⁰ A similar language extension (for C++ and Java) was proposed in [Hoos 2012], where first a source-to-source compiler (which they call a weaver) transforms the program with open design choices into an ordinary parametrized program, which is subsequently configured automatically.

3.4 What Information Does the Designer Use?

In this section, we will discriminate design approaches based on the kind of information the designer uses in solving the ADP. Here, we roughly distinguish between analytical and empirical approaches, using *a priori* and *a posteriori* information, respectively.

3.4.1 Analytical

Pure analytical approaches only use *a priori* information: What is known or can be derived by means of logical reasoning. Intuitively, *a priori* information is “whatever can be said about candidate designs without actually using them”. For example, we know that selection sort (see Algorithm 2) sorts any sequence of n numbers correctly, and will need exactly $\frac{(n-1)n}{2}$ comparisons to do so. Pure analytical approaches are predominant in the program synthesis and optimizing compiler communities.

However, the availability of this kind of information is often limited. In particular, it is difficult to say *a priori* which of two algorithms will perform better on an input, let alone across a set/distribution of such instances. To do so nonetheless, the designer will (1) collect additional *a posteriori* information (see next section) and/or (2) make assumptions. For example, optimizing compilers commonly assume that certain code-transformations will result in more efficient programs. Clearly, faulty assumptions, may have a detrimental impact on the performance of our design method.

3.4.2 Empirical / Simulation-based

Empirical approaches, use *a posteriori* information: Experience obtained through experimentation. For instance, following an empirical approach, the designer would use both algorithms and justify his/her beliefs about their relative performance based on his/her obser-

²⁰Petabricks was later extended to support design decisions conditional on arbitrary, programmer-specified features of the input [Ding et al. 2015] (\sim algorithm selection).

vations (e.g. the time they took, their output(s), etc.). As empirical approaches typically involve algorithm execution, we will also commonly refer to them as being *simulation-based*. Simulation-based approaches are predominant in the genetic programming, algorithm selection/scheduling/configuration communities.

However, this empirical/simulation-based approach has limitations of its own. For example, collecting experimental data (involving numerous algorithm executions) may take a lot of time. Also, we have *no guarantees* that the observations we make can be generalized. Remark that while empirical approaches cannot prove general statements, e.g. “algorithm a is correct”, a single observation can disprove them. In that sense, observations not falsifying a theory, are often seen as supporting it. Also, in some settings, the use of statistics may allow us to derive “how likely” a theory is to hold, given the data.

3.4.3 Who Uses What Information?

In summary, the designer should attempt to maximally exploit both sources of information. Recall that our focus, in this work, will be on semi-automated design (see Section 3.3.3). Here, there are two designers: one human, the other machine. Collecting and interpreting experimental data is time-consuming, tedious, requires little ingenuity, and is as such an obvious candidate for automation. In contrast, the human designer possesses a wealth of “knowledge” which may be difficult to encode into a computer program, which he should use in the reduction, e.g. to constrain the design space.

3.5 When Does Design Take Place?

In this section, we will discriminate different (simulation-based) design approaches based on when the design takes place. Here, we roughly distinguish between two methodologies, which we will denote the *offline* and *online* approach, respectively. While the former is probably the most common, examples of both can be found in most of the communities discussed in Section 3.3 and beyond, albeit the terminology used may vary. We discuss the relative strengths and weaknesses of both approaches and characterize a third *semi-online* approach which we argue to be preferable in many practical settings.

3.5.1 Offline

In the offline setting, the algorithm is designed before it is being used, i.e. “solving the ADP” precedes “using the solution”. In simulation-based approaches, in particular, we have an initial training phase during which the “best” design is determined based on an initial

subset of *training instances*, after which this design is used for all further problem instances. For example, the following workflow is prototypical for offline algorithm configuration:

1. One is given a parametrized algorithm.
2. One collects a set of problem instances X' representative for those one would like to solve in practice, i.e. $X' \sim \mathcal{D}$.
3. One determines a configuration which performs well on X' .
4. One uses this configuration to solve further problem instances.

Various factors may make this offline approach infeasible in practice. First, it might be difficult to obtain a sufficiently large set of representative training inputs. Furthermore, if performance depends on it, one must also have access to a representative execution environment (e.g. machine). Second, as simulation-based approaches are typically resource-intensive, the training cost may outweigh later gains from using a more efficient design. Also, as these methods are generally at best asymptotically correct (see Section 2.6.2) there is no natural end to this training phase, making it difficult to decide “when we have trained enough”. After training, the design does not change anymore, it is not modified to account for possible changes in the usage context (\mathcal{D} , execution environment, etc.) or to take advantage of further experience gained while using it.

3.5.2 Online

Aforementioned shortcomings of the offline approach have motivated an online (or lifelong) learning approach which contrasts itself by continuously refining the design “while it is being used”.²¹ In its most literal interpretation, we consider a sequence of problem instances we would like to solve, each of which must be solved exactly once, in order, and we wish to minimize the total cost associated with doing so. In the online approach, we further discriminate between

cross-input learning methods which refine the design *after* solving a problem instance.

within-run learning methods which refine the design *while* solving a problem instance.

Intrinsic to the online approach is an exploration vs. exploitation trade-off. While exploration is essential to refine the design, it also comes at a cost, i.e. using a design which is probably worse. In the offline approach, exploration is restricted to the training phase,

²¹Remark that, by this definition, only simulation-based approaches can be used in this setting.

where errors are permitted.²² Finally, the fact that each problem instance can be solved only once complicates comparison between designs.²³

3.5.2.1 Learning Algorithms

Remark that if we embed an online design process into the design itself, the system as a whole may be viewed as a *learning algorithm* which gets better at solving problems with experience. Such learning algorithm can be applied to a wide variety of usage context, and will automatically adapt to them over time. This is clearly an aesthetically pleasing concept, and online design techniques are often presented in such an integrated form. In recent years, many “learning” variants of well-established algorithms have been proposed; E.g. TS [Battiti and Tecchiolli 1994], SA [Ingber 1996], GA [Harik and Lobo 1999], ACO [Randall 2004] and many more. Most commonly, these learning variants replace some parameters of a framework by embedded parameter control strategies which dynamically adapt their values. Adjectives commonly associated with such methods are “dynamic”, “adaptive”, “reactive”, “learning”, “self-tuning”, “parameterless”, “off-the-shelf”, “out-of-the-box”, etc. A somewhat comprehensive overview of this area of research, which includes selection hyper-heuristics (see Section 2.7.2.5), can be found in [Battiti et al. 2008]

A critical perspective (opinion): We are personally rather skeptical about this trend. First, we feel that the focus on integral solutions (*self*), encourages an unnecessarily tight coupling between meta-optimization, dynamic decision mechanisms and algorithm specific aspects. Second, the vast majority of these techniques do *not* implement cross-run learning, i.e. they do not transfer experience across runs. Dropping any learning metaphors, they are entirely ordinary algorithms. Finally, often ‘techniques’ make bold claims, yet it is often unclear whether performance gains compared to default parameter values, are due to the fact that they over the course of the execution (1) use multiple different values (\sim exploration), (2) use values they learned work well (\sim exploitation) and this (a) for a specific input, or (b) for the specific dynamic execution states encountered. Two examples:

- [Karafotias et al. 2014] uses RL techniques to control numerous parameters of an EA. He shows that the RL approach outperforms the default configuration, yet a

²²Clearly, the choice of the duration of this training phase presents a similar trade-off.

²³For instance, the blocked evaluation strategy which we will discuss in Section 4.1.2, p. 109, cannot be applied.

- random parameter adaptation scheme was shown to perform nearly as good ²⁴
- Reactive Tabu Search (RTS) [Battiti and Tecchiolli 1994] is a learning variant of Tabu Search (TS, see Section 2.7.2.2) which dynamically adapts the tabu tenure parameter. [Mascia et al. 2014b] presents an empirical investigation of “what RTS actually learns” and found evidence for (2a), but none for (2b). [Mascia et al. 2014b] also shows that similar performance gains can be obtained by setting the tabu tenure parameter statically, conditioned only on features of the specific input.

In summary, these approaches risk crossing the fine line between “parameterless” and “hard-coding ad hoc design decisions” and between “integrating design automation techniques” and “manually designing an algorithm”. This gives rise to the question: Can these methods even be considered design automation? We will not give a definite answer here, but independent of the answer, methods lacking cross-input transfer, (at best) design disposable algorithms and are therefore not the focus of our dissertation.

Generation vs. selection hyper-heuristics (revisited): In Section 2.7.2.5, we have pointed out that generation and selection hyper-heuristics are, at first sight, two very different concepts, the former being an automated design technique, the latter an algorithm for our target problem. However, these concepts may overlap in the context of learning algorithms (without cross-input transfer) which can both be viewed as (A) “an algorithm for our target problem” and (B) “designing an algorithm for a target problem instance”. On the one hand, selection hyper-heuristics are often viewed as learning online how to select search operators. On the other hand, generation hyper-heuristics are frequently used to design a heuristic for only a single problem instance. As doing so, using a simulation-based technique (e.g. GP), requires solving this instance multiple times, it is only logical to also keep track of the best solution found thus far and return it together with the best design. However, in this setting, we argue that typically the solution, and not the disposable design, is of actual interest. This is why we will tend to view such methods as (A) rather than (B) in this dissertation; e.g. we characterized selection hyper-heuristics as just another metaheuristic in Section 2.7.2.5. Note that various (other) metaheuristics could also be viewed as designing disposable algorithms. For instance, ACO could be adapted to return anytime a simple stochastic construction heuristic based on the current pheromone matrix.

²⁴[Karafotias et al. 2013] argued for the importance of comparing to random adaptation schemes to help distinguishing between (1) and (2). This can be seen as an instance of accidental complexity analysis (see Section 7.2), a practice we have advocated in [Adriaensen and Nowé 2016a] and which we will discuss and illustrate in Chapter 7.

3.5.3 Semi-online

While the offline approach may sometimes be impractical, we believe that true online learning is again overly restrictive and that often what we will call a semi-online approach suffices in practice. More specifically, in practical settings where:

1. We can solve problem instances multiple times.
2. Cheap resources are available, e.g. an idle processing unit.

For instance, consider a scenario in which one cares about minimizing response time: When we ask the system to solve a problem we want a prompt and high-quality result. On the other hand, there is also time in between requests (e.g. overnight) during which computational resources are readily available, i.e. cheap.

In the semi-online approach, time/quality critical requests are always served using the best-known design, while cheap resources are allocated to finding better designs. Like in the offline setting, we maintain a strict separation between design (exploration) and use (exploitation). Similar to the online setting we continuously refine the design over its lifetime, possibly adapting to changes in usage context.

In what remains, we will argue that in practical settings satisfying (1) and (2), “online” is not the property we are looking for, but rather “anytime” (see Section 2.6.4), i.e. at any point in time we can query the design process for the best design found thus far.

Offline anytime → semi-online: We will do so by showing that every anytime offline approach can be transformed into a semi-online approach, as follows:

1. start the anytime offline design process in a separate thread.
2. for each request to solve x (asynchronous):
 - (a) obtain the best-known design a_{best} (anytime solution) from the design process.
 - (b) solve x using a_{best} .
 - (c) return the solution obtained to the client.
 - (d) add x to the set of training inputs and (optionally) update experimental data (experience) to include the observations made in 2b.

Following this approach, all requests will be handled using the best design found thus far, while continuously refining this design and extending the training set (\sim semi-online).

3.6 Summary

In this chapter, we have introduced the Algorithm Design Problem (ADP) and have presented a broad overview of how it is currently being solved. Below, we summarize the content covered in each section.

In Section 3.1, we characterized the ADP as the problem of finding the best algorithm to solve a given problem. Here, we introduced this notion intuitively, explained its role in our perspective on algorithm design, and presented a somewhat formal definition thereof.

In Section 3.2, we examined what attributes make one algorithm “better” than another. In Section 3.2.1, we discussed performance. Here, we first considered the performance of an algorithm on a single input, i.e. its ability to solve a given problem instance using less resources (efficiency) and/or better (efficacy) than others. Subsequently, we introduced the concept of a “disposable” algorithm, designed to perform well on a single input, and contrasted it with “reusable” algorithms which are designed to perform well across a wide range of inputs. We continued to discuss how to measure performance across a range of inputs, introducing the notion of average-case performance and distributional problems, in the process. In Section 3.2.2, we argued that while performance is important, it is “not the only thing that matters” and argued for simplicity as another factor strongly affecting the reusability of an algorithm, an argument which we will continue in Section 7.1. In Section 3.2.3, we discussed the tension between “time and space-efficiency”, “efficiency and efficacy”, “peak performance and generality” and “simplicity and performance”. In our discussion of “peak performance vs. generality”, we considered the no free lunch theorem for optimization and discussed its relevance in the context of solving the general combinatorial optimization problem. In summary, we found that while it is trivial to find examples of these trade-offs for particular algorithms, it is largely an open problem whether, and to which extent, these are intrinsic for any problems actually encountered in practice.

Subsequently, we turned our discussion towards how the ADP is currently being solved. In Section 3.3, we discriminated different solution approaches based on the role the human designer is envisioned to play in the design process. In Section 3.3.1, we briefly characterized the predominant manual approach, discussing its strengths and weaknesses. In Section 3.3.2, we discussed the possibility of, and potential benefits associated with, automating algorithm design, i.e. solving the ADP automatically. Subsequently, we presented a high-level characterization of various research “communities” which have considered this problem in one form or the other. First, we considered those tackling the ADP in its unconstrained form, i.e. without a priori restricting the space of possible designs. Here, we discussed historically significant approaches such as “program synthesis” and “genetic programming”, but also more recent approaches such as “neural Turing machines”, originating from the machine learning community. In Section 3.3.3, we discussed those tackling

“constrained” variants of the ADP, such as the “algorithm selection”, “algorithm scheduling”, “algorithm configuration” and “optimizing compilers” communities. However, prior to this discussion, we explained that we focus on these constrained approaches because (A) the unconstrained variant is often computationally intractable and (B) we typically possess knowledge allowing us to exclude certain designs a priori. Here, we also presented an alternative perspective on our categorization, viewing un/constrained as fully-/semi-automated solution approaches, respectively, to a naturally unconstrained ADP.

In Section 3.4, we discriminated different solution approaches based on the kind of information the designer uses to justify his/her decisions. Here, we roughly distinguished between pure analytical and empirical (\sim simulation-based) approaches, which we briefly discussed in Section 3.4.1 and Section 3.4.2, respectively. While the former only use a priori information, what is known, or can be derived by means of logical reasoning, the latter complement this with a posteriori information, experience obtained through experimentation, i.e. actually testing candidate designs. Here, we discussed the limitations of both sources of information and the importance of combining them in solving the ADP. In Section 3.4.3, we briefly discussed doing so in the context of semi-automated design.

In Section 3.5, we distinguished between different solution approaches based on “when” the design takes place, discriminating between offline (“design before use”) and online (“design during use”) approaches, which we discussed in Sections 3.5.1 and 3.5.2, respectively. In Section 3.5.2.1, we discussed “learning algorithms”, i.e. algorithms that integrate online learning mechanisms, and we expressed our concerns about the lack of rigor in some of the research performed in this area. We examined the strengths and weaknesses of both off- and online approaches and, in Section 3.5.3, we argued that in many practical settings it is possible to combine the benefits of both in what we call a semi-online (“design in spare time”) approach. In this setting, we showed that “anytime” and not “online” is the key property of interest, describing how every offline anytime approach can be turned into a semi-online approach.

4 | Design by Algorithm Selection

In Chapter 3, we have introduced the Algorithm Design Problem (ADP, see Section 3.1) and have presented an overview of some of the various communities which have contributed to solving it automatically, i.e. “automate algorithm design”, thus far. Here, our coverage of the specific algorithmic techniques employed in each of these communities was relatively high-level. This was intentional, as we would like to argue that there is a significant overlap: Different communities are tackling closely related computational problems and evidently face similar challenges in doing so. It is our belief that in discussing these techniques on a community level we would only further encourage this fragmented perspective.

In this chapter, we will try to present a more unified, problem-centric perspective, where we will consider generic formulations of the ADP and discuss how to solve problems of this form. As motivated in Sections 3.3, 3.4 and 3.5, we will focus here on semi-automated, simulation-based, anytime methods for solving the ADP offline. As we see it, most of these approaches can be viewed as solving the ADP by reducing it to a variant of one of the following three computational problems, around which our discussion will revolve:

- Per-set Algorithm Selection Problem (set-ASP, see Section 4.1)
- Per-input Algorithm Selection Problem (input-ASP, see Section 4.2)
- Dynamic Algorithm Selection Problem (dynamic ASP, see Section 4.3)

In Section 4.4, we relate these approaches, discussing their respective strengths and weaknesses. Finally, in Section 4.5, we summarize the content covered in this chapter.

Why selection and not... e.g. configuration? We use the “algorithm selection” terminology here to stress that in these problems we are (1) to choose from alternatives, rather than design algorithms from scratch (\sim constrained ADPs), (2) to select a single instance (vs. algorithm scheduling, p. 89), and (3) not assuming any specific algorithm representation. The latter is why we favor it over the AC terminology (p. 90), despite the fact that many of our concrete examples are taken from the AC community. Note that only one of these formulations, i.e. the input-ASP, actually corresponds to the ASP, as formalized in [Rice 1976], and as commonly considered in the AS community (p. 88).

Sorting ADP: We will illustrate some of these techniques experimentally as well. Here, we will be using “sorting sequences of integers in ascending order” as a target problem and are to find a solver sorting a wide range of sequences (time) efficiently on our machine (see Appendix A.1). Also, we are willing to spend more time sorting longer sequences. In our illustration, we do so to avoid that poor algorithmic choices for shorter sequences would go unpunished just because relative time differences are smaller. In real-world settings, one could imagine offering “sorting as a service”, where customers pay based on input-size.

4.1 Design by Per-set Algorithm Selection

The first approach we consider, solves the ADP by set-ASP reduction (see Figure 4.1).

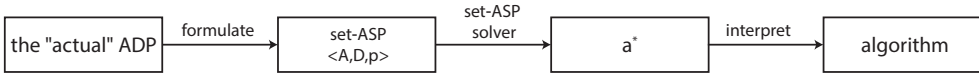


Figure 4.1: Diagram illustrating the design by set-ASP reduction approach.

4.1.1 Formulating the ADP as a set-ASP

In the set-ASP, we are to find an algorithm instance maximizing average-case performance for a given distributional target problem in some constrained algorithm space ($\subseteq A_U$)

Definition 4.1: Per-set Algorithm Selection Problem (set-ASP)

Instances of the set-ASP are defined by triples $\langle A, \mathcal{D}, p \rangle$ where

A : a set of algorithms (*algorithm space*).

\mathcal{D} : a distribution over a set of inputs X (*input distribution*).

p : a function $X \times A \rightarrow \mathbb{R}$ quantifying the performance of an algorithm on an input (*per-input performance measure*).

Candidate solutions are elements of A . In the set-ASP, given $\langle A, \mathcal{D}, p \rangle$, we are to find an algorithm instance a^* maximizing average-case performance, i.e. satisfying $o(a) \leq o(a^*)$, $\forall a \in A$, where $o(a) = \mathbf{E}_{x \sim \mathcal{D}} p(x, a)$ as per Definition 3.2.

Problems of this form are commonly tackled in the AC and GP (p. 83) communities, while in the AS community one traditionally considers the per-input ASP (see Section 4.2). Here, the per-set ASP has also been referred to as the “per-distribution” or “winner-takes-all” approach to algorithm selection [Leyton-Brown et al. 2003].

Next, we will make some additional assumptions about the form in which instances $\langle A, \mathcal{D}, p \rangle$ are provided to the solver:

A may be a finite or an infinite set, which may be structured or not. Truly general set-ASP solvers can therefore not critically rely on (1) A being finite, or (2) A being structured (e.g. instances being represented in any particular fashion). In general, as large, unstructured sets cannot be searched, we find that prior art always assumes either (1) or (2) and we structure our discussion in Section 4.1.2 accordingly.

\mathcal{D} : We do not assume \mathcal{D} to be given explicitly. Rather, we will assume to be given a procedure $\mathcal{D}.\text{sample}()$ which can be used to generate samples according to \mathcal{D} . The procedure itself is treated as a black box. If \mathcal{D} is a uniform distribution over a finite set of inputs X , \mathcal{D} may also be provided as a collection of inputs (\sim training inputs).

p : We do not assume p to be given explicitly. Rather, we will assume we are given a possibly stochastic evaluation procedure $p.\text{eval}$ satisfying $p(x, a) = \mathbf{E}[p.\text{eval}(x, a)]$. Put differently, rather than being given the actual performance of every algorithm on every input, we are given some procedure $p.\text{eval}$ which can be used to obtain possibly noisy, unbiased observations thereof. The evaluation procedure itself is treated as a black box. However, most commonly $p.\text{eval}(x, a)$ executes a given algorithm a on input x and evaluates the “desirability” $p(e)$ of this execution.¹ For instance, if p is average runtime, $p(e)$ will be the time an execution took and the evaluation procedure effectively samples the RTD (see Section 3.2.1). If p is solution quality, $p(e)$ is the quality of the solution obtained and the evaluation procedure samples the SQD. Note that variance in performance observations may either stem from stochasticity in the execution of the (randomized) algorithm or noise in measuring $p(e)$ (e.g. measuring runtime), or both. Remark that $\mathcal{D}.\text{sample}$ and $p.\text{eval}$ together specify o and satisfy $o(a) = \mathbf{E}[p.\text{eval}(\mathcal{D}.\text{sample}(), a)]$.

¹In Section 6.2 we describe a variant of the set-ASP (ACP) which opens up this black box (wb-ACP).

Example - Sorting set-ASP: Before discussing how to solve the set-ASP, we will briefly discuss the set-ASP reduction depicted in Figure 4.1, and specifically we will formulate our sorting ADP as such, i.e. specify our choices for A , \mathcal{D} and p .

A: The algorithm space A can be an arbitrary subset of the design space A_U , allowing the designer to exclude any designs he/she assumes to be suboptimal. To retain optimality, we must not eliminate all optimal designs in the process. In our sorting example, we will consider eight well-known sorting algorithms: selection sort, bubble sort, insertion sort, merge sort, quicksort, heapsort, radix sort and Tim sort.

o: Our choice of \mathcal{D} and p determines o which induces a total order on A . We retain optimality iff $\forall a^* \in A : (o(a) \leq o(a^*), \forall a \in A) \implies (a' \preceq a^*, \forall a' \in A_U)$. More generally, it is desirable that $\forall a, a' \in A : o(a) \leq o(a') \implies a \preceq a'$.

D: In most real-world settings \mathcal{D} , i.e. the actual likelihood that we are asked to solve any given instance (e.g. sort any given sequence) of the distributional target problem, is not known explicitly. Instead, we most commonly collect a set of training instances X' (sequences of integers) which we believe to be representative for those we wish to solve in practice. Alternatively, we may have a procedure which can be used to generate representative problem instances. In our sorting example input sequences to be sorted are generated by the following process: First, the length of a sequence is drawn from a log-uniform distribution in $[2, 2 * 10^5]$. The elements appearing in the input sequence are *all* drawn uniformly at random from *one* of the following sets (all equally likely):

- (A) $[-10^9, 10^9]$
- (B) $[b - a, b + a]$ where a is drawn log-uniformly from $[0, 10^9]$ and b uniformly from $[-10^9 + a, 10^9 - a]$.
- (C) V whose elements are selected uniformly at random from $[-10^9, 10^9]$ and $|V|$ is log-uniformly distributed in $[1, |x|]$.

Finally, the order in which these elements appear in the sequence is determined using one of the following schemes (again, all equally likely):

- (i) Elements appear in the order in which they were generated (\sim randomly).
- (ii) First the elements are sorted in ascending order. Subsequently, n pairs of elements (selected uniformly at random) are swapped, where n is drawn from a log-uniform distribution over $[1, |x| \log(|x|)]$.²
- (iii) First the elements are sorted in ascending order. Subsequently, n pairs of successive elements (selected uniformly at random) are swapped, where n is drawn from a log-uniform distribution over $[1, |x| \log(|x|)]$.

²After $|x| \log(|x|)$ random swaps, the order of the elements in the sequence is roughly random.

- (iv) Same as 2, but sorted in decreasing order.
- (v) Same as 3, but sorted in decreasing order.

This results in a rather heterogeneous mix of input sequences and as such encodes our desire for generality, i.e. “sort a wide range of sequences”. To ensure reproducibility, the actual $\mathcal{D}.\text{sample}()$ will resample (with replacement) a sequence uniformly from a pool of 15000 sequences (1000 in each category) generated using the procedure described above.

p : Finally, we discuss our choice of per-input performance measure p . In our sorting example, our evaluation procedure will use a to sort x , measure how long this took and normalize this measurement, dividing it by minus the length of x , i.e. we choose $p(x, a) = \mathbf{E}[\frac{t(e)}{-|x|}]$. Remark that the normalization by $|x|$ encodes our “willingness to spend more time sorting longer sequences”, e.g. taking one second to sort 1000 elements is as desirable as taking one minute to sort 60000. The negation reduces what is naturally a minimization to a maximization problem. To ensure reproducibility and reduce the time required to run our experiments, the actual $p.\text{eval}(x, a)$ will resample (with replacement) a value uniformly from a pool of 50 performance observations which were a priori obtained by evaluating a on x as described above.

Table 4.1 shows for each sorting algorithm a the mean (i.e. $o(a)$) and standard deviation of R_a : the pool of $50 * 15000 = 750000$ performance observations which $p.\text{eval}(\mathcal{D}.\text{sample}(), a)$ resamples from uniformly (with replacement). Unsurprisingly, we find that Java’s built-in Tim sort performs best on average.

Table 4.1: The mean and standard deviation of the performance observed for the eight sorting algorithms in our algorithm space.

algorithm (a)	$\mu(R_a) = o(a)$	$\sigma(R_a)$
bubble sort	-99885	-404144
selection sort	-42995	-277923
insertion sort	-58514	-480923
merge sort	-291	-342
quicksort	-9663	-350559
heapsort	-305	-418
radix sort	-580	-854
Tim sort	-280	-1868

4.1.2 Solving the set-ASP

The set-ASP is naturally viewed as a Global function Optimization Problem (GOP, see Definition 2.4) with search space A and objective function o . In what follows, we will therefore start discussing approaches which solve the set-ASP by reducing it to the GOP, and gradually introduce more specialized variants. Here, we initially focus on solving set-ASPs with small (finite) A 's, later widening our scope to tackling larger (infinite) A 's.

How to compute $o(a)$? We already discussed the GOP reduction in the context of GP (p. 83) and the set-ASP reduction is essentially analogous. One aspect of the reduction we did not yet discuss, and which will be our focus here, is that even the most generic black box optimizers assume to be given a procedure computing $o(a)$, $\forall a \in A$, i.e. the average-case performance of any given algorithm. However, we cannot compute $o(a) = \mathbf{E}_{x \sim \mathcal{D}} p(x, a)$ exactly in the set-ASP (at least not in general) since \mathcal{D} and p are not provided to us explicitly, but rather we are given procedures $\mathcal{D}.\text{sample}$ and $p.\text{eval}$ respectively, which are treated as black boxes. Fortunately, as $o(a) = \mathbf{E}[p.\text{eval}(\mathcal{D}.\text{sample}(), a)]$, and assuming $\mathbf{Var}[p.\text{eval}(\mathcal{D}.\text{sample}(), a)]$ is finite, we can compute $o(a)$ asymptotically (see Section 2.6.2). In a nutshell, using $\mathcal{D}.\text{sample}$ we can obtain a sample of inputs $X' \sim \mathcal{D}$ and using $p.\text{eval}(x, a)$ we can obtain a sample of performance observations $R'_{x,a}$ for algorithm a on every input $x \in X'$, which can in turn be used to compute

$$\bar{o}(a) = \frac{1}{|X'|} \sum_{x \in X'} \bar{p}(x, a) \quad \text{where} \quad \bar{p}(x, a) = \frac{1}{|R'_{a,x}|} \sum_{r \in R'_{a,x}} r \quad (4.1)$$

In our sorting example this would boil down to generating a set of sequences, sorting each of them at least once (but possibly repeatedly) using a , averaging the performance observations for each sequence and finally averaging these per-sequence estimates to obtain $\bar{o}(a)$. This estimator is consistent, i.e. guaranteed to converge in probability to $o(a)$ as $|X'| \rightarrow +\infty$, assuming $|R'_{a,x}| > 0$, $\forall x \in X'$.

How to distribute runs over inputs? In the procedure described above, the vast majority of time is spent executing $p.\text{eval}(x, a)$, which typically involves “running a on x ”. The latter can be time-consuming. In our sorting example, some algorithms took multiple minutes to sort certain sequences. Assume we are given a fixed budget of N calls to $p.\text{eval}(x, a)$, how should we distribute these calls over inputs as to maximize the accuracy of our estimate? For instance, assume $N = 100$, should we perform (A) 100 runs on a single input, (B) 10 runs on 10 inputs or (C) a single run on 100 inputs? If $p.\text{eval}$ is deterministic, the choice is obviously (C). However, when $p.\text{eval}$ is noisy, the best decision seems less straightforward. [Birattari and Kacprzyk 2009, p. 87] investigated this matter

in general and showed that the accuracy of \bar{o} , for any N , is maximized if we perform only a single run on each input sampled from \mathcal{D} . Put differently, (C) is preferable independent of how noisy $p.\text{eval}$ is. Let \mathcal{R}_a be the distribution of random variable $p.\text{eval}(\mathcal{D}.\text{sample}(), a)$. We can rewrite the “one run per input” variant of our estimator in Equation 4.1 as

$$\bar{o}(a) = \frac{1}{|R'_a|} \sum_{r \in R'_a} r \quad \text{where} \quad R'_a \sim \mathcal{R}_a \quad (4.2)$$

The latter is an ordinary sample average estimator for $o(a) = \mathbf{E}[\mathcal{R}_a]$. Under the assumption that $p.\text{eval}(\mathcal{D}.\text{sample}(), a)$ generates i.i.d. samples from \mathcal{R}_a , it follows from the central limit theorem that, given a sufficiently large sample R'_a , $\bar{o}(a)$ will be approximately normally distributed with mean $o(a)$ and standard deviation $\frac{\sigma(\mathcal{R}_a)}{\sqrt{|R'_a|}}$. The expected error on $\bar{o}(a)$, i.e.

$\mathbf{E}[|\bar{o}(a) - o(a)|]$, also known as the Standard Error (SE), is therefore given by $\frac{\sigma(\mathcal{R}_a)}{\sqrt{|R'_a|}}$. To the best of our knowledge, the use of sample averages, as described above, is the predominant way of estimating average-case algorithm performance. A procedure computing $\bar{o}(a)$ is shown in Algorithm 8. Remark that this procedure is stochastic. However, it can be made deterministic by computing R'_a at most once (\sim memoizing $o(a)$). The latter may be preferable when using it as an evaluation function e in a black box optimizer, which typically assumes (although it may not require) e to be deterministic.

Algorithm 8 Procedure computing the sample average estimator of algorithm performance, \bar{o} , given in Equation 4.2

```

1: procedure ESTIMATEO( $a, \mathcal{D}, p, N$ )
2:    $X' \leftarrow \emptyset$ 
3:   for  $j = 1 : N$  do                                     ▷ Collect sample of inputs.
4:      $x \leftarrow \mathcal{D}.\text{sample}()$ 
5:      $X' \leftarrow X' \cup \{x\}$ 
6:   end for
7:    $R'_a \leftarrow \emptyset$ 
8:   for all  $x \in X'$  do                                     ▷ Run algorithm once on each input.
9:      $r \leftarrow p.\text{eval}(x, a)$ 
10:     $R'_a \leftarrow R'_a \cup \{r\}$ 
11:  end for
12:   $\bar{o}_a \leftarrow 0$ 
13:  for all  $r \in R'_a$  do                                     ▷ Compute average observed performance.
14:     $\bar{o}_a \leftarrow \bar{o}_a + r$ 
15:  end for
16:  return  $\frac{\bar{o}_a}{N}$ 
17: end procedure

```

Brute Force Search: Having discussed how to estimate the performance of an algorithm instance in the set-ASP, we now turn towards the problem of determining which of a limited (finite) set of algorithms performs best based on estimates of their performance. Probably, the conceptually simplest, anytime, global optimizer solving this problem is given in Algorithm 9. Here, we enumerate the algorithm space systematically, evaluate the performance of each algorithm instance a using Algorithm 8. After each evaluation, we update the incumbent a_{best} to be the algorithm with the best estimate thus far.

Algorithm 9 A Brute Force (BF) search approach (black box optimizer variant)

```

1: procedure COMPAREBF( $\langle A, \mathcal{D}, p \rangle, N$ )
2:    $a_{\text{best}} \leftarrow \perp$  ▷  $a_{\text{best}}$  is the anytime solution
3:    $\bar{o}_{a_{\text{best}}} \leftarrow -\infty$ 
4:   for  $a \in A$  do
5:      $\bar{o}_a \leftarrow \text{ESTIMATEO}(a, \mathcal{D}, p, N)$  ▷ Evaluate performance using Algorithm 8
6:     if  $\bar{o}_a > \bar{o}_{a_{\text{best}}}$  then
7:        $a_{\text{best}} \leftarrow a$ 
8:     end if
9:   end for
10:  return  $a_{\text{best}}$ 
11: end procedure

```

Remark that $N = |R'_a|$ is a parameter of this procedure. The choice of N presents an inherent trade-off between its efficiency and the accuracy of its output:

efficiency: The total number of calls to $p.\text{eval}$ is $N|A|$ and as this is a rather costly operation, the time taken by Algorithm 9 grows roughly proportionally to N .

efficacy: Higher values of N result in more accurate estimates $\bar{o}(a)$ on average (smaller SE) and thereby increase the average accuracy of the result returned by Algorithm 9.

Figure 4.2 illustrates this trade-off empirically for our sorting set-ASP. It shows (blue o's) the average accuracy of the comparison and the time Algorithm 9 took, for different N values, based on data from 1000 independent runs. Accuracy is computed as the fraction of these 1000 runs returning the algorithm performing best on average in our use case (i.e. Tim sort, see Table 4.1). For higher values of N , Algorithm 9 clearly takes longer but its answers tend to be more frequently correct. Note that solving the sorting set-ASP with high accuracy is quite challenging as after $N = 1024$ runs of each algorithm and over 6 hours of optimization, we obtain an accuracy of only roughly 75%. While this trade-off between efficacy/efficiency is general and not specific to Algorithm 9, one might wonder whether it is possible to achieve a higher expected accuracy, using less resources?

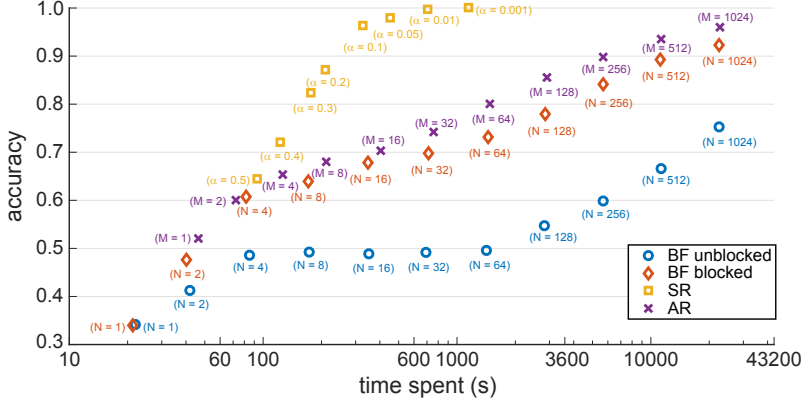


Figure 4.2: Illustration of the trade-off between accuracy and the resources required by different solvers with various parameter settings for the per-set Sorting ASP.

Blocking: One way to improve upon Algorithm 9 is through the use of a so-called blocked evaluation strategy [Addelman 1969]. In blocked evaluation, algorithms are compared based on performance observations obtained *on the same set of inputs* $X' \sim \mathcal{D}$. The idea is that “blocked” performance estimates of two different algorithms a_1 and a_2 are correlated, rather than independent. The variance of $\bar{o}(a_1) - \bar{o}(a_2)$ is in this case given by $\frac{\sigma(\mathcal{R}_{a_1})^2 + \sigma(\mathcal{R}_{a_2})^2 - 2\rho\sigma(\mathcal{R}_{a_1})\sigma(\mathcal{R}_{a_2})}{N}$, where ρ is the correlation between both estimates. Therefore, if both are positively correlated ($\rho > 0$), the variance on relative performance will be lower and the expected accuracy of the comparison higher, for the same N . Intuitively, if we compare estimates based on independent sets of inputs, we introduce additional variability due to the fact that some sets of inputs might be easier/harder to solve than others. A blocked variant of Algorithm 9 is obtained by computing X' at most once in Algorithm 8. Figure 4.2 illustrates the benefit of using a blocked strategy in our sorting set-ASP. It shows (orange \diamond 's) the average accuracy of the comparison and the time a blocked variant of Algorithm 9 took, for different N values, based on data from 1000 independent runs. We observe that, while taking the same time, the results returned by the blocked strategy are more reliable for a fixed N , e.g. for $N = 64$ Algorithm 9 obtained an accuracy of nearly 50%, while its blocked variant obtained an accuracy of over 70%. For $N = 1024$ it obtains an accuracy of well over 90%. In the AC community, the use of blocked evaluations is the de facto standard, implemented by virtually all state-of-the-art configurators. While not uncommon in the GP community either, alternatives have been studied, purposefully evaluating algorithm instances on different X' , as to diversify the search, e.g. dynamic subset selection [Gathercole and Ross 1994].

Incremental Comparison Schemes: While the GOP reduction allows us to solve the set-ASP using any of the available black box optimizers (e.g. those in Section 2.7), the black box evaluation procedure abstracts the intrinsically incremental nature of performance estimation. Remark that this evaluation procedure can be extremely costly, as it involves N calls to $p.\text{eval}$. As a consequence, large choices of N will negatively affect anytime performance. In our sorting example, for $N = 1024$, times ranged from a minute to over an hour depending on the algorithm evaluated. This means that if we are unlucky (e.g. bubble sort is evaluated first) we may need to wait over an hour before a_{best} is even instantiated. An incremental approach to performance evaluation resolves this issue, at the cost of requiring modifications to standard black box optimizers. Algorithm 10 is a functionally equivalent, incremental variant of Algorithm 9. Here, each iteration j , we run each algorithm on an input sampled from \mathcal{D} , and update their performance estimates and a_{best} accordingly. A blocked variant of Algorithm 10 is obtained by swapping lines 4 and 5. The resulting procedure was previously described in [Birattari and Kacprzyk 2009] (and called Brutus). It was also used in this form to compare configurations in the basic variant of ParamILS [Hutter et al. 2009] (BasicILS) and in GGA [Ansótegui et al. 2009]. The anytime performance of Algorithm 10 clearly no longer depends on our choice of N . Also, note that for $N = +\infty$, it is asymptotically correct, while Algorithm 9 would never get past evaluating the first candidate algorithm. Beyond superior anytime performance and asymptotic correctness, this incremental approach also gives us the flexibility to allocate runs to algorithms dynamically as we will discuss next.

Algorithm 10 A Brute Force (BF) search approach (unblocked, incremental variant)

```

1: procedure COMPAREBF( $\langle A, \mathcal{D}, p \rangle, N$ )
2:    $a_{\text{best}} \leftarrow \perp$ 
3:   for  $j = 1 : N$  do
4:     for all  $a \in A$  do                                ▷ Run each algorithm once and update estimates.
5:        $x \leftarrow \mathcal{D}.\text{sample}()$ 
6:        $r \leftarrow p.\text{eval}(x, a)$ 
7:        $\bar{o}_a \leftarrow \frac{(j-1)\bar{o}_a + r}{j}$ 
8:     end for
9:     for all  $a \in A$  do                                ▷ Update anytime solution.
10:      if  $\bar{o}_a > \bar{o}_{a_{\text{best}}}$  then
11:         $a_{\text{best}} \leftarrow a$ 
12:      end if
13:    end for
14:  end for
15:  return  $a_{\text{best}}$ 
16: end procedure

```

How to distribute runs over algorithms? Remark that Algorithm 10 distributes runs equally across the algorithm space. Some algorithms may perform poorly relative to others: While we might be able to tell that one algorithm instance will likely not perform best after only a few runs, doing the same for others may require significantly more data. It therefore makes sense to distribute runs non-uniformly over the algorithm space as to *allocate more runs to better algorithms*. Remark that doing so is particularly important in efficiency optimization settings, since runs of worse (= less efficient) algorithms actually take longer. Clearly, if relative performances were to be known, the set-ASP would effectively be solved a priori. However, using an incremental evaluation approach we can use past performance observations to decide upon further allocations dynamically. There are many different ways to do so, below we will describe two competing approaches used in the AC community. However, dynamic allocation of trials has also been explored in the GP community, e.g. in [Teller and Andre 1997]. Finally, this problem has also been studied in the RL community in the context of the *best-arm identification problem* in multi-armed bandits [Audibert and Bubeck 2010], a problem which is closely related to the set-ASP.

Statistical Racing (SR): The first approach is inspired by a model selection procedure used in machine learning known as *racing* [Maron and Moore 1994, Maron and Moore 1997]. The idea is that we evaluate every algorithm in the comparison only as long as is needed to determine that it is outperformed by some other, allowing us to focus computational effort on the remaining. The statistical racing framework (Algorithm 11) operates similarly to Algorithm 10. However, after each iteration, algorithm instances that performed significantly worse (with some given confidence level $1 - \alpha$) than the best in race (a_{best}) are excluded from the race until only a single algorithm remains. In SR this exclusion is based on statistical hypothesis tests. These tests compare two or more sets of data (performance observations of the algorithms compared) and determine the likelihood of generating data this (or more) extreme under a given null-hypothesis H_0 (the algorithms compared perform as well). If this likelihood, known as the p -value, is under a given threshold α , we say that we can reject H_0 (at least one algorithm performs significantly worse/better than another). One issue is that traditional tests, when comparing more than two algorithms, do not provide information about *which* algorithms perform significantly better/worse than others. An obvious solution would be to perform multiple pairwise hypothesis tests (e.g. Student t-test). However, the likelihood of incorrectly rejecting H_0 (excluding an algorithm which does not perform worse), known as a type I error, in the overall comparison would be much larger than α (a phenomenon known as *alpha inflation*). This issue is known as the multiple comparison problem [Hsu 1996] in statistics. Another issue is that the cost of, i.e. overhead introduced by, multiple pairwise tests can become significant. A common workaround uses a combination of two types of tests.

First, a single multi-group test (e.g. ANOVA) is used to determine whether any algorithm performs significantly worse/better than another. Only if this test succeeds (H_0 is rejected) we resort to pairwise testing of each algorithm to the current best in race a_{best} . [Birattari et al. 2002] was the first to apply SR to the algorithm configuration problem in F-Race, using the non-parametric Friedman test as the multi-group test, and its associated post hoc pairwise test. Later variants of F-Race were developed [Birattari et al. 2010] and used in the iRace [López-Ibáñez et al. 2011] framework to iteratively compare configurations sampled from a larger algorithm space. Figure 4.2 shows (yellow \square 's) the average accuracy of the comparison and the time a simple implementation of F-Race took on the sorting set-ASP, for different α values, based on data from 1000 independent runs. To reduce variance in our comparison, the same sets of inputs were used as in the blocked variant of Algorithm 9. F-Race clearly outperforms the brute force approach on this instance, in particular for $\alpha = 0.001$ it returned Tim sort each of the 1000 runs and this took on average only 20 minutes. Also, for $\alpha = 0.1$ it took on average only 5 minutes to obtain an accuracy greater than Algorithm 9 (blocked, with $N = 1024$) after 6 hours.

Algorithm 11 A Statistical Racing (SR) approach

```

1: procedure COMPARESR( $\langle A, \mathcal{D}, p \rangle, \alpha$ )
2:    $a_{\text{best}} \sim \mathcal{U}(A)$ 
3:   while  $|A| > 1$  do
4:      $x \leftarrow \mathcal{D}.\text{sample}()$  ▷ Blocked evaluation.
5:     for all  $a \in A$  do ▷ Run each algorithm once, and update performance data/estimates.
6:        $r \leftarrow p.\text{eval}(x, a)$ 
7:        $R_a \leftarrow R_a \cup \{r\}$ 
8:        $\bar{o}_a \leftarrow \frac{(j-1)\bar{o}_a + r}{j}$ 
9:     end for
10:    for all  $a \in A$  do ▷ Update anytime solution.
11:      if  $\bar{o}_a > \bar{o}_{a_{\text{best}}}$  then
12:         $a_{\text{best}} \leftarrow a$ 
13:      end if
14:    end for
15:    if reject  $H_0[o(a) = o(a'), \forall a, a' \in A]$  based on  $\{R_a \mid a \in A\}$  with confidence  $1 - \alpha$  then
16:      for  $a \in A \setminus \{a_{\text{best}}\}$  do ▷ Perform pairwise tests
17:        if reject  $H_0[o(a) = o(a_{\text{best}})]$  based on  $\{R_a, R_{a_{\text{best}}}\}$  with confidence  $1 - \alpha$  then
18:           $A \leftarrow A \setminus \{a\}$  ▷ Eliminates an algorithm performing significantly worse.
19:        end if
20:      end for
21:    end if
22:  end while
23:  return  $a_{\text{best}}$ 
24: end procedure

```

Aggressive Racing (AR) (Algorithm 12) offers an alternative to SR techniques. A first version of AR was presented in the focused variant of ParamILS [Hutter et al. 2009] (FocusedILS) to compare two configurations. Later, a variant of this procedure was presented in [Hutter et al. 2011] and used in configurators ROAR and SMAC to determine the best of a set of configurations. Unlike SR, which distributes computational effort equally amongst all contestants until they are excluded, AR will compare contestants through a series of pairwise races, each time retaining the winner of the previous race in the next one, and the winner of the final comparison is the overall winner. The version we present here, targeted at the small per-set ASP, will repeat this scheme M times, i.e. perform $M * (|A| - 1)$ such races, where M is a parameter of the framework. As the name suggests, AR will use a different, “more aggressive”, criterion to decide which contestant wins a race, i.e. it will not wait until sufficient evidence is gathered that one contestant is significantly worse than another. In a single race (a_{best} vs. a , lines 5-19), AR will first run a_{best} on a new input sampled from \mathcal{D} . Subsequently, AR will iteratively run a on an input x' randomly selected from the set of inputs X_{missing} on which a_{best} was already evaluated, but a not yet.³ The race continues as long as a is estimated to perform better than a_{best} , based *only* on performance observations made on inputs both algorithms have already been evaluated on ($\notin X_{\text{missing}}$). Only if the race continues until both have been evaluated on the same set of inputs ($X_{\text{missing}} = \emptyset$) and a is still estimated to perform better than a_{best} , a wins the race and becomes the new incumbent. Put differently, a will lose the race when (after a run) its estimated performance is worse than that of a_{best} based on performance observations on the same $|R_a|$ inputs. Note that this is often after only a single run. Figure 4.2 shows (purple \times 's) the average accuracy of the comparison and the time a simple implementation of AR took, for different M values, based on data from 1000 independent runs, using the same sets of inputs as used in earlier experiments. While AR performs better than brute force (BF), it does so far less impressively than SR, e.g. 96% accuracy for $M = 1024$ after an average of about 6 hours. However, in what follows, we argue that there are still features which might make AR preferable over SR in practice. Remark that the behavior of AR (beyond termination) is independent of M . As AR is anytime, there is no reason not to choose $M = +\infty$ and force-terminate it whenever we want, i.e. we do not need to fix/know our budget in advance. Also, AR with $M = +\infty$ is asymptotically correct (assuming A is finite). Note that SR with $\alpha = 0$ has similar properties, but is essentially equivalent to (blocked) BF. These features make AR also preferable in the semi-online setting we described in Section 3.5.3. Furthermore, AR will typically have better anytime performance, especially in settings where A is large and/or α is low, as SR will initially behave like (blocked) BF, i.e. distribute runs uniformly.

³This is similar to the version presented in [Hutter et al. 2011]. In [Hutter et al. 2009] inputs are considered in a fixed order.

Algorithm 12 An Aggressive Racing (AR) approach

```

1: procedure COMPAREAR( $\langle A, \mathcal{D}, p \rangle, M$ )
2:    $a_{\text{best}} \sim \mathcal{U}(A)$ 
3:   for  $i = 1 : M$  do
4:     for  $a \in A \setminus \{a_{\text{best}}\}$  do
5:        $x \leftarrow \mathcal{D}.\text{sample}()$ 
6:        $r \leftarrow p.\text{eval}(x, a_{\text{best}})$ 
7:        $R_{a_{\text{best}}} \leftarrow R_{a_{\text{best}}} \cup \{(x, r)\}$ 
8:        $X_{\text{missing}} \leftarrow \{x \mid \exists r : (x, r) \in R_{a_{\text{best}}}\} \setminus \{x \mid \exists r : (x, r) \in R_a\}$ 
9:       repeat  $\triangleright$  Race between  $a$  and  $a_{\text{best}}$ 
10:         $x' \sim \mathcal{U}(X_{\text{missing}})$ 
11:         $r' \leftarrow p.\text{eval}(x', a)$ 
12:         $R_a \leftarrow R_a \cup \{(x', r')\}$ 
13:         $X_{\text{missing}} \leftarrow X_{\text{missing}} \setminus \{x'\}$ 
14:         $\bar{o}_a \leftarrow \frac{1}{|R_a|} \sum_{(x,r) \in R_a} r$ 
15:         $\bar{o}_{a_{\text{best}}} \leftarrow \frac{1}{|R_{a_{\text{best}}}|} \sum_{(x,r) \in R_{a_{\text{best}}}} [x \notin X_{\text{missing}}] * r$ 
16:      until  $\bar{o}_a \leq \bar{o}_{a_{\text{best}}} \vee X_{\text{missing}} = \emptyset$ 
17:      if  $\bar{o}_a > \bar{o}_{a_{\text{best}}}$  then
18:         $a_{\text{best}} \leftarrow a$   $\triangleright a$  won the race
19:      end if
20:    end for
21:  end for
22:  return  $a_{\text{best}}$ 
23: end procedure

```

Scaling up to large algorithm spaces: All the comparison schemes described thus far enumerate A exhaustively (high-level search is systematic). Furthermore, all (except for AR) evaluate every algorithm instance once before evaluating any twice. This results in poor anytime performance if A is large or infinite, and is even impossible if A is uncountably infinite. The general approach to deal with this problem is to (Turing) reduce the problem of finding the best instance in A to solving multiple smaller set-ASPs, iteratively comparing small subsets of A , e.g. using one of the techniques described above. Now, the question that remains is: Which algorithm instances A_i to compare in each iteration i ?

One approach is to draw this sample uniformly at random, i.e. $A_i \sim \mathcal{U}(A)$. This approach was followed in ROAR, where each $\langle A_i \cup \{a_{\text{best}}\}, \mathcal{D}, p \rangle$ is solved using AR with $M = 1$.⁴ While seemingly naive, ROAR was shown to perform surprisingly well in practice, sometimes outperforming FocusedILS in configuration tasks [Hutter et al. 2011]. This implies that

⁴ROAR can be obtained by modifying line 4 of Algorithm 12, replacing A by $A_i \sim \mathcal{U}(A)$, i.e. racing a_{best} each iteration i against $A_i \sim \mathcal{U}(A)$, rather than A . Remark that the behavior of AR on multiple comparisons between i.i.d. sampled algorithm instances A_i is identical to that on $\bigcup_i A_i$, and as such the sample size ($|A_i|$) used in ROAR does not matter.

naturally occurring instances of the ACP are not always hard. Often a large fraction of the configurations perform well, rather than only a few, and as such one is likely to be sampled using a uniform sampling strategy. ROAR, despite being presented in [Hutter et al. 2011] as a configurator, does *only* require us to be able to draw samples uniformly from \mathcal{A} and is as such more widely applicable.

The main downside of this uniformed sampling strategy is that when good algorithm instances are rare, ROAR is unlikely to ever find one. In general, the strategy used to overcome this problem is to inform the sampling. More advanced schemes will attempt to bias the search dynamically, i.e. adapt the distribution from which is sampled based on the performance observed for the algorithms evaluated thus far, as to increase the likelihood of finding better algorithm instances. Doing so requires us to have a priori knowledge (see also Section 3.4) or make assumptions about the relations between algorithm instances which can be used to generalize performance observations made thus far to unseen algorithm instances. A common (implicit) assumption is that ϕ is smooth. Put differently, similar algorithm instances perform similarly, for some measure of similarity. The crux is to exploit this relation as to increase the likelihood of sampling algorithm instances similar to those that performed good, while decreasing the likelihood of sampling those similar to algorithms known to perform poorly.

Below, we will restrict our discussion to how this is done in four prominent, state-of-the-art configurators. As the similarity measures exploited are defined in configuration space, i.e. Θ , these techniques cannot be directly applied to the general per-set ASP. However, the underlying ideas can be generalized to a wide range of structured algorithm spaces.

ParamILS [Hutter et al. 2007b, Hutter et al. 2009] implements an Iterated Local Search (ILS, see Section 2.7.2.2) scheme in a one-exchange neighborhood. The one-exchange neighborhood of a configuration consists of all valid configurations that can be obtained by changing the value of a single parameter: $E(\theta) = \{\theta' \in \Theta \mid \sum_{i=1}^{|\theta|} [\theta_i \neq \theta'_i] = 1\}$. Initially, a few (default 10) configurations are sampled uniformly at random, the best of which is taken as a starting point of the search. Next, an Iterative Improvement (II) procedure is applied, each time replacing the current with the first improving neighbor encountered (first-improvement), until a local optimum is reached. Subsequently, the incumbent is perturbed (multiple parameter values (default 3) are re-assigned randomly) and the II procedure is repeated. At each point, the best configuration encountered thus far is retained as θ_{best} . [Hutter et al. 2009] describes two variants of ParamILS: BasicILS, which uses (blocked) BF with fixed N in pairwise comparisons, and FocusedILS, which uses a variant of AR with $M = 1$. Note that ParamILS, unlike the three configurators described below, cannot handle parameters with large (infinite) domains as identifying local optimality requires exhaustive enumeration of the one-exchange neighborhood.

iRace [Balaprakash et al. 2007, López-Ibáñez et al. 2011] follows a population-based search strategy. It randomly initializes a set of N_1 configurations.⁵ Each iteration i it races N_i configurations (using SR), until a minimum number of elite configurations Θ_{elite} remain. Subsequently, $N_{i+1} - |\Theta_{\text{elite}}|$ new configurations Θ_{new} are generated by sampling them from a mixture of distributions centered around Θ_{elite} . Finally, the new and the elite configurations ($\Theta_{\text{elite}} \cup \Theta_{\text{new}}$) are raced in the next iteration.

GGA [Ansótegui et al. 2009] implements a gender-based evolutionary search algorithm, maintaining a population of configurations (males and females with given age). GGA initializes this population randomly. Each iteration i (a.k.a. generation) racing (using BF with $N = i$) is used to select the best competitive individuals (males) from this population. These fittest males are randomly assigned females and a recombination/crossover (reproduction) is performed during which a new individual is created combining parameter values (genes) from both parents, possibly followed by a mutation which takes the form of a random one-exchange move for categorical parameters and Gaussian additive noise for numerical ones. Finally, old individuals die as to maintain a stable population size.

SMAC [Hutter et al. 2011] is an instance of the wider class of so-called Sequential Model-based Optimization (SMBO) methods.⁶ SMBO frameworks maintain an explicit regression model $M : \Theta \rightarrow \mathbb{R}$ (a.k.a. a response surface, surrogate or fitness landscape model) trained using all previous performance observations; e.g. SMAC uses a Random Forest (RF) model [Breiman 2001]. M gives us a performance prediction $M.o(\theta)$ for any configuration θ . Some models M (e.g. RF) also provide a prediction $M.unc(\theta)$ of the error on $M.o(\theta)$. SMBO frameworks iteratively select the configurations to be compared based on this model. SMAC uses the Expected Improvement (EI) criterion [Jones et al. 1998] for selecting configurations. Here, $M.o(\theta)$ and $M.unc(\theta)$ are used to estimate the expected improvement on θ_{best} , the best configuration encountered thus far, as follows:

$$EI(\theta) = (M.o(\theta) - M.o(\theta_{\text{best}})) * \Phi\left(\frac{M.o(\theta) - M.o(\theta_{\text{best}})}{M.unc(\theta)}\right) + M.unc(\theta) * \phi\left(\frac{M.o(\theta) - M.o(\theta_{\text{best}})}{M.unc(\theta)}\right).$$

where ϕ and Φ are the standard normal density and cumulative distribution functions. EI will be high for configurations estimated to perform well and for those with high estimated uncertainty; and as such this criterion offers an automatic balance between exploration and exploitation. To determine the configurations to be compared each iteration, SMAC performs multiple iterated local searches using the EI returned by the model as objective, i.e. obtaining a diverse set of configurations locally optimizing EI which are then raced against θ_{best} (using AR with $M = 1$).

⁵Here, N_1 depends on the number of parameters. Before each iteration, N_i is determined dynamically.

⁶Which also encompasses GGA++ [Ansótegui et al. 2015], a model-based extension of GGA.

4.2 Design by Per-input Algorithm Selection

4.2.1 The input- and subset-ASP

The second formulation we will discuss is the per-input ASP (input-ASP), in the spirit in which the ASP was originally formulated by John Rice in 1976. When considering a limited sample of algorithms A solving some target problem, we find that often no single algorithm instance performs best on all problem instances (see also Section 3.2). In such settings, it is beneficial to condition the choice of algorithm (design decisions) on features of the problem instance to be solved. In the input-ASP, we are to answer the following question: “Given a problem instance x , which algorithm $a \in A$ should be used to solve it?”. As discussed in Section 3.3.3, p. 88, this problem was first formulated as a computational problem in [Rice 1976], but it is only three decades later that it received widespread attention and that we saw the emergence of the “algorithm selection” community. Over the last decade, the possibility of explicitly conditioning design choices on features of the input has also been examined in various other communities; including algorithm scheduling [Lindauer et al. 2016], algorithm configuration [Xu et al. 2010, Kadioglu et al. 2010], and optimizing compilers such as Petabricks [Ansel et al. 2009, Ding et al. 2015].

John Rice (1976) originally formalized three variants of the problem. Here, we will first present the basic formulation and later focus on the “selection based on features” variant.

Definition 4.2: Per-input Algorithm Selection Problem (input-ASP)
[Rice 1976, pp. 3]

Instances of the input-ASP are defined by triples $\langle A, X, p \rangle$ where

A : a set of algorithms (*algorithm space*).

X : a set of possible inputs (*input space*).

p : a function $X \times A \rightarrow \mathbb{R}$ quantifying the performance of an algorithm on an input (*per-input performance measure*).

Candidate solutions are functions of the form $X \rightarrow A$ called *selection mappings*, specifying the algorithm to be used for any given input. In the input-ASP, given $\langle A, X, p \rangle$, we are to find a selection mapping s^* satisfying $p(x, a) \leq p(x, s^*(x)), \forall a \in A, \forall x \in X$.

Here, the most discriminative feature of the input-ASP, w.r.t. the set-ASP, is that its solution takes the form of a selection mapping $X \rightarrow A$, rather than a single algorithm instance $a \in A$. Remark that, instead of the input distribution \mathcal{D} , we are (only) given the input space X because the optimal selection mapping s^* does not depend on \mathcal{D} .

In the remainder of this section, we will consider an optimization variant of the “selection based on features” problem formulated by John Rice:

Definition 4.3: Per-subset Algorithm Selection Problem (subset-ASP)
[Rice 1976, pp. 9]

Instances of the subset-ASP are defined by quadruples $\langle A, \mathcal{D}, \phi, p \rangle$ where

A : a set of algorithms (*algorithm space*).

\mathcal{D} : a distribution over a set of inputs X (*input distribution*).

ϕ : a function $X \rightarrow \mathbb{R}^m$ mapping an input to a feature vector. (*feature mapping*).

p : a function $X \times A \rightarrow \mathbb{R}$ quantifying the performance of an algorithm on an input (*per-input performance measure*).

Candidate solutions S are functions of the form $\mathbb{R}^m \rightarrow A$ mapping feature to algorithm space. In the subset-ASP, given $\langle A, \mathcal{D}, \phi, p \rangle$, we are to find a selection mapping s^* satisfying $o(s) \leq o(s^*)$, $\forall s \in S$, where $o(s) = \mathbf{E}_{x \sim \mathcal{D}} p(x, s(\phi(x)))$.

Here, we will assume A , \mathcal{D} and p to be passed to the solver in the same form as in the set-ASP (see Section 4.1.1). We do not assume ϕ to be given explicitly. Rather, we will assume we are given a procedure $\phi.\text{extract}(x)$ which can be used to extract a vector of real-valued features for any given input x . The procedure itself will be treated as a black box. Let us define the feature space Ω as the range of ϕ , i.e. $\Omega = \{f \in \mathbb{R}^m \mid \exists x \in X : \phi(x) = f\}$. Remark that ϕ does not have to be injective, i.e. multiple x might map to the same features f . As such ϕ can be seen as partitioning X into $|\Omega|$ subsets $X_f = \{x \in X \mid \phi(x) = f\}$ and restricting possible selection mappings to those satisfying $\phi(x) = \phi(x') \implies s(x) = s(x')$, $\forall x, x' \in X$, i.e. selecting an algorithm for each of these subsets. We therefore also refer to this problem as the per-subset Algorithm Selection Problem (subset-ASP). Remark that both the input-ASP (see Definition 4.2) and the set-ASP (see Definition 4.1) can be seen as special cases of the subset-ASP, with ϕ injective and ϕ constant, respectively.

4.2.2 Formulating the ADP as a subset-ASP

Before discussing how to solve the subset-ASP, we will briefly discuss the subset-ASP reduction depicted in Figure 4.3. We already discussed the reduction of the ADP to the set-ASP (see Section 4.1.1), and, in what follows, we will focus on how the reduction to the subset-ASP differs. Recall that the many-one reduction of a computational problem (see Definition 2.12) consist of a pair of functions, a *formulation* and an *interpretation*.

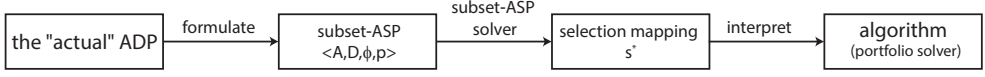


Figure 4.3: Diagram illustrating the design by subset-ASP reduction approach.

Interpretation: Unlike in the set-ASP, the solution to the subset-ASP is no algorithm for our target problem, but rather a selection mapping. This selection mapping s is interpreted as an algorithm which, given as input a problem instance x :

1. Extracts features from x : computes $\vec{f} = \phi(x)$.
2. Selects an algorithm a based on these features: computes $a = s(\vec{f})$.
3. Solves x using a : computes $y = a(x)$.
4. Returns y .

Realizations of the framework described above are also known as *portfolio solvers*.

Formulation - sorting subset-ASP: Next, we discuss the formulation of the ADP as a subset-ASP, i.e. the choice of $\langle A, \mathcal{D}, \phi, p \rangle$. We already discussed the choices for A , \mathcal{D} and p in the context of the set-ASP and for our sorting subset-ASP we will make the same decisions we made in Section 4.1.1. In general, however, there is a noteworthy difference in the criteria on which one should base one's choice of A . To illustrate this, assume a binary feature discriminating between two classes of inputs X_0 and X_1 , equally likely under \mathcal{D} . We are to decide which two out of four algorithms $\{a_1, a_2, a_3, a_4\}$ to include in A . The average-case performance of every algorithm on each of the classes is given below:

	a_1	a_2	a_3	a_4
X_0	0.9	0.8	0.7	0.1
X_1	0.1	0.7	0.8	0.9

Observe that a_2 and a_3 ($o = 0.75$) perform better on average than a_1 and a_4 ($o = 0.5$) across all inputs, while a_1 and a_4 perform best on X_0 and X_1 respectively. In the set-ASP, one eliminates those algorithm instances which one believes to have inferior average-case performance than some $a \in A$, i.e. one would choose $A = \{a_2, a_3\}$. If we make the same choice for A in the subset-ASP, the optimal selection mapping would be $s^* = \{(0, a_2), (1, a_3)\}$ with $o(s^*) = 0.8$. On the other hand, if we would have chosen $\{a_1, a_4\}$ instead, the optimal selection mapping would have been $s^* = \{(0, a_1), (1, a_4)\}$ with $o(s^*) = 0.9$. The crux is that the poor performance of a_1 and a_4 on X_1 and X_0

respectively, is irrelevant in the subset-ASP as these algorithms will not be selected on these inputs. Therefore, rather than including algorithms which (individually) perform well across all instances, we should choose A to be a set of complementary algorithms, each excelling on a particular class of inputs. Formally, we retain optimality only if

$$\forall \vec{f} \in \mathbb{R}^m, \exists a \in A : \sum_{x \in X_{\vec{f}}} D(x)p(x, a') \leq \sum_{x \in X_{\vec{f}}} D(x)p(x, a), \quad \forall a' \in A_U.$$

Conceptually, in the subset-ASP, one can only eliminate algorithms which one believes to have inferior average-case performance to some $a \in A$ on *all* $|\Omega|$ discriminable subsets. For instance, in our sorting example, while we could reasonably exclude $\mathcal{O}(n^2)$ algorithms (e.g. bubble sort, selection sort and insertion sort) from A in any sorting set-ASP, where one is likely to be asked to sort long sequences, in subset-ASPs, where one can discriminate short from long input sequences, it makes sense to include them as they may outperform algorithms with superior asymptotic complexity (e.g. merge sort) on shorter sequences.

Finally, we discuss the choice of ϕ . There are many different choices one could make for ϕ .extract in this reduction. In general, the following should be taken into consideration:

Discriminativeness: It should be possible to discriminate between inputs best solved using different algorithms based on the features it computes. Formally, let

$$X_a = \{x \in X \mid p(x, a') \leq p(x, a), \forall a' \in A\}$$

be the set of instances for which algorithm a is non-dominated, then ϕ should satisfy $\phi(x) = \phi(x') \implies \exists a \in A : x, x' \in X_a$.

Complexity reduction: Clearly, ϕ being injective is a sufficient condition for being discriminative, but it is not a necessary one. In formulating the ADP as a subset-ASP, in the context of semi-automated design, we would like to make it as easy as possible for the computer to solve. Here, we generally prefer ϕ 's which map X to a low dimensional (small), structured feature space, simplifying the selection process. In particular, inputs with similar features should have affinity with the same algorithm, while inputs with different features should ideally have affinity with different algorithms. Remark that the solvers we will describe in Section 4.2.3 make specific assumptions about the form these similarities in feature space (can) take, making the best choice of features subset-ASP solver dependent.

Efficiency: Note that the portfolio solver computes ϕ and s for every input x . This computation, while not captured by the subset-ASP, introduces overhead in practice which affects the efficiency of the resulting design. While often many features can be computed efficiently from x , computing some features may be expensive, even to the extent that the portfolio solver obtained becomes useless in practice.

Remark that we only touched upon the surface of this matter. The problem of extracting features from natural representations is an active research area [Liu and Motoda 1998].

For our sorting example, we chose the following features of the input sequence l

f_{length} : The number of elements in the sequence.

f_{range} : The difference between the largest and the smallest integer in the sequence.

f_{sorted} : The fraction of consecutive elements l_{i-1} and l_i satisfying $l_{i-1} < l_i$.

f_{equal} : The fraction of consecutive elements l_{i-1} and l_i satisfying $l_{i-1} = l_i$.

The procedure $\phi.\text{extract}$ used for computing them is shown in Algorithm 13.

Algorithm 13 Procedure used to compute ϕ in our sorting subset-ASP.

```

1: procedure EXTRACTFEATURES( $l$ )
2:    $n_{\text{sorted}}, n_{\text{equal}} \leftarrow 0$ 
3:    $l_{\min}, l_{\max} \leftarrow l_1$ 
4:   for  $i \in 2 : |l|$  do
5:     if  $l_{i-1} < l_i$  then
6:        $n_{\text{sorted}} \leftarrow n_{\text{sorted}} + 1$ 
7:       if  $l_{\max} < l_i$  then
8:          $l_{\max} \leftarrow l_i$ 
9:       end if
10:    else if  $l_{i-1} = l_i$  then
11:       $n_{\text{equal}} \leftarrow n_{\text{equal}} + 1$ 
12:    else if  $l_i < l_{\min}$  then
13:       $l_{\min} \leftarrow l_i$ 
14:    end if
15:  end for
16:   $f_{\text{length}} \leftarrow |l|$ 
17:   $f_{\text{range}} \leftarrow l_{\max} - l_{\min}$ 
18:   $f_{\text{sorted}} \leftarrow \frac{n_{\text{sorted}}}{|l|-1}$ 
19:   $f_{\text{equal}} \leftarrow \frac{n_{\text{equal}}}{|l|-1}$ 
20:  return  $\langle f_{\text{length}}, f_{\text{range}}, f_{\text{sorted}}, f_{\text{equal}} \rangle$ 
21: end procedure

```

4.2.3 Solving the subset-ASP

Now, we turn towards solving the subset-ASP. Solvers for the subset-ASP are also known as portfolio builders. Arguably, one of the reasons it took nearly three decades for “algorithm selection” to gain wide attention is because, while John Rice, in 1976, had formulated the problem and proposed a theoretical framework (approximation theory) to study it, he did not present, nor suggest any algorithmic solutions. In what follows, we give a rough overview of different solution approaches.

Finite feature spaces: Remark that ϕ partitions X into $|\Omega|$ subsets. If the feature-space Ω is finite, we can (Turing) reduce the subset-ASP $\langle A, \mathcal{D}, \phi, p \rangle$ to the set-ASP. Here, we solve a finite number of set-ASPs $\langle A, \mathcal{D}_{\bar{f}}, p \rangle$, one for each partition $X_{\bar{f}}$, where $\mathcal{D}_{\bar{f}}(x) = \frac{[\phi(x)=\bar{f}] * D(x)}{\sum_{x' \in X_{\bar{f}}} D(x')}$ is the distribution of inputs $x \in X_{\bar{f}}$. The resulting selection mapping would then simply map x to the solution of set-ASP $\langle A, \mathcal{D}_{\phi(x)}, p \rangle$ which can be obtained using any of the techniques described in Section 4.1.2.

Generalization in feature space using ML: Often, the feature space is too large to make the approach described above tractable. In these cases, we will need to generalize performance observations across inputs with different features. Most commonly traditional ML techniques are used to do so.

Clustering: One approach uses unsupervised learning (clustering). Here, one first generates a sample of inputs $X' \sim \mathcal{D}$ which is subsequently partitioned into k clusters based on similarities in feature space. Subsequently, the best algorithm for each cluster is determined [Malitsky and Sellmann 2010]. Given a test input, we compute its features and solve it using the best algorithm for the nearest cluster. The crux here is that a single algorithm instance is assumed to dominate all others on sufficiently similar inputs. Note that this can be seen as using clustering to (many-one) reduce the subset-ASP $\langle A, \mathcal{D}, \phi, p \rangle$ with a large feature space Ω to a subset-ASP $\langle A, \mathcal{D}, \phi', p \rangle$ with a small (finite) number of partitions, where ϕ' maps each input to the prototype for its nearest cluster in Ω .

As an illustration, we applied this technique to our sorting subset-ASP. The framework used is shown in Algorithm 14 and is an anytime variant of the procedure described above, iteratively constructing X' and updating the incumbent selection mapping s_{best} every time X' doubles in size.⁷ For clustering we used the G-means clustering algorithm [Hamerly et al. 2003] with a maximum of 64 clusters.⁸ The best algorithm for each cluster k (a_k) is determined based on a single evaluation of each algorithm on each input $x \in X_k$ assigned to the cluster (\sim BF with $N = |X_k|$). Figure 4.4 shows (light blue +’s) the average performance of s_{best} , for different sizes of X' , based on data from 1000 independent runs of Algorithm 14. As a reference, we also show the performance of the Virtual Best Solver

⁷We do not retrain our ML models every iteration in Algorithms 14–16, but use an exponential back-off mechanism instead, to reduce computational overhead. If the ML model used can be trained online (i.e. on a stream of data) such back-off mechanism would be unnecessary.

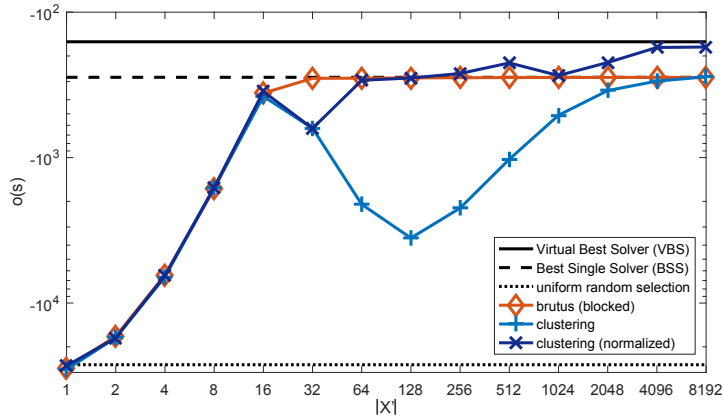
⁸G-means is an extension of the well-known K-means algorithm. While K-means requires the user to specify the # clusters K a priori, G-means is parameterless, deriving K from the data using statistical normality tests, i.e. splitting a cluster if it can reject the H_0 that the data within the cluster is generated by a multivariate Gaussian distribution. We used the implementation of G-means provided in the Java SMILE machine learning library: <http://haifengl.github.io/smile/> (version 1.3.1).

Algorithm 14 Framework for solving the subset-ASP using clustering.

```

1: procedure SELECTCLUSTER( $\langle A, \mathcal{D}, \phi, p \rangle, N, \text{cluster}$ )
2:    $b \leftarrow 1$ 
3:   for  $i \in 1 : N$  do
4:      $x \leftarrow \mathcal{D}.\text{sample}()$  ▷ Add new input  $x$  to training input set.
5:      $X' \leftarrow X' \cup \{x\}$ 
6:     for  $a \in A$  do ▷ Test each algorithm once on  $x$  and update performance data.
7:        $r \leftarrow p.\text{eval}(x, a)$ 
8:        $R_a \leftarrow R_a \cup \{(x, r)\}$ 
9:     end for
10:    if  $i = b$  then ▷ If  $i$  is a power of two
11:       $b \leftarrow 2 * b$ 
12:       $\langle M, \{X_1, \dots, X_K\} \rangle \leftarrow \text{cluster}(X', \phi)$  ▷ Partition training inputs in  $K$  clusters.
13:      for  $k \in 1 : K$  do ▷ Determine the best algorithm for each cluster.
14:         $a_k = \arg \max_{a \in A} \sum_{(x,r) \in R_a} [x \in X_k] * r$ 
15:      end for
16:       $s_{\text{best}} \leftarrow \lambda(x) = a_k, \text{ where } k = M.\text{predict}(x)$  ▷ Update anytime solution.
17:    end if
18:  end for
19:  return  $s_{\text{best}}$ 
20: end procedure

```

Figure 4.4: Performance of s_{best} obtained by Algorithm 14 for different $|X'|$.

(VBS, an oracle selecting the best algorithm for each input), the Best Single Solver (BSS, i.e. Tim sort), uniform random selection, as well as the average performance of a_{best} obtained by a (blocked) BF set-ASP solver which we will refer to as Brutus.

We observe that up and till $|X'| = 16$ our clustering approach performs similarly to Brutus. This is unsurprising as for $|X'| \leq 16$ the G-means algorithm assigns all inputs to a single cluster and the framework reduces to Brutus. For $|X'| \geq 32$ the G-means algorithm assigns inputs to two or more different clusters. We observe that, on average, this results in a drop in performance. This could be explained by the fact that training set X' is now divided into multiple smaller training sets X'_k , which may not be representative for \mathcal{D}_k , the distribution of inputs assigned to cluster k , and a_k inaccurate as a consequence. As of $|X'| \geq 256$ performance improves again. At this point, G-means uses all 64 clusters and the average size of partitions X'_k increases. Overall, the results are disappointing, not only did Algorithm 14 take over $100\times$ longer than Brutus to obtain a mapping with performance similar to the BSS, an increase in resources actually led to worse designs.

One factor hindering the clustering is the differences in scales between our features: while $f_{\text{sorted}}, f_{\text{equal}} \in [0, 1]$, we have $f_{\text{length}} \in [2, 2 * 10^5]$ and $f_{\text{range}} \in [0, 2 * 10^9]$. The K/G-means algorithm used is particularly sensitive to such scale differences as it measures similarity based on Euclidean distance in feature space. As a consequence, in our sorting subset-ASP, it will cluster sequences mainly based on similarities in f_{range} , almost completely disregarding differences in f_{sorted} and f_{equal} . To overcome this issue, we transformed each feature f_i as $f'_i = \frac{\log(f_i - f_i^{\min} + 1)}{\log(f_i^{\max} - f_i^{\min} + 1)}$, where f_i^{\min} and f_i^{\max} are the smallest and largest values feature f_i takes for any input. This has the effect of normalizing all features in range $[0, 1]$ and applying a log-scale. The latter encodes our belief that large differences in length/range are less relevant if length/range are large as well. The dark blue \times 's in Figure 4.4 show the performance of Algorithm 14 using the transformed features. While we still observe a drop in performance for $|X'| = 32$ it is considerably smaller and we eventually (after a day of CPU time) achieve near VBS performance. More specifically, s_{best} closes on average 90% of the gap between the VBS and BSS.⁹

Regression: Another approach involves building $|A|$ regression models $M_a : \mathbb{R}^m \rightarrow \mathbb{R}$, one for each $a \in A$, predicting the average-case performance of a on $X_{\bar{f}}$. Each M_a is trained using observed $(\phi.\text{extract}(x), p.\text{eval}(x, a))$ pairs, obtained by evaluating a on numerous inputs $x \sim \mathcal{D}$. The selection mapping is $s(x) = \arg\max_{a \in A} M_a(\phi(x))$, i.e. selects the algorithm with the highest predicted performance. This approach was first proposed to solve the subset-ASP in [Leyton-Brown et al. 2003], motivated by prior work on performance prediction in the context of empirical hardness models [Leyton-Brown et al.

⁹The “gap closed” performance measure is calculated as $\text{gap_closed}(s) = \frac{o(\text{BSS}) - o(s)}{o(\text{BSS}) - o(\text{VBS})}$.

2002, Nudelman et al. 2004, Leyton-Brown et al. 2009]. Most notably, this technique was used in SATzilla [Xu et al. 2008] (employing ridge regression).

As an illustration, we applied this technique to our sorting subset-ASP. The framework used is shown in Algorithm 15 and is an anytime variant of the procedure described above, iteratively constructing a set of performance observations for each algorithm (R_a), refitting the regression models and updating s_{best} every time X' doubles in size ($|X'| = |R_a|$). As regression strategy, we used the implementation of Random Forests [Breiman 2001] (RFs) provided in the Java SMILE machine learning library.¹⁰ This choice was motivated by the relative robustness of RFs (w.r.t. the choice of hyper-parameters, the scale of features etc.). Figure 4.5 shows (dark green ∇ 's) the average performance of s_{best} , for different sizes of X' , based on data from 1000 independent runs of Algorithm 15. We observe performance similar to Brutus for $|X'| \leq 32$, after which Brutus converges to the BSS, while the selection mapping obtained by our regression approach continues to improve further, eventually closing the gap with roughly 40% for $|X'| \geq 256$.

Algorithm 15 Framework for solving the subset-ASP using regression.

```

1: procedure SELECTREGRESS( $\langle A, \mathcal{D}, \phi, p \rangle, N, \text{regress}$ )
2:    $b \leftarrow 1$ 
3:   for  $i \in 1 : N$  do
4:      $x \leftarrow \mathcal{D}.\text{sample}()$  ▷ Add new input  $x$  to training input set.
5:      $X' \leftarrow X' \cup \{x\}$ 
6:      $\vec{f} \leftarrow \phi.\text{extract}(x)$ 
7:     for  $a \in A$  do ▷ Test algorithms on  $x$  and update performance data.
8:        $r \leftarrow p.\text{eval}(x, a)$ 
9:        $R_a \leftarrow R_a \cup \{(\vec{f}, r)\}$ 
10:    end for
11:    if  $i = b$  then ▷ If  $i$  is a power of two
12:       $b \leftarrow 2 * b$ 
13:      for  $a \in A$  do ▷ Build  $|A|$  performance prediction models.
14:         $M_a \leftarrow \text{regress}(R_a)$ 
15:      end for
16:       $s_{\text{best}} \leftarrow \lambda(x) = \arg \max_{a \in A} M_a.\text{predict}(\phi.\text{extract}(x))$  ▷ Update anytime
17:    end if
18:  end for
19:  return  $s_{\text{best}}$ 
20: end procedure

```

¹⁰A RF regression model consists of an ensemble of k regression trees. Here, we chose $k = 64$, which the empirical study conducted in [Oshiro et al. 2012] suggested to be a reasonable default.

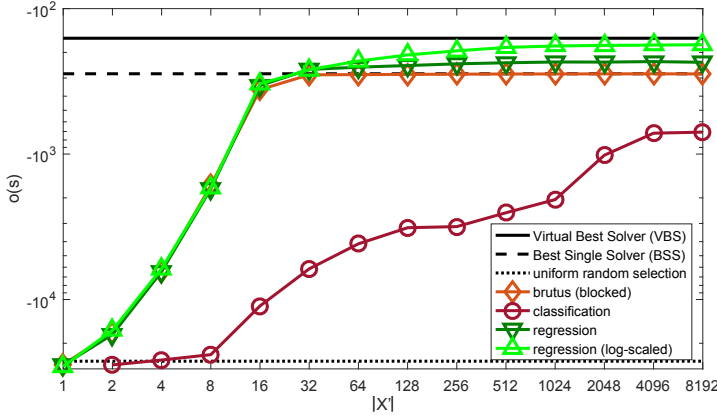


Figure 4.5: Performance of s_{best} obtained by Algorithms 15 and 16 for different $|X'|$.

In scenarios where performance observations vary strongly in magnitude, it is not uncommon to learn a predictive model $M_a(x) \approx \text{sgn}(p(x, a)) \log(|p(x, a)|)$ instead.¹¹ For instance, this log-transformation was applied in SATzilla [Xu et al. 2008] and in SMAC [Hutter et al. 2011] for runtime predictions. We therefore also ran Algorithm 15 on a subset-ASP $\langle A, \mathcal{D}, \phi, p' \rangle$, where $p'(x, a) = -\log(|p(x, a)|)$. The bright green \triangle 's in Figure 4.5 shows the average performance of the s_{best} 's obtained. We observe that this trick further improved performance, eventually closing the gap on average with 85%, for $|X'| \geq 1024$. Compared to the clustering approach, the regression approach performed better on smaller datasets.

Classification: A final approach reduces the subset-ASP to a multi-class Classification Problem (k -CP). In the k -CP, we are to find a classifier assigning “objects” (e.g. animals) to one of k “classes” (e.g. species). In the set-ASP reduction, possible inputs X correspond to the objects and A to the classes. A classifier assigns objects to classes, and as such can be interpreted as a selection mapping. In supervised learning, classifiers are trained based on labeled training data, i.e. pairs $(x, a) \in X \times A$ indicating that x is best solved using a . In the ASP, such information is not available a priori, however, (possibly noisy) training data can be collected by (1) generating a sample of training inputs X' using $\mathcal{D}.\text{sample}$ and (2) determining for each $x \in X'$ which algorithm performs best.

¹¹While this log-transformation does not affect the optimal selection mapping for the input-ASP, as $\arg \max_{a \in A} p(x, a) = \arg \max_{a \in A} \text{sgn}(p(x, a)) * \log(|p(x, a)|)$, the same does not hold for the subset-ASP, in general, as $\arg \max_{a \in A} \mathbf{E}_{x \sim D} p(x, a) \neq \arg \max_{a \in A} \mathbf{E}_{x \sim D} \text{sgn}(p(x, a)) * \log(|p(x, a)|)$.

As an illustration, we applied this technique to our sorting subset-ASP. The framework used is shown in Algorithm 16 and is an anytime variant of the procedure described above, iteratively constructing a set of labeled training data D' , retraining a classifier and updating s_{best} every time X' doubles in size ($|X'| = |D'|$). Remark that labels are determined based on a single evaluation of each algorithm (\sim BF (blocked) with $N = 1$). We again used a Random Forest prediction model, this time for classification (i.e. an ensemble of 64 decision trees). Figures 4.5 and 4.6 show (red o's) the average performance and average accuracy, respectively, of s_{best} 's obtained by Algorithm 16, for different sizes of D'/X' , based on data from 1000 independent runs. We observe that while the s_{best} 's obtained using the classification approach have an average accuracy of 70%, which is higher than that obtained by any of the approaches described thus far, their average performance is much worse than the BSS for $|X'| \leq 8192$. This implies that while s_{best} selects the optimal algorithm 70% of the time, the remaining 30% it selects an algorithm which is sometimes much worse. This is a product of the fact that ordinary classifiers attempt to minimize classification error, disregarding the possibly varying cost associated with misclassification. Research on example-dependent cost-sensitive classification tries to alleviate this problem [Brefeld et al. 2003, Bahnsen et al. 2015]. The clustering and regression techniques described earlier can be viewed as a form of cost-sensitive classification. Indeed, we observe that the best clustering and regression approaches obtain near VBS performance with an accuracy of only 50%, indicating that even when a suboptimal algorithm is selected, the selected algorithm does not perform much worse.

Algorithm 16 Framework for solving the subset-ASP using classification.

```

1: procedure SELECTCLASSIFY( $\langle A, \mathcal{D}, \phi, p \rangle, N, \text{classify}$ )
2:    $b \leftarrow 1$ 
3:   for  $i \in 1 : N$  do
4:      $x \leftarrow \mathcal{D}.\text{sample}()$  ▷ Add new input  $x$  to training input set.
5:      $X' \leftarrow X' \cup \{x\}$ 
6:      $a_x \leftarrow \arg \max_{a \in A} p.\text{eval}(x, a)$  ▷ Determine which algorithm works best on  $x$ .
7:      $D' \leftarrow (x, a_x)$ 
8:     if  $i = b$  then ▷ If  $i$  is a power of two
9:        $b \leftarrow 2 * b$ 
10:       $M \leftarrow \text{classify}(D', \phi)$  ▷ Train a classifier.
11:       $s_{\text{best}} \leftarrow \lambda(x) = M.\text{predict}(x)$ . ▷ Update anytime solution.
12:    end if
13:  end for
14:  return  $s_{\text{best}}$ 
15: end procedure

```

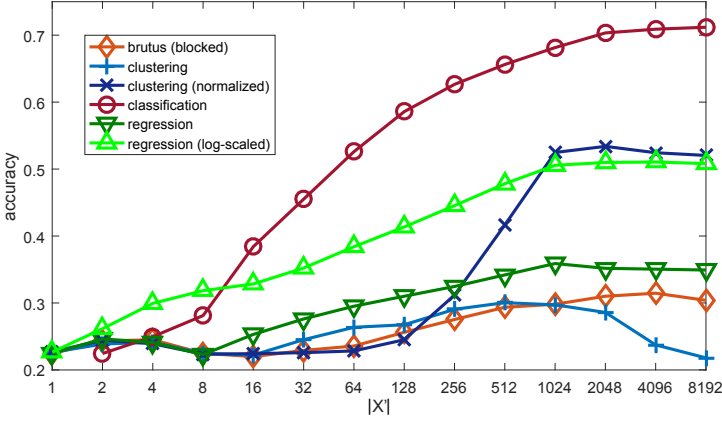


Figure 4.6: Accuracy of s_{best} obtained by Algorithms 14, 15 and 16 for different $|X'|$.

Scaling up to large algorithm spaces: As discussed in Section 3.3.3 (p. 88), the AS community traditionally focuses on instances of the subset-ASP where A is relatively small. However, instances of the subset-ASP with large (infinite) A have been studied. For instance, in the AC community, where some have considered a per-subset variant of the ACP, i.e. the subset-ACP. In the subset-ACP, we are to select a configuration based on input features, i.e. learn a “configuration selection portfolio”.

Parameters as algorithm features: One approach [Hutter and Hamadi 2005, Hutter et al. 2006] extends the regression technique described in the previous section to deal with large or even infinite A . Rather than maintaining a model M_a for each algorithm instance, which does not scale, it learns a single model $M : \mathbb{R}^m \times \Theta \rightarrow \mathbb{R}$ including parameter values as features of the algorithm instances and effectively generalizing performance observations also across algorithm instances. However, there are a few downsides associated with this approach. First, we would like, for each x , to select the configuration which maximizes the performance predicted by our model, i.e. $s(x) = \arg \max_{\theta \in \Theta} M(\phi(x), \theta)$. However computing $s(x)$ involves solving a non-trivial GOP $\langle M(x, \cdot), \Theta \rangle$. Furthermore, as noted by [Kadioglu et al. 2010], it might in practice select configurations which were never used during training, making it sensitive to model errors.

In what follows, we discuss two prominent frameworks, which to our knowledge represent the most successful approaches to the subset-ACP thus far.¹²

¹²Remark that the underlying frameworks do not critically rely on algorithms being represented as configurations and are therefore applicable to any subset-ASP (with structured A).

Hydra [Xu et al. 2010] is a framework decomposing the subset-ACP into multiple small subset-ASP and large set-ACP instances. Hydra takes a portfolio builder and configurator as arguments, which are themselves treated as black boxes. First, Hydra will solve the set-ACP, obtaining $s_{\text{best}}(x) = \theta^*$. Next, iteratively, a configurator is ran to find a configuration maximally complementary to s_{best} . To this end, the objective of configuration is changed from optimizing $o(\theta)$ to optimizing $o_{s_{\text{best}}}(\theta) = \mathbf{E}_{x \sim \mathcal{D}} \max(p(x, \theta), p(x, s_{\text{best}}(x)))$. $o_{s_{\text{best}}}(\theta)$ is an optimistic estimate of the performance of including θ in the portfolio of s_{best} . Subsequently, a subset-ASP is solved where A consists of all configurations currently in the range of s_{best} and θ . The resulting selection mapping is the new s_{best} .

ISAC [Kadioglu et al. 2010]. While the classification and regression approaches described above do not scale to large algorithm spaces, the clustering approach does, as it reduces the subset-ASP, with large A , to multiple set-ASP, with large A , which we know how to solve, assuming A is structured (see Section 4.1.2, p. 114). ISAC applies a clustering subset-ASP approach, called Stochastic Offline Programming [Malitsky and Sellmann 2010], to the subset-ACP. First, ISAC clusters inputs according to “similarity” of normalized features using the G -means algorithm [Hamerly et al. 2003], subsequently it determines the best configuration for each cluster using the GGA configurator [Ansótegui et al. 2009]. Remark that an “input sensitive” extension of the Petabricks compiler [Ding et al. 2015] implements a similar scheme.

4.3 Design by Dynamic Algorithm Selection

In the subset-ASP, we are to determine which of a given set of algorithm instances to use (\sim combination of design decisions to make), conditioned on features of the problem instance to be solved. Remark that in the resulting design (portfolio solver), all decisions w.r.t. how to best solve a given problem instance are made prior to actually solving it, i.e. statically. Therefore, we will also refer to the subset-ASP as the “static ASP”. Remark that the latter includes the input/set-ASP as special cases. Please observe that:

- It is often difficult to accurately and efficiently predict which algorithm instance will perform best on a given input. For instance, sufficiently discriminative features might not be available, or they might be too expensive to compute.
- Not all decisions must actually be made prior to execution. For instance, take algorithm configuration: While algorithm frameworks are traditionally viewed as taking parameter values *as input*, their execution does not actually depend on them until they are actually used. Sometimes, some parameters may not be used at all, in which case there was no need to assign them a value in the first place.

- Deferring choices until runtime allows us to make more informed decisions, i.e. dependent not only on features of the input, but also on information gathered during the execution thus far; e.g. stochastic events that occurred.
- It may be possible and beneficial to “vary” our decisions over the course of the run (e.g. lowering the temperature parameter in the simulated annealing framework).

These observations have motivated “dynamic” approaches to algorithm selection/design, considering “how to best make multiple choices we face over the course of an execution”.

4.3.1 The Dynamic ASP

While problems of this form, which we will refer to as “dynamic ASPs”, have received wide-spread attention in the last few decades (e.g. dynamic algorithm portfolios [Kautz et al. 2002, Carchrae and Beck 2005, Gagliolo and Schmidhuber 2006], recursive algorithm selection [Lagoudakis and Littman 2000], parameter control [Eiben et al. 2007, Karafotias et al. 2014], operator selection [DaCosta et al. 2008, Battiti et al. 2008], selection hyper-heuristics (see Section 2.7.2.5), just-in-time compilation [Krall 1998]), and are viewed as a natural generalization of the “static ASP” as defined by John Rice (1976), we have failed to find any general definition or formalization of this notion as a computational problem, a matter which we attempted to address in [Adriaensen and Nowé 2016b]. In what follows, we will first discuss prior art, followed by our own formal definition of the dynamic ASP.

Related research: Attempts to formalize the dynamic ASP have been made in the context of solving specific dynamic selection problems, most notably

[Lagoudakis and Littman 2000] consider the design of divide & conquer algorithms. These algorithms solve a problem by reducing it to multiple smaller instances of the same problem. Recursive sorting algorithms such as mergesort and quicksort are prototypical examples. While traditional recursive algorithms will use the same (recursive) algorithm to solve all subproblems, in principle any mixture of algorithms could be used. The problem of deciding which algorithm to use to solve each of these subproblems can be considered a dynamic algorithm selection problem. [Lagoudakis and Littman 2000] formulates this problem as a kind of Markov Decision Problem (MDP, [Puterman 2014]), where states correspond to features of the subproblem and actions to different algorithms which can be used to solve them.

[Rodriguez et al. 2007] consider the problem of designing simple construction heuristics (see Section 2.7.2.1), which they formulate as follows: Given a set of rules H which take a partial candidate solution with n solution components as input and return a (partial) candidate solution with $n + 1$ components (i.e. H contains heuristic rules for adding solution components), which rule should one use at each step of the construction process as to maximize the quality of the constructed solution?

[Karafotias et al. 2014] consider the problem of controlling parameters in Evolutionary Algorithms (EAs), i.e. designing parameterless EAs. Like [Lagoudakis and Littman 2000] they reduce this problem to a kind of (partially observable) MDP, where states correspond to observable features of the current population and actions correspond to alternative parameter values. An alternative reduction was also considered, where the states also include the current values of ordinal parameters and actions correspond to increasing/decreasing their value.

Our objective in this section is to provide a general framework for studying this problem. Remark that both [Lagoudakis and Littman 2000] and [Karafotias et al. 2014] consider reductions to MDP-like problems and the use of RL techniques to solve them. While we share the perspective that the MDP is arguably the closest related, well-studied, computational problem, and believe in the potential of using RL methods to solve the dynamic ASP, we would like to argue that both are different problems, whose relation is non-trivial. This was also noted in [Karafotias et al. 2014]: *“There is an apparent analogy between the EA parameter control problem and the full RL problem and a Markov Decision Process (MDP). The EA state observables suggest a state space, the parameters controlled and their ranges point to an action space while a dynamic control mechanism component could be implemented by a specific RL algorithm. Although this high level mapping is straightforward, the exact formulation of the RL problem is far from trivial.”* In Section 2.4.3, we have argued for the importance of formulating computational problems in their most natural and reducible form. The dynamic ASP is no exception here. In what follows, we will therefore first present a formal definition of the dynamic ASP and only afterwards discuss possible reductions of this problem to the MDP. If nothing else, the latter has the benefit of allowing us to reason formally about the relation between both problems.

Dynamic algorithm selection as a sequential decision process: Before giving a formal definition, we provide some of the intuition underlying our formulation. Conceptually, dynamic algorithm selection, in general, can be viewed as a sequential decision process (see Figure 4.7). Assume we are given some algorithm space A_0 and a problem instance x to be solved. Unlike in the static ASP, we start solving x without selecting an algorithm instance $a \in A_0$ a priori. The crux here is that we can continue execution, as long as all algorithms in A_0 agree on the first instructions to be executed. When they do not, i.e. the next instruction depends on our choice of $a \in A_0$, a *choice point* is encountered and in order to continue execution, we must choose which of the possible instructions to execute next. This process continues until termination. Remark that our decision at the i^{th} choice point corresponds to selecting $A_i \subset A_{i-1}$ consistent with our choice (\sim partially selecting an algorithm). At each point, A_i can be viewed as the subset of algorithms consistent with the execution thus far. At some point, we may have $|A_i| = 1$ after which no further choice points are encountered as we essentially selected a single instance from A_0 .

```

while not terminated do
  if choice point then
    decide on continuation;
  end if
  execute next instruction;
end while

```

Figure 4.7: Dynamic algorithm selection as a sequential decision process.

On the other hand, we may have $|A_i| > 1$ when execution terminates, indicating that multiple algorithm instances solve x in the exact same way.

The choice machine: In what follows, we formalize this notion. To this end, we extend the Turing Machine (TM) formalism (see Section 2.3.1.2) to include choice points, alternative decisions, as well as a notion of the desirability of an execution.

Definition 4.4: Dynamic algorithm selection process (TM_{asp})

We define a TM_{asp} as a 10-tuple $\langle Q, q_0, F, \Gamma, B, \Sigma, \Delta, \kappa, \delta', \rho \rangle$, with $Q, q_0, F, \Gamma, B, \Sigma$ as in Definition 2.8 (TM) and $\Delta, \kappa, \delta', \rho$ defined as follows:

Δ is a set of *choice points*.

$\kappa : \Delta \rightarrow 2^{Q \times \Gamma \times \{R, L\}}$ is the domain function, mapping each choice point to a set of alternative transitions.

$\delta' : (Q \setminus F) \times \Gamma \rightarrow (Q \times \Gamma \times \{R, L\}) \cup \Delta$ is the *open* transition function. As for δ , arguments of $\delta'(q, a)$ are the current control state q and tape symbol being scanned a . The value of $\delta'(q, a)$, if defined, is either a triple (p, b, d) , where

$p \in Q$ is the next state.

$b \in \Gamma$ is the symbol written in the cell being scanned, i.e. replacing a .

d is the direction in which the head moves, either L (left) or R (right).

OR a choice point z .^a

$\rho : Q \times \Gamma \times \{L, R\} \rightarrow \mathbb{R}$ is the reward function. Representing the added “value” of a transition to overall performance.

^aIntuitively, in case algorithms in A_i do not agree about which transition to perform.

Here, TM_{asp} may be viewed as a “choice machine” (c-machine) a concept first described in [Turing 1937] as “*Machines whose motion is only partially determined by the configuration*.”¹³ When such a machine reaches one of these ambiguous configurations, it cannot go on until some arbitrary choice has been made by an external operator”. A c-machine is in essence an interactive model of computation. We did not find any formal definitions of the c-machine, and as such the one described above is our own, extended with the concept of a reward function assigning credit to the machine’s motion, borrowed from the RL community. Further adopting RL terminology, we will refer to the “external operator” as an *agent*.

As before, let $X \subseteq \Sigma^*$ denote the set of possible inputs. We continue to formalize what it means to simulate a TM_{asp} on an input $x \in X$. Similar to an ordinary TM, we use $\alpha qb\beta$ as the Instantaneous Description (ID) of a TM_{asp} and simulation of TM_{asp} on x starts in ID q_0x . Now, the move performed by TM_{asp} in an ID $\alpha qb\beta$ is determined as follows

- If $q \in F$: TM_{asp} halts.
- If $\delta'(q, b) \in (Q \times \Gamma \times \{R, L\})$: TM_{asp} behaves like an ordinary TM and moves to ID $\text{step}(\alpha qb\beta, \delta'(q, b))$.
- If $\delta'(q, b) \in \Delta$: the *agent* selects a transition $t \in \kappa(\delta'(q, b))$ and TM_{asp} will move to ID $\text{step}(\alpha qb\beta, t)$.

We use $id \vdash id'$ to denote that TM_{asp} could move from ID id to ID id' in a single move. Finally, $id \vdash^* id'$ indicates that simulating TM_{asp} from id for zero, one or more moves results in id' for some choices of the agent.

Remark that unlike an ordinary deterministic TM M' which specifies an execution for each input x ($e_x^{M'}$), a TM_{asp} M specifies a set of possible executions E_x^M , which we will also denote E_x in contexts where M is clear. Each $e = (a, q, b, d) \in E_x$ satisfies conditions (1), (3) and (4) from Definition 2.9, as well as

$$2. (q_i, b_i, d_i) = \delta'(q_{i-1}, a_i) \vee (q_i, b_i, d_i) \in \kappa(\delta'(q_{i-1}, a_i)), \forall 1 \leq i \leq |a|.$$

Let the desirability of an execution e be the sum of rewards associated with its transitions, i.e. $p(e) = \sum_{i=1}^{|a|} \rho(q_i, b_i, d_i)$. The motion of a TM_{asp} is not only determined by the input, but also by the decisions made by the agent. Conceptually, the objective of an agent in the dynamic ASP is to make these decisions as to maximize the reward accumulated before halting. In the remainder of this dissertation, we will assume that any simulation of a TM_{asp} on any $x \in X$ will eventually halt, independent of the agent’s decisions. As a consequence, quantities $p(e)$ and $|E_x|$ will be finite.

¹³Here configuration refers to the Instantaneous Description (ID) of the machine, not be confused with a configuration as we defined in context of algorithm frameworks.

The deterministic dynamic ASP: Before defining the dynamic ASP as a computational problem, we must decide what form a solution should take. Given our conceptual description, an obvious choice would be “an agent that acts optimally”. However, in what follows, we argue this notion to be too general. In [Turing 1937] Alan Turing presented the c-machine as an alternative to the “automatic machine” (a-machine) which corresponds to the contemporary notion of an “ordinary” TM. From this terminology, we suspect that Alan Turing, at that point in time, envisioned the agent to be a human operating the machine. Under the assumption of free will, this human is in no way restricted in the way he/she can make these decisions. However, in the context of algorithm design (i.e. the reduction depicted in Figure 4.8, p. 137) our solution must be interpretable as an algorithm for our target problem. Put differently, given (only) an input $x \in X$, the resulting design has to simulate not only a TM_{asp} , but also the agent. We therefore require the behavior of the agent to be a (computable) function of the input x . Again adopting terminology from the RL community, we will call the function describing this behavior a *policy*. Concretely, let $t^x = (t_1^x, \dots, t_n^x)$ be the sequence of decisions made by an agent when simulating M on x . We require that there exists a (computable) function π satisfying $\pi(x, i) = t_i^x$. Note that for now, we will limit ourselves to deterministic policies. As a consequence, the execution of M on x following a policy π , which we will denote $e_x^{M_\pi}$, is uniquely determined.

We are now ready to define the dynamic ASP as follows:

Definition 4.5: Dynamic Algorithm Selection Problem (deterministic)
[Adriaensen and Nowé 2016b]

Given a set of possible inputs X and a TM_{asp} M , find a policy π^* satisfying $p(e) \leq p(e_x^{M_{\pi^*}}), \forall e \in E_x, \forall x \in X$.

Unlike in the static ASP, we are not given the algorithm space explicitly. However, the dynamic ASP $\langle M, X \rangle$ can be viewed as implicitly describing the space A_0 of all deterministic algorithms M' satisfying $e_x^{M'} \in E_x^M, \forall x \in X$. A policy is the equivalent of a selection mapping in this space with the subtle difference that it does not discriminate between algorithm instances which solve x in the same way (i.e. satisfying $e_x^{M''} = e_x^{M'}$).

A Probabilistic extension: Remark that the dynamic ASP as defined above, and in [Adriaensen and Nowé 2016b], only captures selection amongst deterministic algorithms as in control states TM_{asp} behaves like an ordinary TM and the decisions at choice points are made deterministically. In what follows, we extend our definition to also capture the design of randomized algorithms. To this end, we first define the *probabilistic* dynamic algorithm selection process PTM_{asp} as an extension of the Probabilistic TM (PTM, see Definition 2.24), similar to how we defined TM_{asp} as an extension of the deterministic TM.

Definition 4.6: Probabilistic Dynamic Algorithm Selection Process (PTM_{asp})

The PTM_{asp} is defined as a 10-tuple $\langle Q, \Delta, q_0, F, \Gamma, B, \Sigma, \kappa, \delta', \rho \rangle$, with $Q, \Delta, q_0, F, \Gamma, B, \Sigma, \kappa, \rho$ as in Definition 4.4 and δ' the *open* transition probabilities,

$$((Q \setminus F) \times \Gamma) \times ((Q \times \Gamma \times \{R, L\}) \cup \Delta) \rightarrow [0, 1]$$

satisfying $\delta'((q, a), z) > 0 \wedge z \in \Delta \implies \delta'((q, a), z) = 1$, i.e. when reading a in a control state q we are either in a choice point z (when $\delta'(q, a, z) = 1$) XOR we transition to a control state p , write b and move the head in direction d , with likelihood $\delta'((q, a), (p, b, d))$.

In a control state, a PTM_{asp} moves like an ordinary PTM. In a choice point z , the transition is chosen from $\kappa(z)$ by an agent. The set of possible executions E_x will now be those (a, q, b, d) satisfying conditions (1), (3) and (4) from Definition 2.9 and for which (exactly) one of the following holds for $\forall 1 \leq i \leq |a|$:

- $\delta'(q_{i-1}, a_i, (q_i, b_i, d_i)) > 0$ (control state)
- $\exists z \in \Delta : \delta'(q_{i-1}, a_i, z) = 1 \wedge (q_i, b_i, d_i) \in \kappa(z)$ (choice point)

We extend the range of admissible behaviors (policies) for the agent in two ways:

- Decisions made at choice points may depend, not only on x , but, also on stochastic events (i.e. transitions) that occurred during the simulation of PTM_{asp} leading up to this choice point.
- Decisions may be made in a randomized fashion.

Clearly, a policy no longer uniquely determines the execution of a PTM_{asp} M for any given input x . Rather, it induces a distribution over the set of possible executions E_x . Let $\pi(x, e', t)$ be the likelihood that an agent following a policy π chooses a transition $t \in \kappa(z)$ in a choice point $z \in \Delta$, where e' is the (partial) execution leading up to the choice point. The likelihood that simulating M on x according to policy π results in an execution $e = (a, q, b, d) \in E_x$ is given by

$$\begin{aligned} \text{pr}(e|\pi, x) = & \prod_{i=1}^{|a|} \left([\delta'(q_{i-1}, a_i, (q_i, b_i, d_i)) = 0] \pi(x, (a_{1:i-1}, q_{1:i-1}, b_{1:i-1}, d_{1:i-1}), (q_i, b_i, d_i)) \right. \\ & \left. + \delta'(q_{i-1}, a_i, (q_i, b_i, d_i)) \right). \end{aligned} \quad (4.3)$$

Remark that $\delta'(q_{i-1}, a_i, (q_i, b_i, d_i)) = 0$ if and only if we are in a choice point after performing $i - 1$ moves. In the probabilistic dynamic ASP we are to make decisions as to maximize the expected desirability of the resulting execution, formally:

Definition 4.7: Dynamic Algorithm Selection Problem (probabilistic)

Given a set of possible inputs X and a $\text{PTM}_{\text{asp}} M$, find a policy π^* satisfying $p(x, \pi) \leq p(x, \pi^*), \forall x \in X$, where $p(x, \pi) = \sum_{e \in E_x} \text{pr}(e|\pi, x) * p(e)$.

Selection based on runtime features: In the remainder of this section and dissertation, we will consider a further generalization of the (probabilistic) dynamic ASP. The formulation below relates to the dynamic ASP defined above, as the subset-ASP relates to the input-ASP, i.e. we restrict our candidate solutions to policies basing their decisions on specific features of the input and the execution thus far.

Definition 4.8: Dynamic Algorithm Selection Problem (constrained)

Instances of the dynamic ASP are defined by triples $\langle M, \mathcal{D}, \phi \rangle$ where

M : a dynamic algorithm selection process (see Definition 4.6).

\mathcal{D} : a distribution over a set of inputs X (*input distribution*).

ϕ : a function mapping an input, and a partial execution leading up to a choice point, to an execution context $\omega \in \Omega$. (*dynamic feature mapping*).

Candidate solutions Π are all policies π specifying the likelihood of making a decision t in a choice point z encountered in execution context ω , denoted $\pi(z, \omega, t)$. In the dynamic ASP, given $\langle M, \mathcal{D}, \phi \rangle$, we are to find a policy π^* satisfying $o(\pi) \leq o(\pi^*), \forall \pi \in \Pi$, where $o(\pi) = \mathbf{E}_{x \sim \mathcal{D}} p(x, \pi) = \sum_{x \in X} \mathcal{D}(x) \sum_{e \in E_x} \text{pr}(e|\pi, x) * p(e)$.

Here, as in the static ASP, we will assume \mathcal{D} to be given implicitly in the form of a procedure $\mathcal{D}.\text{sample}$ which can be used for generating samples according to \mathcal{D} . M and ϕ we assume to be given in the form of a computer program $\langle M, \phi \rangle$, written in an ordinary programming language extended with constructs enabling interaction with the agent:

Agent.choice(z , statements, ω): A call to `Agent.choice` specifies a choice point $z \in \Delta$, alternative decisions at that point $\kappa(z)$ and the information $\phi(x, e') = \omega$ based on which the decision must be made. The method takes as arguments:

1. A constant serving as a choice point identifier.
2. A list of statements, exactly one of which will be executed depending on the choice of the agent.
3. A set of expressions on whose values the choice made by the agent may depend.

Agent.feedback(r): Calls to `Agent.feedback` specify ρ . The method takes as arguments a single real-valued reward. Remark that this reward may be negative. Assume that over the course of an execution e , n calls were made to `Agent.feedback` with arguments r_1, \dots, r_n , the desirability of e is given by $p(e) = \sum_{i=1}^n r_i$.

A candidate solution of the dynamic ASP is a policy $\pi : \Delta \times \Omega \times (Q \times \Gamma \times \{\mathbf{R}, \mathbf{L}\}) \rightarrow [0, 1]$, which we assume to be given explicitly. In the context of algorithm design (see Figure 4.8), the design corresponding to a policy π is a program which interprets program $\langle M, \phi \rangle$, ignoring calls to `Agent.feedback` and substituting calls to `Agent.choice(z , statements, ω)` with a statement $t \in \kappa(z)$ with likelihood $\pi(z, \omega, t)$.

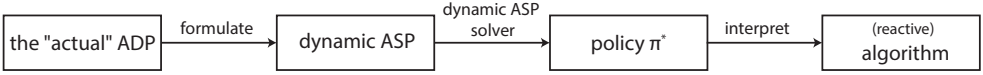


Figure 4.8: Diagram illustrating the design by dynamic ASP reduction approach.

Example - dynamic sorting ASP: In what follows, we will give an example of the dynamic ASP reduction (see Figure 4.8) in the context of our sorting ADP. For \mathcal{D} .sample we use the same procedure we have used in the static sorting ASP (see Section 4.1.1). Pseudo-code for the program $\langle M, \phi \rangle$ describing the $\text{PTM}_{\text{asp}} M$ and feature mapping ϕ is given in Algorithm 17. Here, we have three choice points $\Delta = \{z_{\text{features?}}, z_{\text{reverse?}}, z_{\text{sort}}\}$. The first choice point we encounter in a call to `Sort(l)` is $z_{\text{features?}}$. Here, we are to decide whether (t_{yes}) or not (t_{no}) to compute the relatively expensive features $f_{\text{range}}, f_{\text{sorted}}$ and f_{equal} ; based on the value of the cheap feature f_{length} . If t_{yes} , we do so using Algorithm 13, before ending up in choice point $z_{\text{reverse?}}$. Otherwise, we end up in z_{sort} . In $z_{\text{reverse?}}$ we are to decide whether to reverse l prior to sorting it (t_{yes}), or not (t_{not}), based on all four input features. Next, in z_{sort} , we have to choose one of the eight sorting algorithms ($= A$ in the static sorting ASP). This decision can be dependent on f_{length} and, if computed, also on $f_{\text{range}}, f_{\text{sorted}}$ and f_{equal} . We distinguish between iterative and recursive sorting algorithms. If we choose an iterative algorithm (selection sort, bubble sort, insertion sort, heapsort, radix sort or Tim sort) we will simply use it to sort l and return the sorted sequence. However, if we choose a recursive algorithm (merge sort or quicksort) we first decompose l into two subsequences l' and l'' which we sort by calling the `Sort` procedure recursively, before recombining the sorted subsequences into a single sorted sequence. In these recursive calls, we do not encounter $z_{\text{features?}}$ and $z_{\text{reverse?}}$ (as $|l'| < \text{size}$). However, we do encounter z_{sort} where we again can choose any of eight sorting algorithms to sort the subsequence, this time solely based on its length.¹⁴

¹⁴Our dynamic sorting ASP is similar to the one considered in [Lagoudakis and Littman 2000], extended to also include choices for preprocessing (extracting additional features and reversing) the input sequence.

Algorithm 17 Pseudocode for $\langle M, \phi \rangle$ the dynamic sorting ASP

```

1: procedure SORT( $l$ )
2:    $l \leftarrow \text{SORT}(l, |l|)$ 
3:    $\text{Agent.feedback}(\frac{-\text{timelapse}()}{|l|})$ 
4:   return  $l$ .
5: end procedure SORT

6: procedure SORT( $l, \text{size}$ )  $\triangleright$  In recursive calls, size is the length of the original sequence.
7:    $(f_{\text{length}}, f_{\text{range}}, f_{\text{sorted}}, f_{\text{equal}}) \leftarrow (|l|, \tau, \tau, \tau)$   $\triangleright$  Features of  $l$  ( $\tau =$  not (yet) computed).
8:   if  $|l| = \text{size}$  then  $\triangleright$  If non-recursive call
9:      $\text{Agent.feedback}(\frac{-\text{timelapse}()}{\text{size}})$ 
10:     $\text{Agent.choice}(z_{\text{features?}}, \{t_{\text{yes}}: \text{goto line 11}, t_{\text{no}}: \text{goto line 18}\}, f_{\text{length}})$ 
11:     $(f_{\text{length}}, f_{\text{range}}, f_{\text{sorted}}, f_{\text{equal}}) \leftarrow \text{EXTRACTFEATURES}(l)$   $\triangleright$  Compute features.
12:     $\text{Agent.feedback}(\frac{-\text{timelapse}()}{\text{size}})$ 
13:     $\text{Agent.choice}(z_{\text{reverse?}}, \{t_{\text{yes}}: \text{goto line 14}, t_{\text{no}}: \text{goto line 18}\},$ 
       $(f_{\text{length}}, f_{\text{range}}, f_{\text{sorted}}, f_{\text{equal}}))$ 
14:    for  $i = 1 : \lfloor \frac{|l|}{2} \rfloor$  do  $\triangleright$  Reverse the sequence.
15:      swap  $i^{\text{th}}$  and  $(|l| - i + 1)^{\text{th}}$  element in  $l$ 
16:    end for
17:  end if
18:   $\text{Agent.feedback}(\frac{-\text{timelapse}()}{\text{size}})$ 
19:   $\text{Agent.choice}(z_{\text{sort}}, \{$ 
     $t_{\text{selection}}: l \leftarrow \text{selection\_sort}(l, \text{size}), t_{\text{bubble}}: l \leftarrow \text{bubble\_sort}(l, \text{size}),$ 
     $t_{\text{insertion}}: l \leftarrow \text{insertion\_sort}(l, \text{size}), t_{\text{merge}}: l \leftarrow \text{merge\_sort}(l, \text{size}),$ 
     $t_{\text{quick}}: l \leftarrow \text{quicksort}(l, \text{size}), t_{\text{heap}}: l \leftarrow \text{heapsort}(l, \text{size}),$ 
     $t_{\text{radix}}: l \leftarrow \text{radix\_sort}(l, \text{size}), t_{\text{tim}}: l \leftarrow \text{tim\_sort}(l, \text{size})$ 
     $\}, (f_{\text{length}}, f_{\text{range}}, f_{\text{sorted}}, f_{\text{equal}}))$ 
20:  return  $l$ 
21: end procedure SORT

22: procedure ITERATIVESORT( $l, \text{size}$ )  $\triangleright$  Bubble, selection, insertion, heap, radix and Tim sort.
23:   sort  $l$  using non-recursive sorting algorithm
24:   Return  $l$ 
25: end procedure ITERATIVESORT

26: procedure RECURSIVESORT( $l, \text{size}$ )  $\triangleright$  Quick and merge sort.
27:   decompose  $l$  into two subsequences  $l'$  and  $l''$  (pre-processing)
28:    $l' \leftarrow \text{SORT}(l', \text{size})$ 
29:    $l'' \leftarrow \text{SORT}(l'', \text{size})$ 
30:   combine sorted subsequences  $l'$  and  $l''$  into a sorted  $l$  (post-processing)
31:   return  $l$ 
32: end procedure RECURSIVESORT

```

Finally, we discuss our choice of ρ , which is specified by calls to `Agent.feedback`. As in the static ASP, our per-input objective (\sim desirability of an execution) is the negated time ($-t$) it takes to sort the input sequence l , normalized by $|l|$. Put differently, we should choose ρ such that the sum of rewards accumulated during any execution equals $\frac{-t}{|l|}$. For our time measurements, we use `timelapse` as an auxiliary procedure. The first call to `timelapse` returns the CPU time elapsed since the computation started. Subsequent calls return the CPU time elapsed since the last call to `timelapse`. Remark that many different choices for ρ can be made without affecting optimality. The most natural choice would be to penalize each instruction proportionally to the time it takes to execute. However, doing so may be impractical; e.g. accurately measuring this cost. An alternative would be a single final call to `Agent.feedback`($\frac{-\text{timelapse}()}{|l|}$) after l has been sorted. However, it is desirable to attribute rewards to specific parts of the execution, if possible. Doing so facilitates credit assignment; e.g. rewards can only be attributed to decisions made prior to receiving them. In our sorting ASP, we decided to call `Agent.feedback`($\frac{-\text{timelapse}()}{|l|}$) before every choice point and at the end. In the MDP reduction which we will describe in Section 4.3.2.1, our coarse-grained choice of ρ reduces to the same reward function R as the impractical fine-grained option.

4.3.2 Solving the Dynamic ASP

In the remainder of this section, we turn towards solving the dynamic ASP. Here, one might expect us to discuss some of the approaches which have been taken thus far to solve the general dynamic ASP, as we have done for static ASP. However, when surveying prior art on the topic, we found that they either

- consider restrictive, specific cases of the dynamic ASP; e.g. dynamic scheduling, dynamic restart portfolios, recursive algorithm selection, etc.
- do not qualify as “automating the design of reusable algorithms”. Here, one typically “claims” to solve the dynamic ASP “online” (see Section 3.5.2.1 for a discussion).
- do not exploit the specifics of the dynamic ASP. Here, one typically reduces the dynamic ASP to a static ASP (see Section 4.4.1).

As such, we are of the opinion that

A “general dynamic ASP solver”, “worthy of the name”, does, to date, not exist.

Therefore, in the remainder of this section, we will instead focus on a line of research which we believe to show most promise towards developing such solvers in the future. Here, as in [Adriaensen and Nowé 2016b], we consider solving the dynamic ASP “by reduction to an MDP” (as is depicted in Figure 4.9). First, in Section 4.3.2.1, we discuss the reducibility

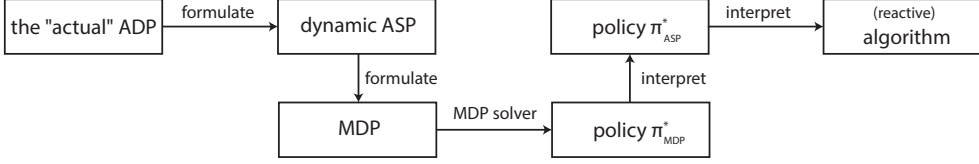


Figure 4.9: Diagram illustrating the design by MDP reduction approach.

of the dynamic ASP to the MDP. Subsequently, in Section 4.3.2.2, we discuss techniques for solving MDPs. Finally, in Section 4.3.2.3, we discuss challenges faced when solving MDPs, in general, and in the context of the dynamic ASP, in particular.

4.3.2.1 Reducibility to the MDP

In this section, we examine the relationship between the dynamic ASP and the MDP [Bellman 1957, Puterman 2014]. In particular, as in [Adriaensen and Nowé 2016b], we will show that the deterministic dynamic ASP (see Definition 4.5) is many-one reducible to the deterministic MDP. To this end, we first define the MDP as follows:

Definition 4.9: Markov Decision Problem (deterministic)

Instances of the MDP are defined by quintuples $\langle S, A, T, R, \gamma \rangle$:

S : A set of states.

$A = \bigcup_{s \in S} A_s$ a set of actions, where A_s is the set of possible actions in state s .

$T : S \times A \rightarrow S$ is the state transition function, where $T(s, a) = s'$ denotes that taking an action a in a state s results in a state s' .

$R : S \times A \rightarrow \mathbb{R}$ is the immediate reward function, where $R(s, a)$ denotes the reward we receive for taking action a in state s .

$\gamma \in [0, 1]$ is a discount factor, representing the difference in importance between short-term and long-term rewards.

Candidate solutions Π are functions of the form $\pi : S \rightarrow A$, with $\pi(s, a)$ denoting that policy π takes action a in state s . In the MDP, given $\langle S, A, P, R, \gamma \rangle$, we are to

find a policy π^* satisfying

$$\begin{aligned}\pi^*(s) &= \arg \max_{a \in A_s} (R(s, a) + \gamma * V_{\pi^*}(T(s, a))). \\ V_{\pi}(s) &= \max_{a \in A_s} (R(s, a) + \gamma * V_{\pi}(T(s, a))).\end{aligned}\tag{4.4}$$

where V_{π} is called the state-value function of policy π , giving the total discounted sum of rewards that will be received when following π starting from any state s .

dynamic ASP (Definition 4.5) \leq_m MDP (Definition 4.9): Next, we describe a general reduction from the deterministic dynamic ASP to the deterministic MDP. Remark that other reductions exist which may be preferable in practice.

Formulation: Let e_{id}^M denote the finite sequence of transitions performed when simulating $\text{TM}_{\text{asp}} M$ starting from ID id until it halts or a choice point is reached. Every instance $\langle M, X \rangle$ of the dynamic ASP (see Definition 4.5) is formulated as an instance $\langle S, A, T, R, \gamma \rangle$ of the MDP (see Definition 4.9) with:

$S = \{\alpha z a \beta \mid \exists (x, q) \in X \times Q : q_0 x \models \alpha q a \beta \wedge \delta'(q, a, z) = 1\} \cup \{\tau\}$, i.e. we have a set of states for each choice point $z \in \Delta$: one for each context in which z can be encountered. Here, the context corresponds to the tape content and head position of $\text{TM}_{\text{asp}} M$. We also have a terminal state τ , indicating that M has halted.

$A = \bigcup_{s \in S} A_s$, where A_s :

$A_{\alpha z \beta} = \kappa(z)$, i.e. we have an action for each possible continuation in z .

$A_{\tau} = \{a_{\text{noop}}\}$.

$$T(s, a) = \begin{cases} \alpha z b \beta & s \neq \tau \wedge id = \text{step}(s, a) \wedge \alpha q b \beta = \text{steps}(id, e_{id}^M) \wedge \delta'(q, b) \in \Delta \\ \tau & s = \tau \vee (s \neq \tau \wedge id = \text{step}(s, a) \wedge \alpha q \beta = \text{steps}(id, e_{id}^M) \wedge q \in F) \end{cases}.$$

i.e., taking an action a in a state s , corresponds to executing a continuation from the execution state until it halts (second case) or a choice point is reached (first case).

$$R(s, a) = \begin{cases} \rho(a) + \sum_{i=1}^{|t|} \rho(t_i) & s \neq \tau \wedge e_{\text{step}(s, a)}^M = t \\ 0 & s = \tau \end{cases}.$$

i.e., the reward associated with a single transition of the MDP is equal to the sum of the rewards associated with the transitions performed when simulating $\text{TM}_{\text{asp}} M$.

$\gamma = 1$ (undiscounted).¹⁵

¹⁵Remark that the resulting MDP is episodic (τ is the final/absorbing state). Therefore, while $\gamma = 1$, the state-value function for any policy will nonetheless be bounded.

Interpretation: Let $e_{x,i}^{M_\pi}$ denote the finite sequence of transitions performed when simulating $\text{TM}_{\text{asp}} M$ on x according to π until the i^{th} choice point is reached. A candidate solution of the MDP $\langle S, A, T, R, \gamma \rangle$, taking the form of a policy $\pi_{\text{mdp}} : S \rightarrow A$, can be interpreted as a policy π_{asp} of the dynamic ASP $\langle M, X \rangle$ with $\pi_{\text{asp}}(x, i) = \pi_{\text{mdp}}(\alpha z a \beta, t)$ where $\alpha q a \beta = \text{steps}(q_0 x, e_{x,i}^{M_{\pi_{\text{asp}}}})$ and $z = \delta'(q, a)$.

probabilistic dynamic ASP (Definition 4.7): While [Adriaensen and Nowé 2016b] restricted itself to the deterministic setting, in the stochastic setting a similar reduction is possible between the probabilistic dynamic ASP and the stochastic MDP.

constrained dynamic ASP (Definition 4.8): One could consider a similar formulation as we described above for the unconstrained dynamic ASP. However, the interpretation would fail in general, as there is no guarantee that every optimal policy of this MDP satisfies the constraints imposed by ϕ . Put differently, we can reduce the constrained dynamic ASP to a variant of the MDP that allows us to impose restrictions of the form “ π is solely a function of Ω ”. However, this “MDP with restricted policy space” is not a well-studied problem [Goetschalckx and Ramon 2007]. Alternatively, one could choose $S = \Omega$, and A as described above; and all candidate policies of this partially-defined MDP would be guaranteed to satisfy the constraints. However, in general, for these choices of S and A , T and R cannot be defined solely as a function of $S \times A$. Put differently, the Markov property is not guaranteed to hold for resulting decision process. Nonetheless, it are these naive (and improper) reductions which are typically applied in contemporary applications of RL in the context of the dynamic ASP; e.g. [Karafotias et al. 2014]. Remark that even though T and R cannot be defined, RL techniques are nonetheless applicable, since they do not require T and P to be given explicitly. However, often overlooked is that “conventional” RL techniques do nonetheless require T and R to exist (\sim the Markov property to hold). Therefore, while applicable, if applied naively, they do not offer any guarantees about the quality of the policy learned (if any) [Singh et al. 1994].

Summary: While the unconstrained dynamic ASP is reducible to the MDP, the same does not hold for the constrained dynamic ASP.

4.3.2.2 Solving MDPs

In this section, we give an overview of techniques that are commonly used to solve MDPs.

Dynamic Programming: If S is finite, Equation 4.4 (and its stochastic variant) can be solved using Dynamic Programming (DP). Commonly used methods include Value and Policy iteration [Bellman 1957, Beranek 1961], and their extensions. These techniques

iteratively enumerate all states $s \in S$, applying the recursive rules in Equation 4.4 to improve the estimates of $V_{\pi^*}(s)$ and π^* respectively, and provably obtaining π^* in polynomial time w.r.t. S and A . If the state transition graph is acyclic, standard backward induction can be used to solve the problem in a single iteration, i.e. $\mathcal{O}(SA)$. However, in many control applications, the state-action space $S \times A$ is extremely large or infinite (continuous), rendering this approach intractable. The same holds for the MDPs encountered in the context of the dynamic ASP, and we will revisit solving large MDPs later in Section 4.3.2.3. Another limitation of the DP approach is that it requires T and R to be known (efficiently computable) for any given state-action pair. In many control applications this is not naturally the case, rather, the agent obtains (samples of) these by interactions with the *environment*. One way to overcome this problem is by constructing an analytical model of the environment and applying DP to this model, rather than the actual environment. Clearly, this is challenging and the quality of the learned policy is critically dependent on the accuracy of the model.

Reinforcement Learning (RL): Alternatively, so-called reinforcement learning techniques [Sutton and Barto 1998] (also known as approximate dynamic programming) may be used, which can learn optimal control policies based on trial & error interactions with the (simulated) environment, without requiring an explicit model of thereof.¹⁶ A wide variety of RL techniques exist. Rather than an exhaustive survey, we give a brief overview, characterizing their nature, relative strengths and weaknesses.

model-based vs. model-free: We first distinguish between model-based and model-free RL. In *model-based* RL, interactions are used to estimate/predict T and R , i.e. learn an explicit empirical model of the environment. This (approximate) model can then be used to solve the MDP using e.g. dynamic programming. Note that most of the model-free approaches described below also have model-based variants, leveraging a learned model of the environment. In *model-free* RL, we solve MDPs without learning T and R explicitly. Model-free RL techniques can roughly be subdivided into two classes, i.e. value-based and policy search techniques, which we briefly discuss below.

Value-based RL techniques learn the value function directly, which is subsequently used to improve/derive the optimal policy, similar to how this is done in policy and value iteration respectively. However, lacking a model, the state-value function V_{π^*} alone is insufficient to derive π^* from Equation 4.4. Therefore, model-free approaches will typically learn the state-action value function instead, i.e. $Q_{\pi^*}(s, a) = R(s, a) + \gamma * V_{\pi^*}(T(s, a))$ (a.k.a. the Q-function). We discuss two traditional value-based approaches:

¹⁶In the context of solving the dynamic ASP, these methods solve the dynamic ASP by simulating TM_{asp} (\sim executing code), rather than analyzing it analytically (\sim static code analysis). As such, this RL approach can be viewed as a simulation-based algorithm design approach (see Section 3.4).

Monte Carlo (MC) Learning: Classical MC simulates the MDP until termination using some fixed behavioral policy to obtain a sequence of interactions (called a trace). Here, we further distinguish between on-policy and off-policy MC:

On-policy MC is the RL equivalent of policy iteration, in that in each iteration it will learn the value function (Q_π) for a fixed policy π and subsequently update π greedily w.r.t. this estimate. One issue is that if π is deterministic, we fail to explore any other actions, making the policy improvement step impossible. One common way of addressing this problem is to follow/learn *soft* policies instead. Soft policies are stochastic policies which while acting greedily w.r.t. some estimate of the value function, will also explore other actions with some low probability. Well-known examples of soft policies are ϵ -greedy and Softmax. Note that we call this approach *on-policy* as the policy which the agent uses to simulate MDP, called the behavioral policy, is the same policy whose value is being evaluated, called the target policy.

Off-policy MC discriminates itself from on-policy MC in that the behavioral policy may, in fact, be unrelated to the target policy. One benefit of this decoupling is that it solves our exploration problem, i.e. we can use an exploratory behavioral policy and yet improve the target policy. The key question here is how to estimate the value of one policy, based on traces generated using another, i.e. how to do *off-policy policy evaluation*. In the context of the ADP, this question translates to "How do we estimate the performance of one design, given performance observations obtained using another design?", and we will investigate this question extensively in Section 6.3.2.

MC is limited in that it cannot solve non-episodic tasks and tends to be inefficient in tasks where a single episode takes a lot of time.

Temporal Difference (TD) Learning techniques attempt to overcome these limitations.

To do so, they exploit the fact that the value of taking an action in a state is decomposed in the immediate reward and the value of the resulting state (\sim long-term reward), allowing us to update the value function estimate of the previous state-action pair, based on the reward received and the estimate of the value of the current state. Here, we can again distinguish between on/off-policy techniques, classical examples of which are SARSA [Rummery and Niranjan 1994] (on-policy) and Q-Learning [Watkins and Dayan 1992] (off-policy). One weakness of TD learning, w.r.t. MC, is that it is sensitive to the distribution of rewards over the simulation. In particular, to *delayed rewards*. Take the extreme, yet not uncommon example, where reward is only received at the end of each episode (e.g. chess, +1 having won, -1 having lost and 0 for a tie). Two techniques addressing this problem are:

Reward Shaping changes the reward function, adding intermediary rewards encouraging desirable behavior (e.g. rewards for capturing pieces, penalties for having pieces captured). From a problem reformulation perspective, we reduce the original MDP to an MDP with intermediary rewards; [Ng et al. 1999] presents a general way of doing so without loss of optimality.

Eligibility Traces (ET) can be seen as a hybrid MC and TD, where the reward is attributed to multiple past actions over the course of a single episode. Remark that the use of ET in off-policy TD (e.g. Q-learning) is challenging, for the same reasons that off-policy MC is challenging, and the ideas of off-policy policy evaluation also find applications in off-policy TD with ET; e.g. [Precup et al. 2000].

Direct Policy Search: A second class of model-free RL techniques are so-called *direct* policy search methods. Remark that value-based (and DP) methods reduce the problem of finding π^* to that of finding V_{π^*} or Q_{π^*} . Direct policy search methods are motivated by the fact that the latter is often a much harder problem than the former. The MDP can be viewed as an optimization problem: Find the policy $\pi \in \Pi$ maximizing the expected sum of discounted rewards. Direct policy search methods, as their name suggests, solve this optimization problem by searching Π directly. Recall that the full space of policies is typically extremely large $|\Pi| = |A|^{|S|}$, making it crucial to guide this search. Here, a wide variety of techniques are used. Many of which are, in analogy to those discussed in Section 4.1.2, p. 114, based on similarity assumptions in policy representation space. This gives rise to the question: "*How to represent policies?*"

Tabular representations indicating an action for each state, are arguably the most natural way to represent arbitrary (deterministic) policies. However, such dense representations do not scale to large or infinite S .

Parametric representations where a fixed set of, typically numerical, parameters specify a configuration space Θ corresponding to a family of policies, i.e. each configuration θ corresponds to a policy $\pi_\theta \in \Pi$.

Non-parametric representations: An example of non-parametric representations are so-called "sparse representation" used in Learning Classifier Systems (LCS), [Urbanowicz and Moore 2009]), where a policy is specified as a set of rules, each applicable to a partition of the state-space. While LCS originate from the GP community and are largely unknown within the RL community [Lanzi 2002], "Pittsburgh-Style" LCS are essentially direct policy search methods using sparse representations and GAs.¹⁷

¹⁷"Michigan-Style" LCS, while often considered as the LCS counterpart for value-based RL techniques, do not enjoy the same theoretical guarantees w.r.t. solving MDPs.

Direct policy search techniques are often considered not to be part of “conventional RL”; e.g. they were not covered in [Sutton and Barto 1998]. One reason is that many of these policy search methods are not “RL specific” and solve a class of problems far more general than MDPs. If we regard these black box optimizers as RL techniques, e.g. the set-ASP solvers discussed in Section 4.1.2 may be as well (see Section 4.4.1).¹⁸ For instance, algorithm configurators (see p. 115) are naturally viewed as a policy search approach using parametric policy presentations. One might wonder: “Why are direct policy search methods being used to solve MDPs in practice?” Surely one would expect more specialized, value-based techniques to outperform these generic solvers? An important reason is that, unlike “conventional RL”, many direct policy search methods do not suffer from the scalability issues we will discuss in the following section.

4.3.2.3 Scaling up to large state-action spaces

Thus far we have described

1. a general reduction from the dynamic ASP to the MDP.
2. current general techniques for solving MDPs.

As such, theoretically, a general dynamic ASP solver is obtained by combining (1) + (2). In practice, things are not as simple. One issue that makes this approach impractical (*at least to date and in general*) is the fact that the MDPs resulting from (1) often have large state-action spaces, and that solving such MDPs using (2), requires an intractable amount of resources. In what remains, we discuss some techniques for addressing this scalability issue, at least in specific cases.

Exploiting sparsity: One feature which may turn large MDPs tractable is that often only a small fraction of the states is actually encountered in practice. Most RL techniques will by nature focus computational efforts on the states that are encountered most frequently in interactions with the environment. While canonical DP methods enumerate the full state space, adaptations exist, called Asynchronous DP, exploiting this sparsity. The memory issue, in this case, can be addressed by using sparse (e.g. hash) table representations.

State abstraction: Also, one may be able to compress the state space, i.e. reduce the MDP to one with a smaller state space S' , a practice also known as state abstraction. Remark that a single abstract state $s' \in S'$ of the resulting MDP will correspond to multiple ground states $S_{s'} \subseteq S$ in the original MDP, i.e. we are confounding states. Be aware that, in doing so, we risk losing the Markov property and with it many of the

¹⁸That being said, there are classes of policy search techniques that do exploit some of the specifics of the MDP; e.g. policy gradient methods like REINFORCE [Williams 1992].

guarantees conventional RL methods provide [Singh et al. 1994].¹⁹ While state abstraction is often done manually and in a heuristic fashion, there also exists a body of theory on state abstraction and even on automated abstraction discovery [Li et al. 2006].

Function approximation: Finally, arguably the most common approach to handle MDPs with large state-action spaces is through the use of *function approximation* [Sutton and Barto 1998, Chapter 8]. Here, regression models are used to represent the value function and updates thereof are generalized across similar states and actions. As argued before, in Section 3.3.2, p. 85, in using regression models, we make assumptions about the form this similarity may take. These assumptions will force different states to share (or have strongly correlated) estimates and as such we are implicitly (partially) confounding states. Again, this does not mean that this approach will necessarily fail in practice, just that “getting it to learn” may require significant experience and trial & error.

4.4 A Broader Perspective

Thus far, in this chapter, we have discussed three generic approaches which we regard as being prototypical for semi-automated algorithm design. In this section, we take a step back, relate them, and briefly discuss their respective strengths and weaknesses.

4.4.1 Relations between the ASPs

Our discussion essentially revolved around three computational problems: The set-ASP (Definition 4.1), the subset-ASP (Definition 4.3) and the dynamic ASP (Definition 4.8). Until now, we have treated these problems in isolation; however, in what follows, we will show them to be closely related; Many-one equivalent (see Section 2.4.1), to be precise.

set-ASP \leq_m subset-ASP \leq_m dynamic ASP: We first formalize what is arguably the most common perspective on the relation between these problems:

- **set-ASP \leq_m subset-ASP:** A set-ASP $\langle A, \mathcal{D}, p \rangle$ can be viewed as an instance $\langle A, \mathcal{D}, \phi, p \rangle$ of the subset-ASP, where ϕ is a constant function ($|\Omega| = 1$).
- **subset-ASP \leq_m dynamic ASP:** A static ASP $\langle A, \mathcal{D}, \phi, p \rangle$ can be viewed as a dynamic ASP $\langle M, \mathcal{D}, \phi' \rangle$, where M has a single choice point ($|\Delta| = \{z\}$), encountered exactly once, at the start of the execution ($\delta'(q_0, b, z) = 1, \forall b \in \Sigma$). Each decision in z corresponds to evaluating an algorithm $a \in A$ on x , i.e. executing $p.\text{eval}(x, a)$. The outcome of this evaluation is the only reward given to the agent before halting. Finally, we have $\phi'(x, e') = \phi(x), \forall x, e'$.

¹⁹Remark that our inability to confound arbitrary states in an MDP is closely related to our inability to reduce the constrained dynamic ASP to an MDP (see Section 4.3.2.1).

This implies that any general dynamic ASP solver can be used to solve the static ASP.

dynamic ASP \leq_m subset-ASP \leq_m set-ASP, i.e. the opposite also holds: *Any general set-ASP solver can be used to solve the subset-/dynamic ASP.* While perhaps counter-intuitive, the crux is that *selection mappings and policies are algorithms* too; and therefore can be instances of A in the set-ASP. Given earlier reductions, it suffices to show

- **dynamic ASP \leq_m set-ASP:** A dynamic-ASP $\langle M, \mathcal{D}, \phi \rangle$ can be formulated as a set-ASP $\langle A, \mathcal{D}, p \rangle$ where $A = \Pi$, i.e. all policies of the dynamic ASP, and $p.\text{eval}(x, \pi)$ simulates M with π on x and returns the sum of rewards collected by the agent.

While this may seem theoretical, we argue that solving the subset-/dynamic ASP by “set-ASP reduction” is highly practical and in many ways preferable to “conventional solution approaches” we described earlier. In Section 6.5.2, we apply this approach to our sorting subset-ASP. [Ansótegui et al. 2017, Kadioglu et al. 2017] can be viewed as prior art following this approach. Also note that many direct policy search RL methods (see 4.3.2.2, p. 145) are readily viewed as solving some variant of the set-ASP.

4.4.2 Opinion: Relative Strengths and Weaknesses

In this section, we discuss the relative strengths and weaknesses of these three approaches. In particular, we highlight what we regard to be the strengths of the set-ASP approach, ending with its main weakness, compared to the subset-/dynamic ASP approaches.

Control & transparency: In the set-ASP formulation, there is a one-to-one correspondence between the algorithm space and the design space, allowing the human designer to include/exclude arbitrary designs. A highly desirable feature in semi-automated design. Subset-/dynamic ASP reductions do not offer a comparable level of control. For instance, in the subset-ASP, our choices of A and ϕ only allow us to restrict selection mappings to procedures computing a function of the form $\Omega \rightarrow A$. Furthermore, when Ω is large, conventional subset-/dynamic ASP solvers critically rely on ML methods. In choosing an ML model, they often implicitly make assumptions restricting the selection mappings/policies that can be learned. It is often unclear, at least to non-ML experts, which mappings (likely) can(not) be learned, or what overhead is associated with computing them.

Optimizes the actual objective: In the set-ASP, our objective is to optimize the average-case performance of the resulting design, and set-ASP solvers do so directly. In contrast, in the subset-/dynamic ASP, we are to optimize the expected performance of the selected algorithm/execution, i.e. the cost of selection (computing $\phi, s/\pi$) is not minimized. Furthermore, the criterion implicitly optimized by conventional solvers (using ML) again differs. For instance, classifiers will minimize classification errors, while the actual

cost of selection errors depends on how much worse the alternative performs. We clearly observed the latter in our experiments in Section 4.2.3, where a cost-oblivious classification approach obtained on average the most accurate, yet worst performing selection mapping.

Anytime: Set-ASP solvers are commonly optimized for anytime performance, e.g. making them suitable to be used in a semi-online setting (see Section 3.5.3). In simulation-based approaches, the vast majority of the time is typically spent collecting experimental data. The question of how to collect this data (e.g. distribute tests over algorithms and inputs) is a central topic in the design of set-ASP solvers. However, in e.g. the AS community, this topic has received relatively little attention. Portfolio builders are often not anytime, and data is commonly assumed to be given, or is collected in a brute force manner. For example, in the Open Algorithm Selection Challenge (OASC) 2017, performance data was given and contestants were only compared based on submitted predictions, and there as such were no resource restrictions for portfolio building/prediction.

Off-the-shelf: Following the set-ASP approach, the human designer describes the designs of interest (A), and how to measure of their performance (\mathcal{D}, p), to the set-ASP solver, which determines which design works best automatically. While the resources it requires to do so may strongly depend on the specific formulation, the quality of its (anytime) solution will generally increase over time (monotonicity) and some are guaranteed to eventually find the best design (asymptotic correctness), *largely independent of the set-ASP*. This makes it relatively easy to (successfully) apply this approach to a wide range of ADPs. Subset-/dynamic ASP solvers do not achieve a comparable level of generality. In particular, the choice of ϕ affects not only how long a solver will take, but also whether it will find a reasonable design in the limit. Underlying this limitation, is the lack of truly “off-the-shelf” ML methods. Furthermore, at least for a non-ML expert, it is difficult to predict which choices will work well, a priori. As a consequence, finding a formulation/solver combination that “works”, often involves a relatively high amount of trial & error and human effort.

Data-inefficient: Despite its benefits, the set-ASP approach also has its shortcomings. Arguably the most blatant of these is its relatively inefficient use of data; e.g., set-ASP solvers use a performance observation in estimating the performance of only a single algorithm instance, i.e. the one used to obtain it. However, often a single observation provides information about the performance of multiple different algorithms; e.g. in solving the subset-ASP by set-ASP reduction, the performance observed for a selection mapping, only depends on the algorithm it selected ($s(\phi(x))$), and is as such equally relevant for any selection mapping which would have selected the same algorithm, i.e. $\frac{1}{|A|}^{\text{th}}$ of all selection mappings. In the context of solving the ADP, data efficiency is important as collecting data is expensive. We will revisit this limitation in Chapter 6, and examine how to address it, without losing the benefits associated with the set-ASP reduction.

4.5 Summary

In this chapter, we discussed three generic approaches which we regard as being prototypical for semi-automated algorithm design. Here, we described concrete algorithmic techniques and demonstrated some of these experimentally, using sorting as a target problem.

In Section 4.1, we considered the *design by per-set algorithm selection* approach, solving the ADP by reduction to the per-set Algorithm Selection Problem (set-ASP). Here, we are to select the best algorithm (from a given set of alternatives) to solve a given distributional problem. In Section 4.1.1, we defined the set-ASP and discussed the reduction from the ADP. In Section 4.1.2, we described algorithms for solving the set-ASP.

In Section 4.2, we considered the *design by per-input algorithm selection* approach, solving the ADP by reduction to the per-input Algorithm Selection Problem (input-ASP).²⁰ Here, we are to select the best algorithm (from a given set of alternatives) to solve any given instance of a distributional problem. In Section 4.2.1, we defined the input-ASP and the subset-ASP. The latter can be seen as a generalization of both set-/input-ASP, where we are to base our selection on specific features of the input, and it is the form in which the input-ASP is most commonly solved. We discussed the subset-ASP reduction in Section 4.2.2, and subset-ASP solvers (a.k.a. portfolio builders) in Section 4.2.3.

In Section 4.3, we considered the *design by dynamic algorithm selection* approach, solving the ADP by reduction to the dynamic Algorithm Selection Problem (dynamic ASP). Here, we are to find the best way of making multiple choices we face while solving instances of a distributional problem. In Section 4.3.1, we were (presumably) the first to present a formal definition of the dynamic ASP. Here, we presented three variants: our original deterministic formulation [Adriaensen and Nowé 2016b], a probabilistic extension thereof, and a “constrained” variant where we are to base our decisions on specific features of the input and execution thus far. In Section 4.3.2, we turned towards solving the dynamic ASP. Here, we argued that, to date, “general dynamic ASP solvers” do not exist. Subsequently, we discussed the relation of our dynamic ASPs to the Markov Decision Problem (MDP); and the potential of using Reinforcement Learning (RL) techniques to solve them.

In Section 4.4, we presented a broader perspective on these three design approaches. In Section 4.4.1, we related them, showing the set-, subset- and dynamic ASP to be many-one equivalent. This, amongst others, implies that set-ASP solvers can solve the subset-/dynamic ASP. In Section 4.4.2, we discussed what we regard to be their relative strengths and weaknesses. Here, we argued that set-ASP solvers offer a level of generality, control, transparency, anytime and asymptotic performance, that cannot be found in contemporary subset-/dynamic ASP solvers; (data-)inefficiency being their main relative weakness.

²⁰Remark that this is the spirit in which the ASP was originally formulated in [Rice 1976], and the form in which the ASP is most commonly considered in the “algorithm selection” community (p. 88).

Part 2:

**Algorithmic Contributions and
Case Studies**

5 | Programming by Configuration

In Chapters 3 and 4, we have presented a broad overview of different attempts to automate algorithm design and demonstrated some of these experimentally, using sorting as a target problem. In this chapter, we will explore one of these techniques in more depth, and apply it to design a reusable metaheuristic framework for the general combinatorial optimization problem. This chapter is structured as follows: In Section 5.1, we provide some background and motivation for the semi-automated design approach used, which solves the Algorithm Design Problem (ADP, see Section 3.1) by reduction to an Algorithm Configuration Problem (ACP), a practice we will refer to as Programming by Configuration (PbC). In Section 5.2, we demonstrate this approach, using it to design a selection hyper-heuristic for the HyFlex framework/benchmark, as described in [Adriaensen et al. 2014a]. Please note that *we are not the first to apply PbC in the context of hard combinatorial optimization* and a discussion of related research can be found in Section 5.2.1.1. In Section 5.3, we discuss the resulting design (FS-ILS) and analyze its quality attributes. Here, we discuss the design decisions made and investigate its parameter sensitivity, peak performance and simplicity, as in [Adriaensen et al. 2014b], and generality, as in [Adriaensen et al. 2015]. In Section 5.4, we present a critical reflection on this research. Finally, in Section 5.5, we summarize the content covered in each section.

5.1 Programming by Optimization

In [Adriaensen et al. 2014a], we proposed and demonstrated a semi-automated approach to solving ADPs, as an alternative to the predominant manual “graduate student search” modus operandi (see Section 3.3.1). At this point in time, we were not aware of [Hoos 2012] which advocates a very similar practice. Also, an analogous “programming paradigm” was proposed in [Ansel et al. 2009]. In the spirit of consolidation, we will adopt the terminology from [Hoos 2012] and refer to this methodology as “Programming by Optimization” (PbO). However, we wish to stress that the approaches put forward in [Ansel et al. 2009] and [Adriaensen et al. 2014a], despite some differences in contemporary realizations, are essentially the same. The crux of PbO can be captured as

1. Deliberately leave “difficult” design decisions open at design time, programming a design space rather than a single algorithm instance.
2. Use a set of automated search and analysis tools to understand the impact of design decisions and gain the insight required to eventually make them.

The observation underlying PbO is that “how to best make difficult design decisions” is, in essence, an optimization problem and that by performing trial & error experimentation the human designer is, in fact, solving this problem manually. As such, PbO aims to automate this task, which is expert knowledge poor, by (1) formulating the ADP as an optimization problem and (2) solving the resulting problem automatically. Figure 5.1 depicts this reduction. Remark that a wide range of computational problems can be viewed

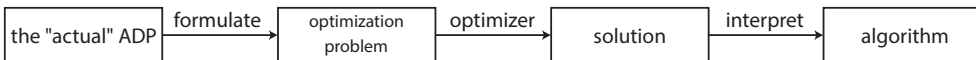


Figure 5.1: Diagram illustrating the Programming by Optimization (PbO) approach.

as optimization problems, and therefore, in its broadest interpretation, PbO captures the full range of semi-automated design approaches (see Section 3.3.3). Put differently, PbO is no algorithm for the ADP, i.e. it does not fully specify how to solve any given ADP instance, rather it is an algorithmic framework corresponding to a family of different possible realizations. However, arguably one of the most prominent realizations thus far, the one proposed in [Hoos 2012] and the one we followed in our demonstration in [Adriaensen et al. 2014a], reduces the ADP to an Algorithm Configuration Program (ACP). Figure 5.2 depicts this reduction. We will use the term “Programming by Configuration” (PbC) to distinguish this specific realization of PbO from the more general framework.

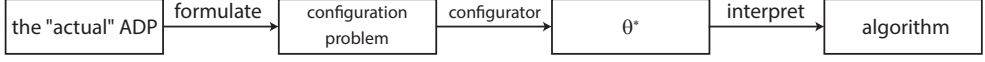


Figure 5.2: Diagram illustrating the Programming by Configuration (PbC) approach.

Thus far, we have already characterized the ACP and research efforts trying to solve it automatically in Section 3.3.3, p. 90. In Section 4.1, we discussed PbC as an instance of the “design by per-set algorithm selection approach” and briefly described various prominent configurators (p. 115). Here, we define the ACP and briefly discuss its specifics.

Definition 5.1: Algorithm Configuration Problem (ACP)

Instances of the ACP are defined by quadruples $\langle a, \Theta, \mathcal{D}, p \rangle$ where

a : an algorithm framework (see Section 2.3.3) having k configurable parameters (a.k.a. the *target algorithm*), where the i^{th} parameter can take values in Θ_i .

$\Theta \subseteq \Theta_1 \times \dots \times \Theta_k$ a set of valid configurations for a .

\mathcal{D} : a distribution over a set of inputs X (*input distribution*).

p : a function $X \times \Theta \rightarrow \mathbb{R}$, where $p(x, \theta)$ quantifies the performance of algorithm a configured using θ , denoted a_θ , on an input x (*per-input performance measure*).

Candidate solutions are elements of Θ . In the ACP, given $\langle a, \Theta, \mathcal{D}, p \rangle$, we are to find a configuration θ^* maximizing the average-case performance of a , i.e. satisfying $o(\theta) \leq o(\theta^*)$, $\forall \theta \in \Theta$, where $o(\theta) = \mathbf{E}_{x \sim \mathcal{D}} p(x, \theta)$.^a

^aVarious definitions of the ACP exist, the one presented here is a simplified version of the one given in [Hutter et al. 2009]. It differs in that we restrict our objective to optimizing average-case performance, i.e. $o(\theta) = \mathbf{E}(\mathcal{R}_{a_\theta})$, while they consider optimizing an arbitrary property of \mathcal{R}_{a_θ} .

Remark that the ACP, as formulated above, is trivially reduced to the set-ASP $\langle A_\Theta, \mathcal{D}, p \rangle$ (see Definition 4.1), with $A_\Theta = \{a_\theta \mid \theta \in \Theta\}$ and where algorithm instances are represented as configurations. In what follows, we will assume \mathcal{D}, p to be given in the form described in Section 4.1.1, i.e. as black box procedures $\mathcal{D}.\text{sample}$ and $p.\text{eval}$. The algorithm framework a is treated as a configurable black box. Θ we assume to be given as a k -tuple of domains $(\Theta_1, \dots, \Theta_k)$. For categorical and ordinal parameters, we assume Θ_i to be specified as an explicit (ordered) collection of values. Domains of numerical parameters, we assume to be given as a range $[\theta_{\min}, \theta_{\max}]$ with $\Theta_i = \{v \in V \mid \theta_{\min} \leq v \leq \theta_{\max}\}$, where $V = \mathbb{N}$ for integer-valued and $V = \mathbb{R}$ for real-valued parameters. When not all combinations of parameter values are allowed, i.e. $\Theta \subset \Theta_1 \times \dots \times \Theta_k$, this is typically specified by explicitly listing “forbidden” configurations.

5.2 Case Study: Designing Reusable Metaheuristics

In this section, we describe how we used PbC to design a reusable metaheuristic framework in [Adriaensen et al. 2014a].¹ First, in Section 5.2.1, we provide some background for our case study. Here, we characterize the design problem we considered and discuss prior design automation attempts in the area. Subsequently, in Section 5.2.2, we discuss the reduction of this problem to the ACP. Finally, in Section 5.2.3, we discuss how we solved the resulting ACP and interpret our results.

5.2.1 The Metaheuristic Design Problem

In this section, we will provide some context and motivation for our case study, what sets it apart from other “design automation” attempts in the area of hard combinational optimization, and introduce the HyFlex framework. In Chapter 2, we already discussed the challenges associated with designing methods for solving hard combinatorial optimization problems. In what follows, we briefly recap this discussion.

Many interesting combinatorial optimization problems cannot be solved exactly in polynomial time, i.e. they are NP-hard (see Section 2.5). A classical example is the traveling salesman problem (see Section 2.2.1 for more information and examples). Luckily, in practice, optimal solutions are often not required and non-exact methods can be used instead. Here, so-called heuristic optimization methods are one approach which has received a lot of attention (see Section 2.7.2). These methods often quickly find good solutions to large and otherwise intractable problems, by iteratively trying to improve a (set of) candidate solution(s). While the actual methods are problem-specific, they can often be viewed to follow a high-level search procedure which is more widely applicable. As such, it has become standard practice to conceptually separate this reusable high-level search template, commonly referred to as a metaheuristic framework (see Definition 2.32), from its problem-specific instantiation. Despite the numerous successful applications of metaheuristics to a wide variety of hard optimization problems, they are not readily applied to new problems. Put differently, the design of a heuristic optimization procedure for some target problem of interest remains challenging, is often done largely from scratch, in an ad hoc fashion, strongly relying on human intuition, experience and labor, making it a costly process.

¹Please note that an earlier version of this case study was previously conducted in [Adriaensen 2013]. In [Adriaensen et al. 2014a], we refined this earlier study, both in terms of the design space and the experimental setup we considered.

5.2.1.1 Related Research

In Section 2.7.2.5, we distinguished two ways in which the community has tried to reduce the cost associated with applying metaheuristics. In what follows, we discuss these in more detail and position our research in this context.

Design automation: First, the community has investigated the possibility of letting computers, rather than humans, design heuristic procedures. As such, we are definitely not the first to apply design automation in the context of hard combinatorial optimization. Recently, the term (generation) *hyper-heuristics* was coined to collectively refer to research in this area (see [Burke et al. 2013, Section 5] for a survey). Furthermore, we are not the first to apply algorithm configuration methods (i.e. PbC) in this context either. In fact, F-Race [Birattari et al. 2002], one of the first dedicated “configurators”, was created for exactly this purpose. To date, the majority of the ACPs in the AClib [Hutter et al. 2014], a library for benchmarking algorithm configurators, consider this setting. While most applications consider tuning a few parameters of some existing solver, e.g. [Birattari et al. 2002, Hutter et al. 2010], larger design spaces, combining ideas from many different solvers, have been considered, e.g. [KhudaBukhsh et al. 2009, Fawcett et al. 2009, López-Ibáñez and Stützle 2010, Marmion et al. 2013, Mascia et al. 2014a, Bezerra et al. 2016]. All of these can be considered applications of PbC, albeit exhibiting different degrees of automation (\sim levels of PbO, [Hoos 2012]). Remark that configurators used in this context are not typically referred to as hyper-heuristics, a term commonly associated with approaches originating from the Genetic Programming (GP) community (p. 83). Here, prior art has primarily considered automating the design of heuristics for a *specific use case*. For example, heuristic procedures for specific SAT benchmarks were designed in [Hutter et al. 2007a, KhudaBukhsh et al. 2009] (using PbC) and [Fukunaga 2004, Bader-El-Den and Poli 2007] (using GP).

Off-the-shelf metaheuristics: Beyond design-automation, the community has continued to search for off-the-shelf metaheuristics, i.e. high-level search strategies that achieve acceptable performance, independent of their problem-specific configuration. These research efforts include a wide variety of nature-inspired frameworks (see Section 2.7.2.4), as well as selection hyper-heuristics (see Section 2.7.2.5 and [Burke et al. 2013, Section 4]).

In summary, the community has approached the problem of applying existing metaheuristic frameworks to new problems in two ways:

1. Automating the configuration of metaheuristic frameworks for a specific use case.
2. Searching for metaheuristic frameworks that are easy to configure for any use case.

While (1) has shown clear potential to significantly reduce the human effort associated with applying metaheuristics and improve the performance of the resulting design, the human and computational resources required are still significant, making *made-to-measure* design not always a cost-efficient option. On the other hand, (2) is arguably still to live up to its potential, i.e. a true off-the-shelf metaheuristic does not exist and the performance gap with made-to-measure methods is substantial. Furthermore, thus far, the space of candidate off-the-shelf metaheuristics has predominantly been explored manually, following a graduate student search approach (see Section 3.3.1).

These observations have motivated the case study we performed in [Adriaensen et al. 2014a], in which we attempt to overcome these limitations by combining both approaches, i.e. we consider the semi-automated design of off-the-shelf metaheuristic frameworks. Rather than using PbC to design a heuristic optimization procedure for some specific hard combinatorial optimization problem (e.g. SAT), we use it to design a problem-independent metaheuristic framework (a high-level search strategy) which performs well, can be reused, across a wide range of combinatorial optimization problems. We are not aware of any prior applications of PbC in this context.

5.2.1.2 HyFlex

The design space in our case study consists of high-level search strategies. More specifically, we considered selection hyper-heuristics (see Section 2.7.2.5). In order to assess how “good” different high-level search strategies are, we must test them on multiple problem domains. For hyper-heuristics specifically, this implies that we not only need benchmark problems, but also low-level heuristics for all these domains (i.e. a heuristic set H). To avoid implementing all these problem-dependent aspects ourselves, we used the HyFlex framework [Ochoa et al. 2012].² In what follows, we briefly describe the HyFlex framework, how it became and still is an important benchmark for selection hyper-heuristics.

Description: HyFlex is a Java class library for developing and testing selection hyper-heuristics. HyFlex (version 1.0) provides six different problem domains: the MAX-SAT, bin packing, personnel scheduling, permutation flow shop, traveling salesman and vehicle routing problem (see Section 2.2.1 for a description). In [Adriaensen et al. 2015], we extended this benchmark set with three additional domains: the 0-1 knapsack, quadratic assignment and max-cut problem (see p. 180). For each domain HyFlex provides:

- A set of 10-12 benchmark instances, to be solved (X).
- An evaluation procedure (e), measuring the cost of a solution, to be minimized.

²http://www.asap.cs.nott.ac.uk/external/chesc2011/hyflex_download.html (version 1.0).

Table 5.1: # problem instances and low-level heuristics provided for each domain

problem domain	abbrev.	$ X $	init	ls	mut	rr	xo	$ H $
MAX-SAT	SAT	12	1	2	6	1	2	12
Bin Packing	BP	12	1	2	3	2	1	9
Personnel Scheduling	PSP	12	1	4	1	3	3	12
Permutation Flow Shop	PFS	12	1	4	5	2	3	15
Traveling Salesman	TSP	10	1	6	5	1	3	16
Vehicle Routing	VRP	10	1	4	4	2	2	13
0-1 Knapsack	KP	10	1	6	5	2	3	17
Quadratic Assignment	QAP	10	1	2	2	3	2	10
Max-Cut	MAC	10	1	3	2	3	2	11

- A set of low-level heuristics (H), subdivided in 5 categories:
 1. **initialization (init)**: Constructs a solution from scratch.
 2. **mutation (mut)**: Modifies a solution, by changing some solution components.
 3. **ruin-recreate (rr)**: Partly destroys a solution to reconstruct it afterwards.
 4. **local-search (ls)**: Similar to mutation, but guarantees that the output solution is at least as good as the input.
 5. **crossover (xo)**: Combines two solutions, into a new solution.

Each of these heuristics may furthermore be parametrized by the *intensity of mutation* (α) and *depth of search* (β) parameters.

Figure 5.1 gives an overview of the number of instances and low-level heuristics provided for each domain, per category. Note that each domain has a single construction heuristic. Detailed descriptions of these components can be found in [Hyde et al. 2011] (SAT), [Hyde et al. 2009] (BP), [Curtois et al. 2009] (PSP), [Vázquez-Rodríguez et al. 2009b] (PFS), [Vázquez-Rodríguez et al. 2009a] (TSP), [Walker et al. 2012] (VRP) and [Adriaensen et al. 2015] (KP, QAP, MAC). One of HyFlex's core design principles is that all access to these domain-specific components must occur through a problem-independent interface. Thanks to this explicit separation, any method using HyFlex can be readily applied to any domain implemented in HyFlex, without alterations.

CHeSC 2011 competition: HyFlex has been used to support the first Cross-domain Heuristic Search Challenge (CHeSC 2011), during which the performance of all 20 contestants was evaluated (based on 31, 10 minute runs) on 30 instances, five from each of the six problem domains implemented in HyFlex. To test generalization to new instances, two of the five instances used in the competition were not available before submission,

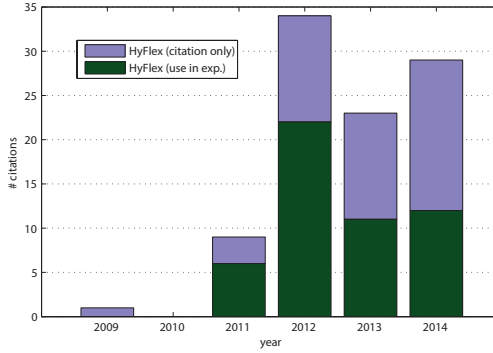


Figure 5.3: An overview of the use of HyFlex in published research (... – 2014)

i.e. were hidden. To test generalization to new domains, the TSP and VRP domains were hidden as well (also, see Appendix A.2). The winner [Misir et al. 2012b] was the algorithm obtaining the highest accumulated score across all these instances. Scores per instance were based on the ranking of median performances over 31 runs on that instance, using the pre-2010 Formula One scoring system, where the top eight algorithms score respectively 10, 8, 6, 5, 4, 3, 2 and 1 point.

HyFlex as a benchmark: After the competition, HyFlex became a popular benchmark for selection hyper-heuristics and has been used in the implementation of many hyper-heuristics ever since. Figure 5.3 shows the number of citations³ and use⁴ of HyFlex up to the year 2014. In comparing these numbers to those of other frameworks for testing cross-domain hyper-heuristics [Ryser-Welch and Miller 2014]: Hyperion [Swan et al. 2011] and hMod [Urre et al. 2013], which are cited 17 and 5 times respectively, we find HyFlex (cited 95 times) to be the most widely adopted at the time of our case study.⁵

Note that this does not mean HyFlex is the best or without flaws. On the contrary, we would like to argue that due to some unfortunate design choices, HyFlex shows only a glimpse of what is possible using hyper-heuristics. In particular, it provides too little information about domains, instances and solutions that the high-level search strategy can exploit. Furthermore, providing a good implementation for a problem domain requires a lot of effort and expert knowledge, which might be available for classical benchmark problems, but less so for real-world problems. Similar concerns were uttered in [Mascia and Stützle 2012, Di Gaspero and Urli 2012, Van Onsem et al. 2014, Swan et al. 2016].

³# articles referencing [Ochoa et al. 2012], as listed by Google Scholar™.

⁴# articles that actually use HyFlex.

⁵Beginning 2018, [Ochoa et al. 2012, Swan et al. 2011, Urre et al. 2013] were cited 126, 29, 6 times.

5.2.2 Reduction to an ACP

In the previous section, we characterized the ADP considered in our case study. In what follows, we describe how we reduced this ADP to an ACP (see Definition 5.1), i.e. we discuss our choices for a , Θ , \mathcal{D} and p .

5.2.2.1 Target Algorithm (a)

In natural occurrences of the ACP, a is some pre-existing algorithm framework with k configurable parameters p_1, \dots, p_k to be tuned. As discussed in Section 2.3.3, a can be viewed as implicitly specifying a family of algorithms, each configuration corresponding to a single algorithm instance. In the context of algorithm design, a is not given. Rather, a must be *chosen* such that each candidate design corresponds to some configuration of a . Furthermore, if we are also interested in the logic underlying the design (see Section 3.2.2), any configuration of a should, in addition, be interpretable as an understandable algorithm, e.g. be simple and modular. In particular, only part of the logic of a may be executed for a given configuration, and we would like it to be easy to eliminate any dead code.

As it turns out, there are many ways to choose a and the performance of the configurator depends on this decision. For instance, a naive approach would be to choose a to be a framework taking a single categorical parameter whose value determines which algorithm will be executed, as depicted in Algorithm 18. A benefit of this formulation is that it makes it easy to eliminate redundant logic in the interpretation of a configuration. However, it does not scale up. First, formulating large design spaces as such is tedious and is even impossible for infinite design spaces. Second, the resulting configuration space does not exhibit any structure which can be exploited by the configurator.

Algorithm 18 The “naive” delegating algorithm framework

1: function SOLVE(x, θ)	$\triangleright \theta = 1$ (a single parameter)
2: if $\theta_1 = a_1$ then	
3: return $a_1(x)$	\triangleright Execute a_1
4: else if $\theta_1 = a_2$ then	
5: return $a_2(x)$	\triangleright Execute a_2
6: ...	
7: else if $\theta_1 = a_n$ then	
8: return $a_n(x)$	\triangleright Execute a_n
9: end if	
10: end function	

A “better” alternative is to attempt to identify those aspects in which candidate designs differ, i.e. design choices, and expose only these as parameters, where values correspond to alternative decisions. Not only does this allow us to specify a large set of designs more concisely, in a modular fashion (e.g. as combinations of design decisions), similar configur-

ations will correspond to similar programs (partially share the same logic). In [Adriaensen et al. 2014a] we suggested a hierarchical description of design spaces, further generalizing this notion by parameterizing each component by its component-specific design choices. Similar to algorithm frameworks, components with parameters define families of components. This allows us to more concisely specify components as alternative design decisions and captures the intrinsic structural conditionalities between design choices in a composition. The resulting design is by nature modular and eliminating redundant logic is straightforward. This format closely resembles the one used in [Marmion et al. 2013].

Algorithm 19 The high-level search strategy considered in our case study

```

1: function SOLVE(problem,  $t_{\text{allowed}}$ , sm, os, ac, rc)                                ▷ problem is the HyFlex domain.
2:    $v_{\text{inc}}, v_{\text{run\_best}}, v_{\text{best}} \leftarrow \text{problem.init.apply}()$                 ▷ Construct initial solution.
3:   while  $t_{\text{elapsed}} < t_{\text{allowed}}$  do                                          ▷ Fixed budget  $t_{\text{allowed}}$  (see Section 2.6.4)
4:      $o \leftarrow \text{sm.select}(\text{os})$                                           ▷ Use sm to select an option from os.
5:      $v_{\text{prop}} \leftarrow o.\text{apply}(v_{\text{inc}})$                                     ▷ Apply selected option to generate proposal.
6:      $v_{\text{run\_best}} \leftarrow \text{problem.best\_of}(v_{\text{prop}}, v_{\text{run\_best}})$         ▷ Update  $v_{\text{run\_best}}$ : best since last restart.
7:      $v_{\text{best}} \leftarrow \text{problem.best\_of}(v_{\text{run\_best}}, v_{\text{best}})$           ▷ Update  $v_{\text{best}}$ : best thus far (anytime solution).
8:     if ac.accept( $v_{\text{prop}}$ ) then                                          ▷ Use ac to decide whether to accept proposal, or not.
9:        $v_{\text{inc}} \leftarrow v_{\text{prop}}$ 
10:    end if
11:    if rc.restart() then                                          ▷ Use rc to decide whether to restart search, or not.
12:      initialize()                                          ▷ Reinitialize all but a few variables (see text).
13:       $v_{\text{inc}}, v_{\text{best}} \leftarrow \text{problem.init.apply}()$                 ▷ Construct initial solution for new run.
14:    end if
15:  end while
16:  return  $v_{\text{best}}$ 
17: end function

```

In our case study, we consider simple, single point, high-level search strategies, implemented using HyFlex, as candidate designs (see Algorithm 19). First, the domain-specific construction heuristic is used to generate an initial candidate solution (line 2), then iteratively a selection mechanism is used to select an option (line 4). Here, options correspond to one or more consecutive applications of the domain-specific heuristics. Next, the selected option is applied to generate a proposal candidate solution (line 5), and an acceptance criterion is used to decide whether to accept it or not (lines 8-10). Finally, we either perform a new iteration or restart the search process (lines 11-14). Here, the top-level design decisions are the selection mechanism *sm*, options *os*, acceptance criterion *ac* and restart criterion *rc* used. For each of these, we consider many alternatives, which may, in turn, be described as combinations of subordinate components. Table 5.2 lists for each of the (families of) components considered in our case study, their type, name, component specific design choices and a brief description. In accordance with the Liskov Substitution Principle [Liskov and Wing 1994], components of the same type are interchangeable.

Note that these components can be combined and parametrized in numerous ways. In

Table 5.2: Different families of components considered in our case study

type	full name (abbrev.)	design choices	description
Algorithm	Algo (A)	sm: Select os: Options ac: Accept rc: Restart	High-level search strategy shown in Algorithm 19.
Select	Uniform (U)	n.a.	Selects an option uniformly at random.
Select	QSelect (QS)	sel: SelectionRule eval: EvaluationRule	Uses a selection rule <i>sel</i> to select an option o_i based on the quality values $q(o_i)$ assigned to each option. Initially each option is assigned 10^{300} , and ties in ranks are broken randomly. After each application of o_i an evaluation rule <i>eval</i> is used to re-evaluate o_i based on all its n_i previous applications.
SelectionRule	RouletteWheel (RW)	n.a.	Selects an option o_i with probability $\frac{q(o_i)}{\sum_{o_j} q(o_j)}$.
SelectionRule	EpsilonGreedy (EG)	ϵ : Real	Selects the best option with probability $1 - \epsilon$, and an option uniformly at random otherwise.
SelectionRule	PolyRank (PR)	k : Integer	Selects the r_i^k worst option with probability $\frac{r_i^k}{\sum_{r_j} r_j^k}$.
EvaluationRule	Change (C)	n.a.	$e(v_{inc}) - e(v_{prop})$.
EvaluationRule	Improvement (I)	n.a.	$\max(0, e(v_{inc}) - e(v_{prop}))$.
EvaluationRule	Duration (D)	n.a.	The time in ms it took to generate v_{prop} .
EvaluationRule	Accepted (AC)	n.a.	1 if $c_{proposed}$ was accepted, 0 otherwise.
EvaluationRule	Noop (N)	n.a.	1 if $v_{prop} = v_{inc}$, 0 otherwise.
EvaluationRule	NewBest (NB)	n.a.	1 if $e(v_{prop}) < e(v_{run_best})$, 0 otherwise.
EvaluationRule	Total (T)	oe: EvaluationRule	The accumulation of evaluation rule oe over all evaluations.
EvaluationRule	Average (AV)	oe: EvaluationRule	The moving average of evaluation rule oe over all evaluations.
EvaluationRule	ExpAverage (EA)	oe: EvaluationRule α : Real	The exponential moving average of evaluation rule oe over all evaluations, using a weighting factor α .
EvaluationRule	WindowAverage (WA)	oe: EvaluationRule M : Integer	The average of evaluation rule oe in the last M evaluations.
EvaluationRule	Speed (S)	n.a.	# proposals o_i generated per millisecond, i.e. $\frac{n_i+1}{\text{Total}(\text{Duration})(o_i)}$.
EvaluationRule	SpeedAccepted (SA)	n.a.	# accepted proposals o_i generated per millisecond, i.e. $\frac{\text{Total}(\text{Accepted})(o_i)+1}{\text{Total}(\text{Duration})(o_i)}$.
EvaluationRule	SpeedNew (SN)	n.a.	$\frac{\text{Total}(\text{Accepted})(o_i) - \text{Total}(\text{Noop})(o_i) + 1}{\text{Total}(\text{Duration})(o_i)}$.
EvaluationRule	ImprovementOverTime (IOT)	M : Integer	$\frac{10000 \cdot \text{WindowAverage}(\text{Improvement}, M)(o_i) + 1}{\text{Average}(\text{Duration})(o_i)}$.
Options	LocalSearch (LS)	n.a.	Each option corresponds to applying one of the <i>greedy</i> domain-specific perturbative heuristics (category ls).
Options	PerturbativeHeuristic (PH)	n.a.	Each option corresponds to applying one of the domain-specific perturbative heuristics (categories ls, mut and rr).
Options	IteratedLocalSearch (ILS)	n.a.	Each option corresponds to applying the construction or one of the non-greedy domain-specific perturbative heuristics, followed by local search. The local search process iteratively applies the greedy domain-specific perturbative heuristics of the domain, where each iteration a heuristic is selected uniformly at random. When the application of a heuristic does not lead to improvement, it is excluded from the selection, until some other heuristic finds improvement. If all heuristics are excluded, local search is terminated.
Accept	AcceptAll (AA)	n.a.	Accepts all proposals.
Accept	AcceptNoWorse (ANW)	n.a.	Accepts all non-worsening proposals.
Accept	AcceptTopList (ATL)	n : Integer	Accepts all proposals no worse than the n^{th} best solution observed so far.
Accept	AcceptBestList (ABL)	n : Integer	Accepts all proposals no worse than the n^{th} best, new best solution observed so far.
Accept	AcceptLate (AL)	t : Integer	Accepts all proposals no worse than the incumbent solution k iterations ago, where k is number of iterations performed during the t first milliseconds of the run, during which every solution better than the initial solution is accepted [Burke and Bykov 2008].
Accept	AcceptRandomWorse (ARW)	ϵ : Real	Accepts all non-worsening proposals and worsening proposals with a probability ϵ .
Accept	AcceptProbabilisticWorse (APW)	T : Real	Accepts proposals with a probability $e \frac{e(v_{inc}) - e(v_{prop})}{\mu_{impr}}$, where e is the evaluation function and μ_{impr} the average improvement in improving iterations.
Accept	AcceptSA (ASA)	T : Real	Accepts proposals with a probability $e \frac{e(v_{inc}) - e(v_{prop})}{\mu_{impr}} \cdot \frac{t_{allowed}}{t_{allowed} + t_{elapsed}}$ where $t_{allowed}$ is the time we are allowed to optimize and $t_{elapsed}$ the time we have already been optimizing.
Restart	RestartNever (RN)	n.a.	Never perform a restart.
Restart	RestartStuck (RS)	n.a.	Perform a restart when $w > \frac{t_{elapsed}}{t_{elapsed} + t_{allowed}} * w_{max}$, where w is the number of iterations passed since obtaining the best candidate solution so far and w_{max} the greatest number of iterations we ever had to wait for a new best candidate solution. As an exception, the algorithm is not restarted when the time remaining is less than the shortest time it took to find a candidate solution as good as the best candidate solution obtained so far. On restart, most variables are reinitialized. Exceptions are $t_{elapsed}$, v_{best} and w_{max} .

<pre> <root> Algorithm Algo(\$sm,LocalSearch,AcceptAll,\$restart) Algo(\$sm,\$os_not_ls,\$ac,\$rc) <sm> Select Uniform QSelect(RouletteWheel,\$pos_eval) QSelect(\$rank_sel,\$pos_eval) QSelect(\$rank_sel,\$eval) <rank_sel> SelectionRule EpsilonGreedy(0.25) PolyRank({1.0,2.0}) <pos_eval> EvaluationRule Speed SpeedAccepted SpeedNew ImprovementOverTime(5) <eval> EvaluationRule Average(\$oe) ExpAverage(\$oe,{0.2,0.5}) WindowAverage(\$oe,{5,10,20}) </pre>	<pre> <oe> EvaluationRule NewBest Improvement Change <os_not_ls> Options PerturbativeHeuristic IteratedLocalSearch <ac> Accept AcceptAll AcceptNoWorse AcceptTopList(20) AcceptBestList(10) AcceptLate(10000.0) AcceptRandomWorse(0.1) AcceptProbabilisticWorse(0.5) AcceptSA(2.0) <rc> Restart RestartNever RestartStuck </pre>
---	--

Figure 5.4: Description of the design space considered in our case study.

fact, we could compose an infinite number of components of the type `Algorithm`. In principle, all possible compositions could be considered, however, we considered a particular, finite subset in [Adriaensen et al. 2014a]. Figure 5.4 describes the resulting design space unambiguously, listing for each design choice the alternatives considered. The grammar-like notation used here resembles the one used in [Marmion et al. 2013]. Alternatives for a design choice p_i are either specified in a named section $\langle p_i \rangle$ and referenced using $\$p_i$ or anonymously and inline, in which case a comma-separated list, surrounded by curly brackets, is used for multiple alternatives. When choosing a certain alternative introduces further (conditional) design decisions, these are specified in round brackets after its name.

Next, we must specify an algorithm framework a such that each candidate design corresponds to some configuration of a . Our a takes a categorical parameter for every design choice whose values correspond to alternative decisions, resulting in a total of 12 (multi-valued) parameters, 9 corresponding to named and 3 corresponding to anonymous/inline design choices. Parameters are passed in the order in which they are defined in Figure 5.4 (named or anonymous) and we represent the i^{th} alternative decision using i . For example, design choice `sm` is translated to a parameter with 4 alternative values, i.e. $\{1, 2, 3, 4\}$. Our a uses these parameters to initialize all subordinate components bottom-up (resolving dependencies), eventually constructing and executing an instance of `Algorithm 19`.

5.2.2.2 Configuration Space (Θ)

In the ACP, configurators search a space of configurations Θ for one optimizing average-case performance. Here, Θ may be a subset of all possible configurations $\Theta_1 \times \dots \times \Theta_k$ of a , a feature which can be exploited to exclude configurations which do not correspond to candidate designs of interest. In our case study, we specified the target algorithm such that every configuration corresponds to a candidate design. However, multiple configurations correspond to the same design, as some of the parameters are conditional. Remark that conditions for a parameter to be active can be derived from its hierarchical specification, i.e. for a parameter to be “active” its corresponding design choice must either be root or be faced when choosing some alternative, chosen for an “active” design choice. For instance, the configuration $(2, 1, 2, 1, 3, 1, 1, 3, 1, 2, 8, 1)$ corresponds to the design: `Algo(Uniform,IteratedLocalSearch,AcceptSA(2.0),RestartNever)`. However, due to conditionalities, so does any $(2, 1, *, *, *, *, *, *, *, 2, 8, 1)$. While we could simply ignore this fact, without loss of optimality, the resulting configurations space would be considerably larger (\sim more difficult ACP), e.g. in our case study a has a total of 331776 different configurations, corresponding to only 2414 different designs. Luckily, most contemporary configurators allow the user to specify such conditionalities as part of the input.

5.2.2.3 Input Distribution (\mathcal{D})

In Section 4.1.1, we already discussed the choice of \mathcal{D} in general, in the context of the set-ASP reduction. Here, we discuss it specifically for our case study. A possible input $x \in X$ is any instance of a problem domain that could possibly be implemented in HyFlex. Furthermore, for each target problem there are many possible HyFlex implementations (e.g. making different choices for H). As a consequence, the possible variety of inputs our design could be presented with is virtually endless. In contrast, HyFlex “only” provides 6 different problem domains with a mere total of 68 instances. In our case study, we consider the subset of 30 instances (5 from all 6 domains) which were also used during the CheSC 2011 competition (X_{chesc} , see Appendix A.2). Our choice for these instances was mainly motivated by the availability of a baseline, i.e. the results obtained by the 20 contestants competing in CheSC 2011, which we used to assess the per-input performance.

5.2.2.4 Performance Evaluation (p)

In Section 4.1.1, we already discussed the choice of p in general. In our case study, we use the same per-input performance measure as used during the CheSC 2011 competition, i.e. we wish to minimize the cost of the solution obtained after 10 minutes (t_{allowed}) of optimization. However, we cannot use this criterion directly as $p.\text{eval}$, since the magnitude

of the objective function differs strongly across instances, even within the same domain. For example, a MAX-SAT instance with v variables will have candidate solutions with objective function values in $[0, v]$. On bin packing instances, the objective function will be in the $[0, 1]$ range. As such, this naive choice of p would cause us to largely neglect performance differences on the bin packing domain. In our case study, we attempt to overcome this problem by evaluating solution cost relative to the costs obtained by the CHESC 2011 contestants in similar settings. Concretely, $p.\text{eval}(x, \theta)$ will

1. Execute a_θ for t_{allowed} minutes on x , obtaining solution cost c_x^θ . Here, t_{allowed} is chosen such that t_{allowed} time on our machine (see Appendix A.1) corresponds to 10 minutes on the machine used during the competition.⁶
2. We determine the rank r_x^θ of c_x^θ w.r.t. the median solution costs (lower is better) obtained by the 20 contestants on x during the competition and assign a score of 10, 8, 6, 5, 4, 3, 2 or $10^{(8-r_x^\theta)}$ accordingly, which is returned as output. For instance, if the cost c_x^θ obtained is lower than the median cost obtained on x by 17 of the 20 contestants, we have $p.\text{eval}(x, \theta) = 5$. When multiple methods tie, they are assigned the average of the scores associated with the tied ranks.

Note that this guarantees that $p(x, \theta) \in [10^{-13}, 10], \forall x \in X, \forall \theta \in \Theta$, i.e. the magnitude of p will be similar on each input.

5.2.3 Solving the ACP

5.2.3.1 Setup

In the previous section, we have formulated our metaheuristic design problem as the problem of configuring a target algorithm with 12 categorical parameters. To solve this problem automatically we need to decide which of the many available configurators to use to do so, i.e. we are faced with a “configurator selection problem”. In Section 4.1.2, p. 115, we briefly describe some prominent examples. Instead of advocating any particular framework, we will discuss some of the properties we based our choice of configurator on (see Section 4.1.2 for a more general, in-depth discussion). Configurators will typically spend the vast majority of their time evaluating candidate designs (i.e. executing $p.\text{eval}$). In our case this is not different, as a single evaluation takes 10 minutes. Therefore, the key discriminating factor between frameworks is how they allocate performance evaluations to candidate designs:

⁶Determined using the benchmark program provided on the CHESC 2011 website:
<http://www.asap.cs.nott.ac.uk/external/chesc2011/benchmarking.html>.

- More promising designs should receive more runs. This way, we avoid wasting time on poor designs and are able to reliably discriminate between good designs.
- Asymptotically $\forall \theta \in \Theta : \Pr(\bar{o}(\theta) = o(\theta)) = 1$. Having this property guarantees that we, in the limit, will find the best design, i.e. asymptotic correctness (see Section 2.6.2). Not having this property, we risk overconfidence [Birattari and Dorigo 2004] as $\max_{\theta \in \Theta} \bar{o}(\theta)$ is biased towards overestimating $\max_{\theta \in \Theta} o(\theta)$.

In addition, as we can easily derive the conditionalities between the parameters in our ACP, we want a configurator that can exploit these conditionalities. In our case study, we used the focused variant of ParamILS [Hutter et al. 2009] (FocusedILS),⁷ combining all these properties.⁸ To reduce the duration of our experiments, and to add diversification, we ran 30 processes of FocusedILS in parallel on our machine (see Appendix A.1). Each of these processes used the same default parameter settings, but different random seeds.

5.2.4 Results

The results in this section were accumulated over all 30 configuration processes. In total, 52870 performance observations were obtained, distributed over the 2414 candidate designs and 30 instances. Ideally, we would like evaluations to be distributed equally amongst all instances. On average 1762.33 tests were performed per instance, with a standard deviation of 288.72. This standard deviation, while acceptable, is rather large, and is about 7 times more than what would be expected under a uniform distribution. This is due to the fact that FocusedILS considers benchmarks in a fixed order, and most designs were tested less than $|P|$ times. As discussed above, we want more promising candidate designs to receive more evaluations. Figure 5.5 shows the fraction of runs performed by the fraction of highest evaluated designs. Here, we observe that the better designs clearly receive more runs: 45% of the runs were performed using the 10% best configurations, and 20% were assigned to the very best. In addition, it shows that nearly every candidate design was tested at least once, suggesting a proper exploration of the design space.

Next, we have a look at the best designs found in our experiment. Here, we only consider those evaluated at least once on every instance. This was the case for (only) 143 of the designs. Table 5.3 shows the design decisions made (see Table 5.2 for the abbreviations used), the number of evaluations performed, and the estimated performance of the 15 top ranked designs. Here, we combine the performance observations of all 30 runs to estimate o using Equation 4.1. Looking at the design decisions, we find that 12 out of 15 of these designs use a QSelect selection mechanism with a RouletteWheel selection

⁷<http://www.cs.ubc.ca/labs/beta/Projects/ParamILS/> (version 2.3.8)

⁸In retrospect, other configurators may have been better suited, see Section 5.4 for a discussion.

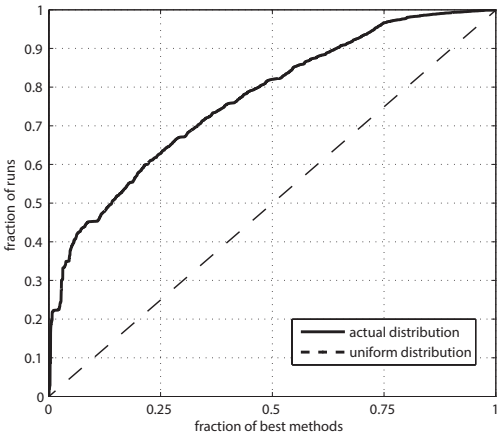


Figure 5.5: Distribution of runs over the fraction best designs

Table 5.3: Top 15 designs found by the configuration processes

r	sm	os	ac	rc	# evals	\bar{o}
1	QS(RW,SN)	ILS	APW(0.5)	RS	1573	5.77
2	QS(RW,SA)	ILS	APW(0.5)	RS	4577	5.38
3	QS(RW,SN)	ILS	APW(0.5)	RN	1322	5.23
4	QS(RW,SA)	ILS	APW(0.5)	RN	5929	5.01
5	QS(RW,SA)	ILS	ATL(20)	RN	810	4.99
6	QS(RW,SN)	ILS	AL(10000)	RN	128	4.93
7	QS(PR(2),EA(C,0.2))	ILS	APW(0.5)	RN	203	4.91
8	QS(RW,IOT(5))	ILS	ATL(20)	RS	236	4.89
9	QS(RW,IOT(5))	ILS	ATL(20)	RN	554	4.81
10	QS(EG(0.25),IOT(5))	ILS	APW(0.5)	RS	102	4.81
11	QS(RW,SN)	ILS	ATL(20)	RN	185	4.8
12	QS(RW,IOT(5))	ILS	APW(0.5)	RS	2192	4.79
13	QS(RW,IOT(5))	ILS	APW(0.5)	RN	1815	4.76
14	QS(RW,SA)	ILS	ASA(2)	RS	193	4.73
15	QS(PR(2),IOT(5))	ILS	APW(0.5)	RN	223	4.71

rule. All 15 use the `IteratedLocalSearch` options. In fact, the best designs using the `PerturbativeHeuristic` or `LocalSearch` alternatives are only evaluated at 3.83, 0.65 and ranked 65th, 133th respectively. 4 out of the 8 acceptance criteria considered appear in one of the top 15 designs, of which `AcceptProbabilisticWorse` (9) and `AcceptTopList` (4) are the most prevalent. Notably, the best method using `AcceptAll` is evaluated at 1.87 and ranks 124th, motivating the use of acceptance criteria in general. While it is less apparent whether the same holds for restart criteria, we do note that the designs ranked first and second, using `RestartStuck`, do outperform their variations, using `RestartNever`, ranked third and fourth respectively. In the remainder of this chapter, we will focus on the best design found, which we will from now onwards refer to as “Fair Share Iterated Local Search” (FS-ILS).

5.3 Fair Share Iterated Local Search (FS-ILS)

In the previous section, we focused on “how” FS-ILS was designed. In what follows, we have a closer look at the framework itself, i.e. the output of the semi-automated design process. First, we have a closer look at the design decisions made in this framework in Section 5.3.1. Subsequently, we analyze FS-ILS experimentally to gain some insight into its performance in Section 5.3.2: We perform parameter sensitivity analysis in Section 5.3.2.1, accidental complexity analysis in Section 5.3.2.2 and compare its performance to that of the 21 CHeSC 2011 contestants in Section 5.3.2.3. This research was performed in the context of [Adriaensen et al. 2014b] and was targeted at presenting FS-ILS to the research community. Finally, in Section 5.3.2.4, we describe the setup and results of an empirical study, conducted in [Adriaensen et al. 2015], where we compared 6 publicly available hyper-heuristics for HyFlex (including FS-ILS) on three *new* domains in order to assess their generality across domains.

5.3.1 Design Decisions

In this section, we describe and discuss the decisions made for each of the four top-level design choices in somewhat more detail.⁹ In particular, we provide a motivation as to why we included them as alternatives in the first place, i.e. we share the train of thought that led us to include FS-ILS as a candidate design. In what follows, we assume some background knowledge of heuristic optimization (see Section 2.7.2 for an overview).

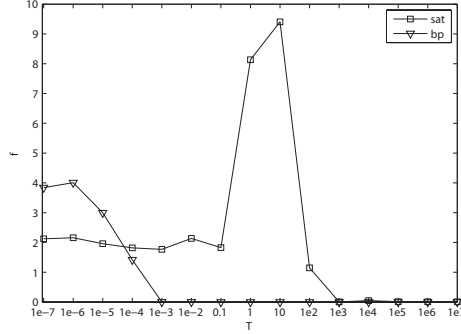
⁹A more detailed description, including pseudocode, can be found in [Adriaensen et al. 2014b].

5.3.1.1 Selection Mechanism (sm)

Options are the only source of diversification in the algorithm. Without sufficiently diverse proposals (v_{prop}) a framework risks getting stuck locally, i.e. fail to explore the search-space globally. Therefore, we included many non-greedy option selection mechanisms in our design space, leaving it up to the acceptance criterion to decide whether a proposal is “acceptable”, i.e. to control diversification. One of these was plain uniform random selection. However, we found that Uniform discriminates against fast options. For instance, consider two options o_1 and o_2 , which take 1 and 10 milliseconds respectively to generate a new candidate solution. Under uniform selection, the algorithm will spend 10 times more resources on option o_2 than it does on o_1 . This motivated us to include the alternative Speed selecting options proportionally to the rate at which they generate solutions. Using this procedure, in the limit, an equal amount of time will be spent on every option, i.e. it is *fair*. Finally, if an option’s proposal does not lead to a *new* incumbent solution (v_{inc}), either because $v_{\text{inc}} = v_{\text{prop}}$, or because the proposal is not accepted, the time spent generating it is effectively wasted. To encourage the selection of options that advance the search process, the selection procedure SpeedNew, eventually included in FS-ILS, selects options proportional to the rate at which they generate new incumbent solutions.

5.3.1.2 Options (os)

One of the key difficulties in selection hyper-heuristics is the evaluation of exploratory low-level heuristics, as they might only lead to improvement many steps later. Therefore, instead of evaluating non-greedy heuristics individually, we evaluate them followed by an Iterative Improvement (II) scheme. This allows us to more accurately assess the desirability of an exploratory move on the longer run. In addition, doing so guarantees sufficient intensification, even if options are selected in a non-greedy fashion and control of diversification is relatively crude, as is the case in FS-ILS. Only a single II procedure was included in our design space. This scheme was inspired by Variable Neighborhood Descent (VND). Each option first applies one of the non-greedy heuristics (from the *init*, *mut* and *rr* categories, see Table 5.1); e.g. for MAX-SAT there are 8 options to choose from. Subsequently, in the II procedure, we iteratively select a greedy heuristic (from the *ls* category) uniformly at random. When an application of a heuristic does not lead to improvement, it is excluded (*tabu*) from the selection, until some other heuristic finds improvement. The II scheme terminates when all heuristics have been excluded. Remark that if we assume that each application of a greedy heuristic leads to improvement, unless we are in a “local optimum for that heuristic”, our II procedure ends in a candidate solution that is a locally optimal for each of the heuristics. While this assumption is violated for many of the greedy heuristics included in HyFlex, it nonetheless leads to an elegant termination criterion.

Figure 5.6: Performance of FS-ILS using basic EMC with different T values.

5.3.1.3 Acceptance Criterion (ac)

FS-ILS uses a variant of the EMC criterion (see Section 2.7.2.2, SA). The basic EMC criterion accepts a worsening solution with probability $e^{\frac{e(v_{\text{inc}}) - e(v_{\text{prop}})}{T}}$, where T is a positive real-valued parameter called the temperature. The main difficulty is choosing T . Whether a worsening Δe is large or not is extremely domain (or even instance) dependent. For example, a worsening of 1 is the smallest possible worsening in MAX-SAT and the largest possible worsening in bin packing. As a consequence, any fixed temperature parameter T will either cause (nearly) all worsening proposals to be accepted in bin packing, or (nearly) none at all in MAX-SAT. Figure 5.6 illustrates the issue, showing the average-case performance of FS-ILS (p as described in Section 5.2.2.4) using the basic EMC criterion on the MAX-SAT and bin packing domains for a wide range of temperature values. FS-ILS uses a rather crude, yet effective, way to make the choice of T less domain dependent by expressing worsening in terms of average improvement, i.e. it accepts a worsening solution with probability $e^{\frac{e(v_{\text{inc}}) - e(v_{\text{prop}})}{T * \mu_{\text{impr}}}}$, where μ_{impr} is the average improvement in improving iterations. Put differently, we normalize the amount of worsening by dividing it by a metric that is similarly domain-dependent. We do retain T as FS-ILS's only parameter, especially since we only considered a single alternative value (i.e. 0.5) in our design process, and we will investigate the influence of this parameter on its performance in Section 5.3.2.1.

5.3.1.4 Restart Criterion (rc)

FS-ILS is restarted after it failed to find a new best solution (i.e. update $v_{\text{run_best}}$) for a certain amount of time. This criterion is inspired by the fact that a lot of frameworks find new best solutions rather regularly. When they stop doing so we possibly have to

wait a very long time for further improvement, if any (the method may be stuck). While restarting in such situation is not guaranteed to result in finding better solutions, it is often better than remaining stuck. The hard part is identifying when the algorithm is stuck. Here, the heuristic we used was inspired by the one used in CHeSC 2011 contestant VNS-TW [Hsiao et al. 2012] to detect a local optimum in the ILS process. Following this heuristic, FS-ILS is restarted if it has not found a new best solution (improved $v_{\text{run_best}}$) for $a * w_{\text{max}}$ iterations where w_{max} is the greatest number of iterations we had to wait for a new best solution thus far. The idea is that over time, w_{max} will approximate the longest time needed to still improve. The choice of a is crucial. On the one hand, to allow w_{max} to grow, it is important that a is sufficiently large (at least > 1). However, to prevent wasting too much time later on in the search (when w_{max} is somewhat accurate) a should be as small as possible. We therefore decided to lower a hyperbolically over time: $a = \frac{t_{\text{allowed}}}{t_{\text{elapsed}}}$. Initially, a will be very large, avoiding preliminary restarts; half way it is only 2, and at the end $a = 1$. As an exception, the algorithm is not restarted when the time remaining is less than the shortest time it took to find a candidate solution as good as the best candidate solution obtained so far. This prevents restarts on instances for which a lot of time is required to obtain a high-quality solution. Note that this restart criterion is rather conservative and its core design concern was to prevent restarts as much as possible, following a “if it ain’t broke, don’t fix it” philosophy, and restart only when a method is obviously stuck and enough time is available to attain a solution of similar quality.

5.3.2 Empirical Analysis

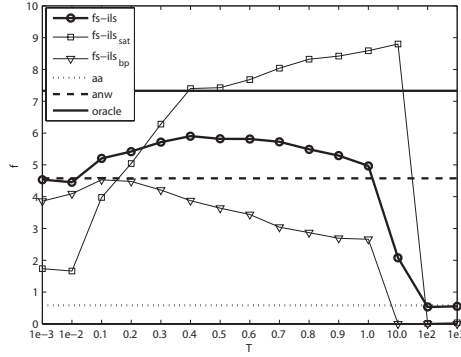
5.3.2.1 Parameter Sensitivity

Setup: The objective of the experiment described in this section is to analyze the influence of the choice of FS-ILS’s single (numerical) parameter, i.e. the temperature T used in its acceptance criterion (see Section 5.3.1.3), on its performance. To this end, we analyze FS-ILS’s performance using 15 different temperature settings, ranging from 0.001 to 1000. In our case study, T was intuitively chosen to be 0.5. To analyze the stability of this decision, 10 out of these 15 temperature values were chosen equidistantly in $[0.1, 1.0]$.

As a reference, we included three further variants of FS-ILS in our experiments:

- A variant accepting all proposals (aa)
- A variant accepting no worsening proposals. (anw)
- A variant using the basic EMC criterion with the temperature value $T \in \{10^{i-7} \mid i \in \mathbb{N} \wedge 0 \leq i \leq 14\}$, maximizing per-input performance (oracle).

In this experiment, we consider p as defined in Section 5.2.2.4 and perform 930 evaluations for each variant of FS-ILS (31 per input $x \in X_{\text{chesc}}$).

Figure 5.7: Performance of FS-ILS with different T values

Results: Figure 5.7 shows the average-case performance of FS-ILS with different temperature values T on all 6 problem domains (fs-ils), as well as for the MAX-SAT (fs-ils_{sat}) and Bin Packing (fs-ils_{bp}) domains separately. In addition, the performances of aa, anw, and the oracle are shown. Note that the scale of the x-axis is linear in $[0.1, 1.0]$, but logarithmic beyond.

We first have a look at the behavior of FS-ILS for extreme temperature values. For high temperature values, FS-ILS accepts nearly all worsening proposals and we observe a performance similar to that of aa. For low temperature values, FS-ILS accepts nearly no worsening proposals and we observe a performance similar to that of anw.

The intuitive choice of $T = 0.5$ we made in [Adriaensen et al. 2014a] seems adequate. Not only is 0.5 evaluated to be one of the best configurations, this parameter choice is furthermore stable as FS-ILS performs similarly using similar temperature choices. Note that Figure 5.7 over-dramatizes the drop in performance for temperature choices beyond $[0.1, 1.0]$ because of the change in scale of the x-axis.

In comparing Figures 5.6 and 5.7 we find that our normalization of the EMC criterion effectively solves the problem described in Section 5.3.1.3. Nonetheless, picking the best value for T is still about finding the best compromise. Some instances prefer more diversification and others less. For instance, on MAX-SAT, higher temperatures (10) are preferred, while on the Bin Packing domain, lower temperatures (0.1) performed better. Furthermore, the oracle performs extremely well, suggesting that FS-ILS can be improved by making the choice of T more adaptive.

Note that the performance evaluation for FS-ILS with $T = 0.5$ is similar to the performance estimated by FocusedILS during its design (see Table 5.3), suggesting that these results were not “tainted” by overconfidence.

5.3.2.2 Accidental Complexity

Setup: The objective of the experiment described in this section is to analyze the presence of accidental complexity in FS-ILS, i.e. whether it can be simplified without loss of performance.¹⁰ To this end, we compare FS-ILS to simpler variants. We consider simpler alternatives for each of the four top-level design choices (see also Table 5.2):

sm: The option selection mechanism used, i.e. QS(RW,SN). Here, we consider simpler variants selecting alternatives proportional to the rate at which they generate proposals, i.e. QS(RW,S), and variants using plain uniform random selection (U).

os: The use of iterative improvement in options (ILS). Here, we consider variants that do not perform iterative improvement, but consider single applications of the perturbative heuristics (from the *mut*, *ls* and *rr* categories) as options instead (PH).

ac: The acceptance criterion used (APW). Here, we consider variants accepting only non-worsening proposals (ANW) and variants accepting all proposals (AA).

rc: The use of restarts (RS). Here, we consider variants that never restart (RN).

Combining these simplifications gives rise to 35 simpler variants of FS-ILS. Remark that these variants were part of the design space considered in [Adriaensen et al. 2014a], with the objective of avoiding accidental complexity “by design”. In this experimental setup, we wish to investigate whether this attempt was successful. For each variant, we performed 300 tests, 10 for each $x \in X_{\text{chesc}}$ (see Appendix A.2), to evaluate its performance. We test the significance of the differences in performance observed when compared to FS-ILS, using the Wilcoxon signed-rank test.

Results: Table 5.4 shows for each of the 36 variants the design decisions made, their average-case performance and the p-value for the Wilcoxon signed-rank test in a pair-wise comparison with FS-ILS. We find that FS-ILS (in bold) is evaluated significantly better than its simpler variants, i.e. every design decision has a significant contribution (w.r.t. the simpler alternatives considered). However, we note that some simplifications have a greater impact on performance than others. This allows us to measure the contribution of a certain design decision to the performance of FS-ILS.

sm: The choice for speed proportional selection, i.e. QS(RW,S) or QS(RW,SN), adds great value as the best design using uniform selection is only ranked 11th and in 20 out of 24 cases configurations using a speed proportional selection scheme outperform their variant using plain uniform selection. In the 4 remaining cases, the design has no control

¹⁰In Chapter 7, we discuss the merits of this practice elaborately.

Table 5.4: Comparison of FS-ILS to simpler variants

r	sm	os	ac	rc	\bar{o}	p-value
1	QS(RW,SN)	ILS	APW	RS	5.77	1.0
2	QS(RW,SN)	ILS	APW	RN	5.54	0.0048
3	QS(RW,S)	ILS	APW	RS	5.2	4.6E-5
4	QS(RW,S)	ILS	APW	RN	4.85	2.0E-8
5	QS(RW,SN)	ILS	ANW	RN	4.58	1.6E-6
6	QS(RW,SN)	ILS	ANW	RS	4.57	8.3E-7
7	QS(RW,S)	ILS	ANW	RN	4.16	1.0E-9
8	QS(RW,S)	ILS	ANW	RS	3.87	4.2E-12
9	QS(RW,SN)	PH	APW	RS	3.23	3.4E-15
10	QS(RW,SN)	PH	APW	RN	3.09	0.0
11	U	ILS	APW	RN	2.99	0.0
12	U	ILS	APW	RS	2.96	0.0
13	U	ILS	ANW	RS	2.64	0.0
14	U	ILS	ANW	RN	2.52	0.0
15	QS(RW,S)	PH	APW	RS	2.46	0.0
16	QS(RW,S)	PH	APW	RN	2.42	0.0
17	U	PH	APW	RN	2.18	0.0
18	U	PH	APW	RS	2.15	0.0
19	QS(RW,S)	PH	ANW	RS	1.84	0.0
20	QS(RW,S)	PH	ANW	RN	1.74	0.0
21	QS(RW,SN)	PH	ANW	RN	1.53	0.0
22	QS(RW,SN)	PH	ANW	RS	1.41	0.0
23	U	PH	ANW	RS	1.36	0.0
24	U	PH	ANW	RN	1.35	0.0
25	QS(RW,SN)	ILS	AA	RN	0.67	0.0
26	QS(RW,S)	ILS	AA	RN	0.6	0.0
27	QS(RW,SN)	ILS	AA	RS	0.48	0.0
28	QS(RW,S)	ILS	AA	RS	0.48	0.0
29	U	ILS	AA	RS	0.41	0.0
30	U	ILS	AA	RN	0.4	0.0
31	U	PH	AA	RS	0.16	0.0
32	U	PH	AA	RN	0.12	0.0
33	QS(RW,SN)	PH	AA	RS	0.02	0.0
34	QS(RW,S)	PH	AA	RS	0.01	0.0
35	QS(RW,SN)	PH	AA	RN	0.0	0.0
36	QS(RW,S)	PH	AA	RN	0.0	0.0

of diversification (AA), and therefore performs poorly with non-greedy selection schemes. Plain uniform selection tends to be greedier than speed proportional selection as applying non-greedy heuristics tends to take less time than applying greedy ones. Configurations using QS(RW,SN) outperform their variant using QS(RW,S) in 10 out of 12 cases. In the cases in which it does not, no local search is performed (PH) and all worsening proposals are rejected. QS(RW,SN) selection becomes too greedy and fails to explore.

os: The use of iterative improvement is clearly important as the best design without is only ranked ninth. Furthermore, all configurations using iterative improvement (ILS) outperformed variants that do not (PH).

ac: Also, the choice of acceptance criterion is important. The four best designs use the (normalized) EMC criterion (APW). Furthermore, configurations using this acceptance criterion outperform those accepting no worsening proposals (ANW), which in turn outperform those accepting all worsening proposals (AA).

rc: The choice for the restart criterion is clearly the most controversial. On the one hand, using the restart criterion does (significantly) improve the performance of the 2 top configurations. On the other hand, not using the restart criterion is the simplification with the smallest impact and using the restart criterion is in no way consistently beneficial in all configurations. Therefore, one might consider omitting the restart criterion, which would further simplify implementation and analysis, as there is no need for reinitialization or run-specific variables.

5.3.2.3 Peak Performance

Table 7.1 shows the outcome of the CHeSC (2011) competition with FS-ILS as competitor (based on 930 runs as in Section 5.3.2.1). It shows the total score and scores on each domain for all contestants. We see that FS-ILS would have won the competition and performs best on three of the six domains. Furthermore, the worst score it obtains on any domain is higher than the worst score of any other contestant.

However, this comparison is limited in many ways. First, results for FS-ILS were obtained on a different platform and while we did use a benchmark application to somewhat reduce this effect, it makes our results sensitive to potential bias in this program. Secondly, we wish to stress the fact that this comparison is not entirely fair. To design FS-ILS, we used information not available to the other contestants, i.e. the benchmark instances used during the competition and the performance of the other contestants, giving FS-ILS an unfair advantage. Nonetheless, FS-ILS could have been a competitor and as such the comparison above does illustrate its competitiveness *on these instances*. However, these are known benchmark instances, solved numerous times in the past. A key limitation

Table 5.5: The results of the CHeSC 2011 competition with FS-ILS as competitor

r	algorithm	s_{total}	s_{sat}	s_{bp}	s_{psp}	s_{pfs}	s_{tsp}	s_{vrp}
1	FS-ILS	182.1	39.6	20	10.5	47	34	31
2	AdapHH	162.18	28.93	45	9	31	35.25	13
3	VNS-TW	115.68	28.93	2	39.5	26	15.25	4
4	ML	110	10.5	8	30	31.5	10.0	20
5	PHUNTER	80.25	7.5	3	11.5	6	22.25	30
6	EPH	74.75	0	8	9.5	16	30.25	11
7	NAHH	65	11.5	19	1	18.5	10.0	5
8	HAHA	64.27	26.93	0	25.5	0.83	0.0	11
9	ISEA	59.5	3.5	28	14.5	1.5	9	3
10	KSATS-HH	53.85	19.85	8	8	0	0	18
11	HAEA	39.33	0	2	1	5.33	9	22
12	ACO-HH	32.33	0	19	0	6.33	6	1
13	GenHive	30.5	0	12	6.5	5	2	5
14	SA-ILS	21.75	0.25	0	17.5	0	0	4
15	DynILS	20	0.0	11	0	0	9	0
16	XCJ	18.5	3.5	10	0	0	0	5
17	AVEG-Nep	16.5	10.5	0	0	0	0	6
18	GISS	16.25	0.25	0	10	0	0	6
19	SelfSearch	4	0	0	1	0	3	0
20	MCHH-S	3.25	3.25	0	0	0	0	0
21	Ant-Q	0	0	0	0	0	0	0

of our comparison here is that it does not provide any information about whether this performance will generalize to slightly different settings, e.g. the *new*, unseen instances which we would like to solve in practice. Put differently, FS-ILS was optimized by design to perform well in this comparison, it was “trained” on these instances and its performance was evaluated relative to these contestants, using a closely related performance metric. The performance we observe in this comparison is therefore likely biased towards being optimistic. The same does not hold, to the same extent, for the CHeSC 2011 contestants, as 18 of the 30 instances used were not available (“hidden”) prior to the competition. In addition, 10 of these were taken from two *new* domains (see Appendix A.2).

Despite the limitations mentioned above, these kind of comparisons are commonplace in “post-CHeSC 2011” research. In fact, it is often *the only* kind of performance evaluation performed [Van Onsem et al. 2014]. As most of these frameworks were designed manually, using the gradient student search approach (see Section 3.3.1), one may argue that the second limitation is not (or less) applicable to these frameworks, as they were not “optimized” for this comparison. However, we would like to speculate that even though not done so explicitly, as in our semi-automated design approach, they are most likely optimized for this setting implicitly. However, due to the lack of transparency of this design process, we cannot know this for sure. These observations motivated the experimental study performed in [Adriaensen et al. 2015] and which we will describe in the next section.

Table 5.6: Overview of the methods compared

framework	author	style	loc	license
AdapHH	Mustafa Misir	Select-Accept	2324	GNU GPL
EPH	David Meignan	Population-based	957	Apache 2.0
FS-ILS	Steven Adriaensen	Iterated Local Search	216	MIT
NR-FS-ILS	Steven Adriaensen	Iterated Local Search	166	MIT
ANW-HH	Steven Adriaensen	Select-Accept	27	MIT
AA-HH	Steven Adriaensen	Select(-Accept)	21	MIT

5.3.2.4 Generality

In this section, we examined the generality of FS-ILS and that of several other publicly available hyper-heuristics for HyFlex across domains. To fairly assess a framework’s generality, it is important to test it on as many *new* domains as possible, as the potential variability in domains is vast. Here, the word *new* is crucial, i.e. the framework/domain should not have been tested using this domain/framework during any phase of its design, as otherwise we risk an optimistic bias. To this end, we extend the HyFlex benchmark set, providing benchmark instances and problem-specific components for three additional problem domains.¹¹ To the best of our knowledge, this is, to date, the first and only public extension of the HyFlex benchmark since the competition in 2011.

This section is organized as follows: First, we describe the frameworks considered in our comparison. Subsequently, we introduce each of the new problem domains. Finally, we present our experimental results.

Baseline: Here, we describe the frameworks considered in our comparison in more detail. Table 5.6 gives an overview of these frameworks and their properties. Note that we include the “lines of code” metric (loc), as a rough indication of their (code) complexity.¹²

AdapHH: AdapHH [Misir et al. 2012b], made publicly available under the name GIHH,¹³ is the framework that won the CHeSC 2011 competition. At the highest level it follows a rather classical single point iterative selection and acceptance scheme, while at a lower level it successfully combines various adaptive mechanisms.¹⁴ AdapHH maintains an *adaptive heuristic set* (ADHS). Here, performance metrics are kept for each of the

¹¹Available here: <https://github.com/Steven-Adriaensen/hyflex> (domains)

¹²Measured using the cloc tool <http://cloc.sourceforge.net/> (version 1.62).

¹³Available here: <https://code.google.com/archive/p/generic-intelligent-hyper-heuristic/>

¹⁴See Appendix A.5 for a more detailed description of AdapHH’s sub-mechanisms.

low-level heuristics and after a number of iterations (called a phase) the performance of each heuristic is reassessed and a quality index is assigned. Within each phase, heuristics are applied with a frequency dependent on the assigned quality index. Also, each phase, heuristics of poor quality are temporarily (or even indefinitely) excluded. An acceptance criterion, called *adaptive iteration limited list-based threshold accepting* (AILLA), is used to decide whether to accept the solutions, generated by these heuristics, as new incumbent solution or not. Here, a list of evaluation function values of previously found new best solutions is maintained and only proposals no worse than the k^{th} element of this list are accepted. The value of k is adapted dynamically during the search. Next to these *basic* features, AdapHH also implements a restart criterion, heuristic adaptation mechanism for α and β and a hybridization scheme (learning effective pairs of heuristics).

EPH: An Evolutionary Programming Hyper-heuristic [Meignan 2011] with co-evolution.¹⁵ This population-based approach ended up 5th in the CHeSC 2011 competition. The method maintains two populations, one of candidate solutions, another of heuristic sequences. Both populations co-evolve in the sense that the new solutions introduced in the population are generated by applying the heuristic sequences, and the fitness of the heuristic sequences is related to how they perform on the current solutions. The population of solutions evolves as follows: First, an initial population is generated using the construction heuristic. Then, the heuristic sequences are applied to these solutions (in order to evaluate their performance). The resulting solution replaces the worst solution in the population if it has a cost different from all others in the population and lower than the worst. A heuristic sequence consists of a sequence of non-greedy heuristics (from the *mut*, *rr* and *xo* categories), followed by a sequence of greedy heuristics (from the *ls* category). Greedy heuristics are either applied once, or using a Variable Neighborhood Descent (VND) strategy. Each heuristic has an associated α and β value. The population of heuristic sequences is initialized randomly and each generation the population is first doubled (by recombination and mutation) and then N individuals are selected based on their fitness using tournament selection. Parameters of the framework (e.g. population size N , use of VND, etc.) are determined during a dedicated initialization phase.

(NR-)FS-ILS: We obviously also include FS-ILS in our comparison. Recall that Accidental Complexity Analysis (ACA) performed in Section 5.3.2.2 suggested that the restart criterion used may not add sufficient value to motivate its complexity. Therefore, we also include the variant without restart (named NR-FS-ILS) in our comparison. A lightweight, standalone implementation of both frameworks has been made publicly available.¹⁶

¹⁵Available here: <https://github.com/dmeignan/eph-chesc>

¹⁶Available here: <https://github.com/Steven-Adriaensen/FS-ILS>

AA-HH and ANW-HH (A_{naive}): AA-HH and ANW-HH are two simple, single point hyper-heuristics. Each of them iteratively generates a proposal by applying a heuristic (from the *mut*, *ls* and *rr* categories) selected uniformly at random. They differ in that AA-HH accepts all proposals, while ANW-HH accepts no worsening proposals, as new incumbent solution. We include these (rather naive) frameworks in our comparison because they are simple. Therefore, it is easier to interpret their results and the state-of-the-art methods described above (hereinafter collectively referred to as A_{sota}), to motivate their complexity, should clearly generalize better. For the sake of reproduction, these frameworks were also made publicly available.¹⁷

ASAP Default Hyper-heuristics: To avoid biasing our new domains to any particular framework, we did not use any of the aforementioned frameworks during the design and testing of these domains. Instead, we used the eight ASAP Default Hyper-heuristics, which were developed during the preparation and testing of the CHeSC 2011 competition software. These methods were inspired by state-of-the-art approaches and the design principles underlying some of these hyper-heuristics, can be found in [Burke et al. 2010a]. Results for these hyper-heuristics were made available for the four public domains and were used in a rehearsal competition which was conducted weekly, prior to the competition.

Problem Domains: In [Adriaensen et al. 2015], we extended the HyFlex benchmark set (X_{old}), adding three new domains (X_{new}). In what follows, we briefly introduce each of them. More detailed descriptions, including H , can be found in [Adriaensen et al. 2015].

0-1 Knapsack Problem (KP): Given a set of n items I , with associated weight $w : I \rightarrow \mathbb{R}$ and profit $p : I \rightarrow \mathbb{R}$ functions, and knapsack capacity V , we are to find the subset K satisfying the capacity constraint, i.e. $\sum_{i \in K} w(i) \leq V$ and maximizing the total profit $\sum_{i \in K} p(i)$. The search space is the subset of 2^I where each candidate solution satisfies the capacity constraint. The objective function, to be maximized, maps each subset to its total profit. As HyFlex considers the minimization of e , we use an evaluation procedure computing the negated total profit. This domain provides 10 benchmark instances, generated using the procedure employed in [Martello et al. 1999].¹⁸ Table 5.7 gives the optimal solution qualities (f_{opt}) as well as the parameters¹⁹ used to generate each instance.

¹⁷Available here: <https://github.com/Steven-Adriaensen/hyflex> (naive)

¹⁸Available here: <http://www.diku.dk/~pisinger/codes.html>.

¹⁹The range of coefficients is always $[1, r]$, with $r = 10000$, and the number of tests is always 1000.

Table 5.7: Instances provided in the KP domain

index	n	type (t)	seed (i)	f_{opt}
0	1000	no small weights (15)	12	104046
1	2000	uncorrelated (1)	13	1263861
2	2000	weakly correlated (2)	14	243145
3	2000	almost strongly correlated (5)	17	431363
4	2000	no small weights (15)	23	396167
5	5000	uncorrelated (1)	24	4417737
6	5000	weakly correlated (2)	25	954172
7	5000	uncorrelated, similar weights (9)	32	1577175
8	5000	almost strongly correlated (5)	28	1530536
9	5000	no small weights (15)	34	1467454

Quadratic Assignment Problem (QAP): Given a set of n facilities F , a set of n locations L , $d : L \times L \rightarrow \mathbb{R}$ a function specifying the distances between each pair of locations and $f : F \times F \rightarrow \mathbb{R}$ a function specifying the flows between each pair of facilities, we are to find an assignment of facilities to distinct locations that minimizes the sum of the distances multiplied by the corresponding flows. The search space S consists of all bijections $F \rightarrow L$. The evaluation function, to be minimized, is $e(s) = \sum_{x,y \in F} f(x,y) * d(s(x), s(y))$. This domain provides 10 benchmark instances, taken from the QAPLIB library [Burkard et al. 1997]. The properties and best known solution qualities (f_{prev}) of these instances, are summarized in Table 5.8.

Table 5.8: Instances provided in the QAP domain

index	name	n	f_{prev}
0	sko100a	100	152002
1	sko100b	100	153890
2	sko100c	100	147862
3	sko100d	100	149576
4	tai100a	100	21052466
5	tai100b	100	1185996137
6	tai150b	150	441786736
7	tai256c	256	43849646
8	tho150	150	7620628
9	wil100	100	273038

Table 5.9: Instances provided in the MAC domain

index	name	type	weights	$ V $	$ E $	f_{prev}
0	g3-8	torus	\mathbb{Z}	512	1536	41684814
1	g3-15	torus	\mathbb{Z}	3375	10125	283206561
2	g14	planar	$\{1\}$	800	4694	3064
3	g15	planar	$\{1\}$	800	4661	3050
4	g16	planar	$\{1\}$	800	4672	3052
5	g22	random	$\{1\}$	2000	19990	13359
6	g34	torus	$\{1, -1\}$	2000	4000	1384
7	g55	random	$\{1\}$	5000	12498	10299
8	pm3-8-50	torus	$\{1, -1\}$	512	1536	458
9	pm3-15-50	torus	$\{1, -1\}$	3375	10125	3014

Max-Cut Problem (MAC): Given a weighted graph G , with vertices V , edges E and weight function $w : E \rightarrow \mathbb{R}$. Find a cut, i.e. a partition of V into two disjoint subsets, such that the sum of the weights of the edges crossing both partitions is maximized. The search space consists of all possible cuts $V \rightarrow \{1, 2\}$ of G . The objective function, to be maximized, is given by $f(s) = \sum_{e \in E_x^s} w(e)$ where $E_x^s = \{(v_i, v_j) \in E \mid s(v_i) \neq s(v_j)\}$ is the set of edges crossing cut s . Our evaluation procedure computes $-f(s)$. This domain provides 10 benchmark instances: Instances 2-7 were generated using Rudy, a graph generator by Giovanni Rinaldi.²⁰ Instances 0-1, 8-9 are torus graphs taken from the 7th DIMACS Implementation Challenge.²¹ The properties and best known solution qualities (f_{prev}) of these instances are summarized in Table 5.9.

Comparative Study: All experiments were performed on the same machine (see Appendix A.1) and each run, a framework was given a time limit, corresponding to 10 minutes on the machine used during the CHeSC 2011 competition, as described in Section 5.2.2.4. As such it is, to some extent, possible to compare results obtained on other machines, to those reported here. To avoid bias, none of the frameworks compared were ever tested on the new instances prior to our experiments. In addition, to cancel out noise due to potential variations in system performance, the runs of different frameworks were interleaved.

In Section 5.2.2.4, we introduced a performance measure based on average ranks, which we used in the design and analysis of FS-ILS. However, as this performance measure is based on the median cost of the solutions obtained by the CHeSC 2011 contestants,

²⁰The full set can be found at <http://web.stanford.edu/~yyyy/yyyy/Gset/>

²¹<http://dimacs.rutgers.edu/Challenges/Seventh/Instances/>

information only available for instances used during the competition (i.e. X_{chesc}), doing so in this study was not possible. In addition, as we optimized FS-ILS w.r.t. this measure during its design, using the same measure in this study could introduce an unfair bias and would not allow us to assess generality w.r.t. “the choice of measure of performance”. Therefore, we report for each of the frameworks considered in our comparison (A), for a given domain (X), multiple different measures of performance:

rank: The rank of each framework $a \in A$ on domain X , ranked by increasing μ_{rank} .

μ_{rank} : The average rank of the median cost \tilde{c}_x^a obtained by each method $a \in A$ on an instance x , ranked by increasing cost. This is similar to the criterion used during the CHeSC 2011 competition.

μ_{norm} : The average normalized evaluation function value, as used in [Di Gaspero and Urili 2012]. Let C_x^a be the set of solution costs obtained by a method $a \in A$ on an instance $x \in X$. Let $C_x = \bigcup_{a \in A} C_x^a$ be the set of results obtained on x by any algorithm. Let $f_{\text{norm}}(c) = \frac{c - \min(C_x)}{\max(C_x) - \min(C_x)}$ and $\mu_{\text{norm}}(x, a) = \frac{1}{|C_x^a|} \sum_{c \in C_x^a} f_{\text{norm}}(c)$. Then $\mu_{\text{norm}}(a) = \frac{1}{|X|} \sum_{x \in X} \mu_{\text{norm}}(x, a)$.

$\sigma_{\text{norm}}^{\text{run}}$: The average standard deviation of $f_{\text{norm}}(c)$ on runs of a performed on the same input, i.e. $\frac{1}{|X|} \sum_{x \in X} \frac{1}{|C_x^a|} \sqrt{\sum_{c \in C_x^a} (f_{\text{norm}}(c) - \mu_{\text{norm}}(x, a))^2}$.

$\sigma_{\text{norm}}^{\text{input}}$: The standard deviation on the average normalized evaluation function value, across different inputs, i.e. $\frac{1}{|X|} \sqrt{\sum_{x \in X} (\mu_{\text{norm}}(x, a) - \mu_{\text{norm}}(a))^2}$.

best: The number of instances for which a obtained the best solution quality, i.e. $\sum_{x \in X} [\min(C_x^a) = \min(C_x)]$.

worst: The number of instances for which a obtained the worst solution quality, i.e. $\sum_{x \in X} [\max(C_x^a) = \max(C_x)]$.

Note that $\sigma_{\text{norm}}^{\text{run}}$ gives a measure of variation in the solution quality obtained in different runs on the same instance, while $\sigma_{\text{norm}}^{\text{input}}$ measures variations in performance on different instances of the same domain.

The remainder of this section is structured as follows: First, we compare the frameworks on all domains of the original HyFlex benchmark, next we compare them on each of the three new domains. Finally, we have a look at their performance over all instances.²²

²² $C_x^a, \forall a \in A, x \in X_{\text{all}}$ can be found here [https://github.com/Steven-Adriaensen/hyflex1t\(data\)](https://github.com/Steven-Adriaensen/hyflex1t(data)).

Table 5.10: Comparison of performance on the original HyFlex benchmark

Maximum Satisfiability (SAT)							
rank	framework	μ_{rank}	μ_{norm}	$\sigma_{\text{norm}}^{\text{run}}$	$\sigma_{\text{norm}}^{\text{input}}$	best	worst
1	FS-ILS	1.33	0.02	0.01	0.02	12	0
2	NR-FS-ILS	2.25	0.03	0.02	0.03	4	0
3	AdapHH	2.42	0.03	0.02	0.02	4	0
4	EPH	4.0	0.1	0.04	0.04	0	0
5	ANW-HH	5.0	0.39	0.06	0.14	0	0
6	AA-HH	6.0	0.86	0.08	0.03	0	12
Bin Packing (BP)							
rank	framework	μ_{rank}	μ_{norm}	$\sigma_{\text{norm}}^{\text{run}}$	$\sigma_{\text{norm}}^{\text{input}}$	best	worst
1	EPH	2.33	0.09	0.04	0.07	6	0
2	AdapHH	2.92	0.17	0.07	0.22	2	1
3	NR-FS-ILS	2.92	0.12	0.04	0.1	1	0
4	ANW-HH	3.17	0.16	0.04	0.17	3	0
5	FS-ILS	3.75	0.13	0.04	0.11	0	0
6	AA-HH	5.92	0.89	0.03	0.18	0	11
Personnel Scheduling (PSP)							
rank	framework	μ_{rank}	μ_{norm}	$\sigma_{\text{norm}}^{\text{run}}$	$\sigma_{\text{norm}}^{\text{input}}$	best	worst
1	EPH	1.92	0.13	0.09	0.08	7	0
2	NR-FS-ILS	2.21	0.15	0.07	0.11	3	0
3	FS-ILS	3.08	0.15	0.06	0.12	2	0
4	AdapHH	3.46	0.18	0.1	0.15	0	0
5	ANW-HH	4.58	0.3	0.15	0.2	1	3
6	AA-HH	5.75	0.48	0.12	0.25	0	9
Permutation Flow Shop (PFS)							
rank	framework	μ_{rank}	μ_{norm}	$\sigma_{\text{norm}}^{\text{run}}$	$\sigma_{\text{norm}}^{\text{input}}$	best	worst
1	NR-FS-ILS	1.54	0.13	0.07	0.05	10	0
2	FS-ILS	2.21	0.16	0.07	0.06	3	0
3	AdapHH	2.75	0.18	0.07	0.07	3	0
4	EPH	3.5	0.21	0.08	0.06	2	0
5	AA-HH	5.25	0.6	0.06	0.09	0	3
6	ANW-HH	5.75	0.7	0.14	0.07	0	9
Traveling Salesman Problem (TSP)							
rank	framework	μ_{rank}	μ_{norm}	$\sigma_{\text{norm}}^{\text{run}}$	$\sigma_{\text{norm}}^{\text{input}}$	best	worst
1	NR-FS-ILS	2.4	0.05	0.02	0.02	1	0
2	EPH	2.45	0.06	0.02	0.03	3	0
3	AdapHH	2.55	0.05	0.02	0.03	4	0
4	FS-ILS	2.6	0.06	0.02	0.03	3	0
5	AA-HH	5.5	0.53	0.16	0.12	0	5
6	ANW-HH	5.5	0.5	0.19	0.17	0	5
Vehicle Routing Problem (VRP)							
rank	framework	μ_{rank}	μ_{norm}	$\sigma_{\text{norm}}^{\text{run}}$	$\sigma_{\text{norm}}^{\text{input}}$	best	worst
1	FS-ILS	2.2	0.07	0.04	0.06	2	0
2	NR-FS-ILS	2.3	0.09	0.03	0.08	3	0
3	AdapHH	2.9	0.07	0.02	0.09	2	0
4	EPH	3.3	0.17	0.1	0.13	3	0
5	ANW-HH	4.7	0.31	0.09	0.22	0	2
6	AA-HH	5.6	0.78	0.05	0.31	0	8
Original HyFlex Domains (X_{old})							
rank	framework	μ_{rank}	μ_{norm}	$\sigma_{\text{norm}}^{\text{run}}$	$\sigma_{\text{norm}}^{\text{input}}$	best	worst
1	NR-FS-ILS	2.26	0.1	0.04	0.09	22	0
2	FS-ILS	2.54	0.1	0.04	0.09	22	0
3	AdapHH	2.84	0.12	0.05	0.14	15	1
4	EPH	2.92	0.13	0.06	0.09	21	0
5	ANW-HH	4.76	0.39	0.11	0.24	4	19
6	AA-HH	5.68	0.69	0.08	0.25	0	48

Original HyFlex domains (X_{old}): Table 5.10 summarizes the performance of each framework on all 68 instances in the original HyFlex benchmark set (X_{old}), per domain. Overall we find that (NR-)FS-ILS outperforms AdapHH, which in turn outperforms EPH. Furthermore, we find the relative performance of these methods on each domain to be largely consistent with those reported in Table 7.1 for $X_{\text{chesc}} \subset X_{\text{old}}$. This with some notable exceptions: EPH seems to be slightly more competitive, in particular, we observe that it outperforms AdapHH on the BP domain, while on X_{chesc} , AdapHH clearly outperforms EPH. As $\sigma_{\text{norm}}^{\text{input}}$ of AdapHH is large, this is most likely caused by AdapHH performing very good on some, while poorly on other BP instances and X_{chesc} just happening to contain more of the former. Also, we find that NR-FS-ILS performs better than FS-ILS (Overall, and on 4 out the 6 domains), while we showed FS-ILS to perform significantly better on X_{chesc} in Section 5.3.2.2. Note that while FS-ILS was optimized for X_{chesc} , FS-ILS was never tested on the 38 instances in $X_{\text{old}} \setminus X_{\text{chesc}}$, yet is similarly competitive on X_{old} as X_{chesc} , suggesting its performance generalizes well to new instances of the 6 original domains. Similar to what was observed during the rehearsal competition, we find that AdapHH, EPH and (NR-)FS-ILS (A_{sota}) clearly outperform rather naive alternatives such as AA-HH and ANW-HH (A_{naive}). Note that it is not true that “more complex” is always better, i.e. (NR-)FS-ILS while being far simpler, outperforms AdapHH and EPH. A matter we discuss in detail in Chapter 7.

0-1 Knapsack Problem (KP): Table 5.11 summarizes the performance of each framework on instances of the 0-1 Knapsack Problem. Here, we observe that AdapHH and EPH perform well, while (NR-)FS-ILS does not. Key to understanding why (NR-)FS-ILS (and ANW-HH) perform poorly, is the observation that the variability per run ($\sigma_{\text{norm}}^{\text{run}}$) is high for these methods. Also, we find that (NR-)FS-ILS performs worse for larger, more difficult instances. Upon closer inspection, we found that the iterative improvement phase on these instances (starting from an empty solution) can last up to several minutes and the quality of the candidate solution generated varies strongly.

Table 5.11: Comparison of performance on KP

0-1 Knapsack Problem (KP)							
rank	framework	μ_{rank}	μ_{norm}	$\sigma_{\text{norm}}^{\text{run}}$	$\sigma_{\text{norm}}^{\text{input}}$	best	worst
1	AdapHH	1.6	0.03	0.02	0.03	8	0
2	EPH	1.85	0.05	0.03	0.08	5	0
3	AA-HH	3.5	0.15	0.01	0.26	2	0
4	NR-FS-ILS	4.65	0.36	0.19	0.19	1	6
5	ANW-HH	4.65	0.33	0.15	0.32	0	4
6	FS-ILS	4.75	0.39	0.22	0.21	1	2

Quadratic Assignment Problem (QAP): Table 5.12 summarizes the performance of each framework on instances of the Quadratic Assignment Problem. Here, NR-FS-ILS performs best and all A_{sota} perform better than A_{naive} . Note that AA-HH clearly outperforms ANW-HH, which performs worst on all QAP instances.

Table 5.12: Comparison of performance on QAP

Quadratic Assignment Problem (QAP)							
rank	framework	μ_{rank}	μ_{norm}	$\sigma_{\text{norm}}^{\text{run}}$	$\sigma_{\text{norm}}^{\text{input}}$	best	worst
1	NR-FS-ILS	1.95	0.1	0.05	0.08	5	0
2	AdapHH	2.5	0.1	0.05	0.07	2	0
3	FS-ILS	2.85	0.1	0.04	0.08	3	0
4	EPH	3.7	0.13	0.06	0.07	0	0
5	AA-HH	4.0	0.15	0.03	0.11	1	0
6	ANW-HH	6.0	0.63	0.15	0.07	0	10

Max-Cut Problem (MAC): Table 5.13 summarizes the performance of each framework on instances of the Max-Cut problem. Here we observe, somewhat surprisingly, that AA-HH performs best, followed by AdapHH. Looking at the per-input performance, we find that AA-HH performs best on all instances with unit weight edges (2-9), but ranks only fifth on instance 0. This may be because the worsening proposed by exploratory operators in the weighted case tends to be much greater (more variable) than in the unit case. EPH and especially ANW-HH perform clearly the worst. The performance of (NR-)FS-ILS varies strongly from instance to instance (i.e. high $\sigma_{\text{norm}}^{\text{input}}$). We find performance to be mediocre to poor on the larger instances (1,5,7,9), while competitive on the smaller.

Table 5.13: Comparison of performance on MAC

Max Cut Problem (MAC)							
rank	framework	μ_{rank}	μ_{norm}	$\sigma_{\text{norm}}^{\text{run}}$	$\sigma_{\text{norm}}^{\text{input}}$	best	worst
1	AA-HH	1.5	0.16	0.05	0.1	8	0
2	AdapHH	2.25	0.19	0.07	0.07	1	0
3	NR-FS-ILS	3.1	0.31	0.07	0.2	0	0
4	FS-ILS	3.95	0.34	0.08	0.22	1	2
5	EPH	4.6	0.47	0.12	0.14	0	1
6	ANW-HH	5.6	0.7	0.1	0.08	0	7

Table 5.14: Comparison of performance across the extended HyFlex benchmark

Original HyFlex Domains (X_{old})							
rank	framework	μ_{rank}	μ_{norm}	σ_{norm}^{run}	σ_{norm}^{input}	best	worst
1	NR-FS-ILS	2.26	0.1	0.04	0.09	22	0
2	FS-ILS	2.54	0.1	0.04	0.09	22	0
3	AdapHH	2.84	0.12	0.05	0.14	15	1
4	EPH	2.92	0.13	0.06	0.09	21	0
5	ANW-HH	4.76	0.39	0.11	0.24	4	19
6	AA-HH	5.68	0.69	0.08	0.25	0	48
New Domains (X_{new})							
rank	framework	μ_{rank}	μ_{norm}	σ_{norm}^{run}	σ_{norm}^{input}	best	worst
1	AdapHH	2.12	0.11	0.04	0.09	11	0
2	AA-HH	3.0	0.15	0.03	0.18	11	0
3	NR-FS-ILS	3.23	0.26	0.1	0.2	6	6
4	EPH	3.38	0.22	0.07	0.2	5	1
5	FS-ILS	3.85	0.28	0.12	0.22	5	4
6	ANW-HH	5.42	0.55	0.13	0.25	0	21
Extended Hyflex Benchmark (X_{all})							
rank	framework	μ_{rank}	μ_{norm}	σ_{norm}^{run}	σ_{norm}^{input}	best	worst
1	NR-FS-ILS	2.56	0.15	0.06	0.15	28	6
2	AdapHH	2.62	0.11	0.05	0.12	26	1
3	FS-ILS	2.94	0.15	0.06	0.17	27	4
4	EPH	3.06	0.15	0.07	0.14	26	1
5	AA-HH	4.86	0.53	0.06	0.34	11	48
6	ANW-HH	4.96	0.44	0.12	0.26	4	40

Overview: Table 5.14 summarizes the performance of each method on all 98 instances in the extended HyFlex benchmark set (X_{all}), partitioned in X_{old} and X_{new} (see Appendix A.2). At first sight (rank and μ_{rank} on X_{all}) one might conclude that NR-FS-ILS generalizes best. When looking at μ_{norm} , however, AdapHH performs best across X_{all} . Furthermore, the σ_{norm} and min values suggest the performance of AdapHH to be more reliable. Remark that AdapHH ranks second or third on 7 out of the 9 domains (KP first and PSP fourth) and only performs worst on a single instance. In addition, cross-domain generalization should be measured w.r.t. new instances (not used during design), i.e. X_{new} . Here, we find that AdapHH clearly outperforms all others. Curiously, AA-HH ranks second, i.e. we cannot generalize the clear superiority of A_{sota} over A_{naive} observed on X_{old} . However, we would like to argue that this seemingly dramatic realization should not be worrying. It is no “problem” that a naive method performs better on some particular domains, as long as the overall performance is worse, i.e. it generalizes poorly. Overall, we find A_{naive} to be performing clearly worse than A_{sota} on X_{all} and this for all measures of performance. Also, both frameworks show similar performance, suggesting that the extended benchmark set is well-balanced. Finally, as FS-ILS performs worse than NR-FS-ILS on X_{all} and nearly all domains considered (except for SAT), we can further simplify FS-ILS by omitting the restart mechanism.

5.4 Critical Reflection

In this section, we present a critical reflection on the research described in this chapter.

Design of FS-ILS: We made some unfortunate decisions in the design of FS-ILS. In what follows, we summarize what we, in retrospect, would have done differently.

We considered “only” 2414 different candidate designs. We limited the size of the design space, at the time, to be able to obtain the best of these (\sim solve the “constrained” ADP), with high confidence. However, in doing so, we quite likely missed better alternatives just because we excluded them a priori (\sim failed to solve the actually unconstrained ADP). In particular, for numerical parameters, we only considered 1-3 alternative values.

As configurator, we used FocusedILS. In retrospect, this was not the best choice. Given our small design space, exhaustive, incremental comparison methods such as F-Race would likely have performed better than FocusedILS. Also, FocusedILS does not support continuous parameters. In using other configurators that do (e.g. iRace, GGA, ROAR, SMAC), we could have avoided such crude discretization of numerical parameters.

As a training set, we chose X_{chesc} (see Section 5.2.2.3) because it allowed us to measure per-input performance (p) in a scale-independent manner (see Section 5.2.2.4). However, this choice made any post hoc comparison with the CheSC 2011 contestants unfair (see Section 5.3.2.3). A better alternative would have been to use X_{prior} and the relative performance w.r.t. the ASAP Default Hyper-heuristics as p . This way, we only would have used information which was also available during the design of the CheSC 2011 contestants. Also noteworthy here is that when F-/iRace is used with the Friedman tests (vs. Student t-test+ANOVA), it actually optimizes the scale-independent “average rank” objective (vs. average-case performance), eliminating the need for manual normalization.

Reusability objective: As discussed in Section 5.2.1.1, the objective of our case study was to demonstrate PbC’s ability to design a reusable, off-the-shelf metaheuristics. However, this was no unmitigated success.

While our study in Section 5.3.2.4, showed that FS-ILS outperforms the state-of-the-art on the problem domains for which it was designed (X_{old}), it also revealed its shortcomings w.r.t. solving new problem domains (X_{new}). In particular, the manually designed AdapHH performed much better on X_{new} . That being said, we wish to stress that mediocre performance is mainly due to FS-ILS’s extremely poor performance on the KP domain. On the other two domains, it was competitive. In particular, a variant without restart mechanism (NR-FS-ILS) outperformed AdapHH on QAP. Also, it is worth noting that NR-FS-ILS is much simpler than AdapHH, whose implementation counts $14\times$ more lines of code.

Lacking relevant analytical information, we are forced to use empirical methods in the design of heuristics (see Section 3.4). A major challenge in designing “reusable” heuristics,

in particular, is gathering a sufficiently large, heterogeneous set of training instances. In Section 3.5.3, we suggested a semi-online approach which addresses this problem by constructing the training set incrementally, a posteriori. It is our belief that the 68 instances and six domains offered by HyFlex are simply insufficiently representative of the possible variety our selection hyper-heuristic could be presented with. Furthermore, we believe that the three new domains are indeed quite different from the original six. For instance, while ANW-HH performs better than AA-HH on X_{old} , it performs decisively worse on all new domains. One way to increase diversity, using HyFlex, would be to consider artificial domains restricting the heuristic set ($H' \subset H$). Also, as a bonus, this would induce a bias towards selection hyper-heuristics that are less sensitive to the choice of H .

5.5 Summary

In this chapter, we presented a case study in which we apply a semi-automated design approach, which we refer to as Programming by Configuration (PbC), to design a reusable metaheuristic optimization framework. Subsequently, we discussed the resulting design, which we refer to as Fair Share Iterated Local Search (FS-ILS), and presented an extensive empirical analysis thereof. This research has been published in the form of three independent research articles: [Adriaensen et al. 2014a] (Sections 5.1-5.2), [Adriaensen et al. 2014b] (Sections 5.3.1, 5.3.2.1-5.3.2.3) and [Adriaensen et al. 2015] (Section 5.3.2.4). In what follows, we briefly summarize the content covered in each section.

In Section 5.1, we introduced the semi-automated design approach we used in our case study, and discussed its relation to similar paradigms proposed in prior art. In the spirit of consolidation, we have decided to adopt the terminology from [Hoos 2012] in this dissertation and will refer to this methodology as “Programming by Optimization” (PbO). Following this *modus operandi*, difficult decisions are deliberately left open at design time, programming a rich and potentially large design space, rather than a single algorithm. Subsequently, optimization methods are applied to automatically generate the best algorithm instance for a specific use case. We discussed that, thus far, optimizers of choice for PbO have been algorithm configurators, and we introduced the term “Programming by Configuration” (PbC) to refer to this specific realization of PbO. Finally, we defined the Algorithm Configuration Problem (ACP) and argued PbC to be a realization of the generic “design by set-ASP reduction” approach (see Section 4.1).

In Section 5.2, we described how we used PbC to design a reusable metaheuristic framework in [Adriaensen et al. 2014a]. Here, we first provided some context and motivation for our case study. In Section 5.2.1.1, we distinguished two approaches the metaheuristic community has taken to reduce the cost associated with applying metaheuristic frameworks to newly encountered problems: (1) off-the-shelf metaheuristics and (2) design automation;

and introduced the scenario considered in our case study as a logical combination of both approaches. In Section 5.2.1.2, we described HyFlex, a Java library providing problem-specific components for 6 different combinatorial optimization problems, which we used in the implementation and evaluation of the candidate designs considered in our case study. In PbC, we solve the ADP by reduction to the ACP. In Section 5.2.2, we discussed this reduction in general and for our case study in particular. Finally, in Section 5.2.3, we discussed solving the resulting ACP.

In Section 5.3 we had a closer look at the resulting design: FS-ILS. First, in Section 5.3.1, we discussed the design decisions made. FS-ILS is a simple, state-of-the-art, selection hyper-heuristic combining an iterated local search selection strategy with a conservative restart criterion. Each iteration, a non-greedy heuristic is selected proportionally to the acceptance rate of its previously proposed candidate solutions (after iterative improvement) by a domain-independent variant of the EMC criterion. Subsequently, in Section 5.3.2, we analyze FS-ILS experimentally. In Section 5.3.2.1, we performed a parameter sensitivity analysis, showing that FS-ILS is largely robust to changes in its sole numerical parameter. In Section 5.3.2.2, we analyzed the presence of accidental complexity, showing that FS-ILS performs significantly better than simpler variants, validating every design decision in the process. In Section 5.3.2.3, we compared its performance to that obtained by the 20 contestants during the CHeSC 2011 competition and show that FS-ILS would have won this competition decisively. Subsequently, we discussed the limitations of these kind of comparisons, which motivated the experiments performed in Section 5.3.2.4. Here, we conducted a comparative study of several publicly available selection hyper-heuristics, including FS-ILS and the CHeSC 2011 winner (AdapHH), in order to assess their generality across domains. To this end, we extended the HyFlex benchmark set with three additional problem domains: The 0-1 Knapsack, Quadratic Assignment and Max-Cut problem. In summary, we found that FS-ILS generalizes well to unseen (i.e. not used during its design) instances of the 6 original HyFlex domains. While FS-ILS is also competitive overall, it performed poorly on some of the instances of the new domains, the larger 0-1 Knapsack instances, in particular. Also, we found that on the full and extended HyFlex benchmark, FS-ILS does not seem to benefit from its restart mechanism (on the contrary), we can therefore omit it without loss of performance. Finally, while we found a naive uniform random selection hyper-heuristic to be surprisingly competitive on the new domains, overall, the state-of-the-art frameworks compared, clearly performed better than their naive counterparts. In particular, AdapHH clearly generalizes best to the new domains and performs most consistently overall.

Finally, in Section 5.4, we presented a critical reflection on our own research. Here, we discussed some unfortunate decisions we made during our case study, better alternatives, and whether it can be considered a “success”, or not.

6 | Towards a White Box Approach to Automated Algorithm Design

In Chapter 5, we have introduced and demonstrated Programming by Configuration (PbC), an emerging semi-automated design approach solving the Algorithm Design Problem (ADP) by reduction to the Algorithm Configuration Problem (ACP). In this chapter, we present a critical reflection on this successful and increasingly popular methodology, and investigate how we can do better (answer Q.A.3). Note that while our focus here is on PbC, our discussion is more widely applicable to the “design by per-set algorithm selection” approach (see Section 4.1). This chapter is outlined as follows: In Section 6.1, we discuss what we regard to be the main limitations of PbC, and suggest how these could be addressed. In particular, we argue, as in [Adriaensen and Nowé 2016b], that in reducing the ADP to an ACP, we abstract information that can be usefully exploited to solve it using less resources. In Section 6.2, we propose a more informative variant of the ACP, considered in [Adriaensen et al. 2017], capturing additional information about how parameter choices relate to algorithm performance. In Section 6.3, we show how this information can be used in estimating algorithm performance. In Section 6.4, we describe an optimizer implementing this novel performance estimation technique as a Proof of Concept. In Section 6.5, we validate this PoC experimentally, comparing its performance to that of various configurators, on a variety of ADPs. In Section 6.6, we present a critical reflection on this research. Finally, in Section 6.7, we summarize the content covered in each section.

6.1 Opinion: Limitations of PbC

The case study we performed in [Adriaensen et al. 2014a] (see Section 5.2), is just one of the many successful applications of PbC. The relative ease with which PbC can be applied to a wide variety of different scenarios arguably makes it one of the most “off-the-shelf” solutions to automated algorithm design thus far (see also Section 4.4.2). However, it also has shortcomings. In this section, we present a high-level discussion of what we regard to be the main limitations of PbC, and suggest how these could be addressed.

It is computationally expensive. Arguably the main limitation of PbC is that it requires a large amount of computational resources. For instance, in [Adriaensen et al. 2014a], we allocated roughly one CPU year (!) to the design of FS-ILS (distributed over 30 parallel design processes, this took less than two weeks). While arguably an extreme example, budgets of multiple CPU days are not unusual (e.g. 20 days [KhudaBukhsh et al. 2009], 150 days [Fawcett et al. 2009]). Remark that this limitation is not “PbC specific”, and to some extent inherent to simulation-based approaches (see Section 3.4.2), i.e. gathering experience through experimentation takes time. However, in the remainder of this section we will discuss some attributes of PbC that further aggravate this issue.

It is an offline approach. Contemporary realizations of PbC solve the ADP offline, i.e. “designing the algorithm” precedes “using it”. We already discussed the limitations of offline design in Section 3.5.1. Here, we briefly summarize them and discuss their relevance in the context of designing reusable algorithms.

In terms of economics, the offline approach requires us to pay the full computational cost “up front”. Remark that the reusability of the resulting design allows us to amortize this cost over its longer lifetime. Another downside is that it may be difficult to predict a priori what kind of problem instances a design will be used for (\mathcal{D}). This is particularly relevant if we consider the design of reusable algorithms, i.e. for any possible use case vs. a specific use case, as the possible variety of inputs such design could, in principle, be presented with is vast. Finally, more information becomes available while the algorithm is being used (e.g. instances it is being used for and its performance thereon) which offline approaches fail to exploit. In addition, if \mathcal{D} changes over time, the design is not adapted accordingly. Remark that “adaptation to context changes” is arguably less of an issue for reusable algorithms, since they are designed to work well in all contexts.

In Section 3.5.2, we discussed online design, i.e. refining the design “while it is being used”, as an alternative addressing these limitations. Most configurators could be used in an online (cross-input) setting without any major adaptations. Let x_i be the i^{th} problem instance to be solved. Let θ^i be the i^{th} configuration evaluated by the configurator. One could simply apply the configurator to configure a target algorithm a for a single input x_0 and choose $p.\text{eval}$ such that the evaluation of a_{θ^i} solves x_i . To further improve

online performance of anytime configurators one could (gradually) decrease exploration by evaluating the incumbent (increasingly) more frequently. This brings us to main challenge associated with the online setting: the inherent exploration vs. exploitation trade-off. In Section 3.5.3, we have argued that in many practical settings this trade-off can be avoided entirely by exclusively allocating cheap/spare resources to exploration (\sim semi-online approach). Furthermore, we have shown that every anytime design procedure (e.g. configurators) can be transformed into a semi-online approach.

In summary, while contemporary applications of PbC consider the offline setting, configurators can be easily ported to semi-online or even true online settings. Do note that while these transformations enable “cross-input” learning, they do not enable “within-run” learning. The ability to evaluate multiple designs over the course of a single run is crucial for enabling the design of algorithms which require a long time (e.g. multiple days) to execute. That being said, whether and how this can be done, in a general and scalable manner, is still largely an open problem.

It is a black box approach. In [Adriaensen and Nowé 2016b] we discussed another limitation of PbC. Here, we argued that, in reducing the ADP to an ACP, we abstract information which could otherwise be exploited to solve it faster. In particular, as discussed in Section 4.1.1, in the set-ASP, the performance of a configuration θ on a given input x , i.e. $p(x, \theta)$, is treated as a black box function mapping an input and a configuration to some real cost value. The ACP does not capture the fact that this mapping is a consequence of algorithm execution, i.e. a configuration affects the execution of the target algorithm on an input in a particular way, which in turn relates to execution cost. We conjecture that information about the computation of $p.\text{eval}$ can be usefully exploited in solving the ADP.

Our belief is founded in the observation that the human designer also uses this kind of information in his/her trial & error experimentation. That is, if an algorithm does not perform as expected, he/she will often not restrict him/herself to changing it and observing the impact of these changes on performance. Instead, he will trace the execution, observe what values variables take at runtime (through simple logging or using debug tools), to gain some insights into what behavior *caused* this observation. This kind of inspection leads to insights which may be generalizable across multiple designs and enables a more directed exploration of the design space. Our goal, in a sense, is to also allow a computer to use this kind of information in the design (and analysis) of algorithms.

Motivating example - Unused parameters: Let us consider a simple example which has motivated our research. Recall that sometimes not all parameters of an algorithm framework are used. Clearly, the values of parameters which were not used, cannot be held responsible for the performance we observe. Put differently, the performance observed can be generalized across all configurations differing only in the values of these unused parameters. Also, we can focus on varying the subset of parameters that were actually

used. Ordinary configurators are clueless about whether a given parameter was used or not, and therefore cannot exploit this information. Remark that some configurators do allow the user to specify conditional parameters a priori. However, this is only possible if these are run-independent, i.e. unused for every run (on any input) using a given configuration.

6.2 A White Box Formulation

As discussed in the previous section, we would like to use additional runtime information to reduce the resources required to solve a given ADP. We address this challenge in two steps: First, in this section, we will expose this information to our solver, i.e. present a formulation of the ADP which (unlike the ACP) captures this information. In subsequent sections, we will discuss how we can exploit this information to solve the ADP faster.

6.2.1 How about the Dynamic Algorithm Selection Problem?

Remark that in Section 4.3.1, we already introduced a formulation, i.e. the dynamic ASP, which captures all this information. We also discussed traditional (value-based) Reinforcement Learning (RL) methods as a general solution technique capable of exploiting (some of) this information (see Section 4.3.2.2). However, we also found these methods to suffer from scalability issues (see Section 4.3.2.3), which severely limit their applicability to large and challenging real-world ADPs. While hybridization with traditional Machine Learning (ML) techniques (e.g. value function approximation) allows us to address these scalability issues, general, off-the-shelf ML techniques do not exist. After considerable experimentation with these methods, it is our opinion that while of definite interest for solving specific instances of the ADP, especially if the designer happens to be a RL/ML expert, current RL and ML methods do not permit a level of generality and control similar to that offered by contemporary configurators (see also Section 4.4.2). That being said, the aforementioned shortcomings have prompted the RL community to investigate alternative solution techniques, collectively referred to as (direct) policy search methods (see p. 145). However, despite being classified as “RL methods”, these techniques have far more in common with traditional black box optimization techniques, and do often not exploit additional runtime information either. In fact, configurators themselves are naturally viewed as direct policy search methods. Nonetheless, there are exceptions, and it is our belief that certain concepts used in the RL community can be used to at least partially exploit white box information, while retaining the properties that made PbC attractive in the first place. In [Adriaensen et al. 2017] we investigated the potential of one of these techniques: “off-policy policy evaluation using importance sampling”, which we will present in subsequent sections. In the remainder of this section, we will present a variant of the ACP which only captures the information required by this technique.

6.2.2 A White Box Variant of the Algorithm Configuration Problem

In this section, we will present the formulation of the ADP we considered in [Adriaensen et al. 2017], which is a more informative variant of the ACP.

Definition 6.1: White Box Algorithm Configuration Problem (wb-ACP) [Adriaensen et al. 2017]

Instances of the wb-ACP are defined by quintuples $\langle a, \Theta, \mathcal{D}, \text{pr}, p \rangle$ where

a : an algorithm framework (see Section 2.3.3) having k configurable parameters (a.k.a. the *target algorithm*), where the i^{th} parameter can take values in Θ_i .

$\Theta \subseteq \Theta_1 \times \dots \times \Theta_k$ a set of possible configurations for a .

\mathcal{D} : a distribution over a set of inputs X (*input distribution*).

pr : a function $\Theta \times E \rightarrow \mathbb{R}$, where $\text{pr}(\theta, e)$, also denoted $\text{pr}(e|\theta)$, specifies the likelihood that executing a_θ , on an input $x \sim \mathcal{D}$, results in an execution e .

p : a function $X \times E \rightarrow \mathbb{R}$, quantifying the desirability of an execution.

Candidate solutions are elements of Θ . In the wb-ACP, given $\langle a, \Theta, \mathcal{D}, \text{pr}, p \rangle$, we are to find a configuration θ^* maximizing the average-case performance of a , i.e. satisfying $o(\theta) \leq o(\theta^*)$, $\forall \theta \in \Theta$, where $o(\theta) = \sum_{e \in E} \text{pr}(e|\theta)p(e)$ and E (*execution space*) is the set of all possible executions of a for any $\theta' \in \Theta$, on any $x \in X$.

Conceptually, every configuration can be viewed as a distribution over executions, whose performance corresponds to the expectation of p , over this distribution (see Figure 6.1). In the wb-ACP, we are to find the configuration whose distribution maximizes $\mathbf{E}[p]$.

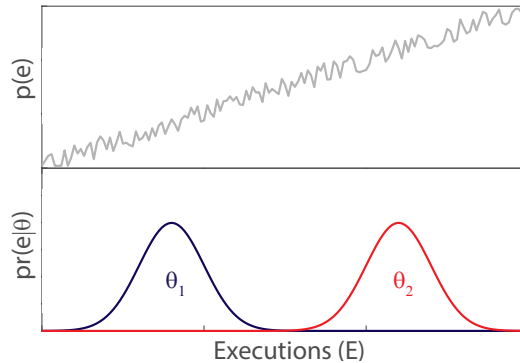


Figure 6.1: Distribution perspective

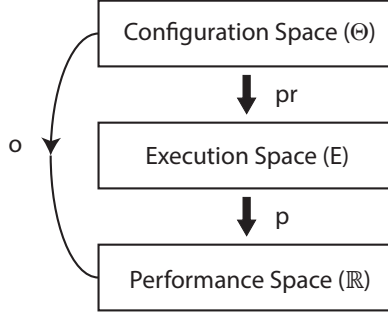


Figure 6.2: Diagram showing how pr and p relate designs to their performance.

Our formulation above is closely related to the ACP as we defined it in Definition 5.1. In particular, as we will show in Section 6.2.3, it is m -equivalent to the ACP. The most relevant difference, in the context of this chapter, lies in our definition of o . In the ACP, we are given a performance evaluation procedure ($p.\text{eval}$), mapping (an input and) a configuration to a real-valued performance observation, which is itself treated as a (stochastic) black box. In the wb-ACP, we open up this black box by explicitly defining this mapping as a consequence of algorithm execution, i.e. our choice of configuration affects execution in a particular way (pr), which in turn relates to its observed desirability (p). Here, pr and p can be viewed as relating configurations with their performance, over execution space, as depicted in Figure 6.2. Remark that execution space can as such be viewed as decoupling configuration and performance space. A feature which we will exploit in Section 6.3 to generalize performance observations beyond the configuration used to obtain them.

Next, we will make some additional assumptions about the form in which instances of the wb-ACP, i.e. $\langle a, \Theta, \mathcal{D}, \text{pr}, p \rangle$, are presented to the solver:

a : The algorithm framework a is treated as a configurable black box as in the ACP. However, a crucial difference is that executing a_θ on an input x , does not only result in an output y , but also logs additional information about the execution e which serves as input to pr and p . Note that we do not require e to determine every aspect of the execution, but only those features required to compute pr and p .

Θ, \mathcal{D} are assumed to be given as in the ACP.

o : the average-case performance of a configuration is specified by functions p and pr .

p : A *deterministic* procedure computing the desirability of an execution e . This could be the (negated) time it took, or the quality of the solution it obtained.

pr specifies the distribution over executions corresponding to any given configuration. In the wb-ACP, we do not merely assume to be given a function which generates samples according to this distribution, but we want a procedure that computes the density of this distribution, i.e. $\text{pr}(e|\theta) = \sum_{x \in X} \mathcal{D}(x) \text{pr}(e|\theta, x)$. Remark that computing pr exactly requires X and \mathcal{D} to be known explicitly. Also, computing $\text{pr}(e|\theta, x)$, while theoretically possible (see Equation 2.1), requires full knowledge of a , including the likelihood of the possible outcomes of any stochastic events. All of this is information which is typically not readily available. Luckily, for the technique we will describe in Section 6.3, we do not require $\text{pr}(e|\theta)$ to be known exactly. Rather, we will assume to be given a deterministic procedure pr' which satisfies the following criteria:

$$\begin{aligned} \text{a)} \quad & \forall \theta \in \Theta, \forall e \in E : \text{sgn}(\text{pr}'(e|\theta)) = \text{sgn}(\text{pr}(e|\theta)). \\ \text{b)} \quad & \forall \theta, \theta' \in \Theta, \forall e \in E : \\ & \text{pr}(e|\theta) * \text{pr}(e|\theta') > 0 \implies \frac{\text{pr}'(e|\theta)}{\text{pr}'(e|\theta')} = \frac{\text{pr}(e|\theta)}{\text{pr}(e|\theta')}. \end{aligned} \tag{6.1}$$

Intuitively, $\text{pr}'(e|\theta)$ captures the “relative relevance” of $p(e)$ to θ . Clearly, $\text{pr}' = \text{pr}$ is a valid choice, but often functions exist that can be computed far more easily, yet satisfy (a) and (b): The actual $\text{pr}(e|\theta)$ can often be rewritten as the product of a configuration dependent factor F_{easy}^θ , which is easily computed, and a complex configuration independent factor F_{hard} . Since F_{hard} is configuration independent, this factor will be the same for every configuration and cancel out when considering ratios of likelihoods, allowing us to choose $\text{pr}'(e|\theta) = F_{\text{easy}}^\theta$ since

$$\frac{\text{pr}(e|\theta)}{\text{pr}(e|\theta')} = \frac{F_{\text{easy}}^\theta * F_{\text{hard}}}{F_{\text{easy}}^{\theta'} * F_{\text{hard}}} = \frac{F_{\text{easy}}^\theta}{F_{\text{easy}}^{\theta'}} = \frac{\text{pr}'(e|\theta)}{\text{pr}'(e|\theta')}$$

6.2.3 Relation to Other Formulations

In this section, we discuss the relations between the wb-ACP and previously examined formulations of the ADP: the ACP, subset-ASP and dynamic ASP. In particular, we describe possible generic (exact) reductions from these problems to the wb-ACP. We understand that formulating an ADP as a wb-ACP (e.g. choosing pr') may initially seem like a daunting task (if at all possible). Therefore, these reductions serve as readily applicable examples to convince our reader that this is not the case, but merely an artifact of the level of generality/abstraction we are operating at. For examples of specific instances of the wb-ACP reduction, we refer the reader to Section 6.5.

6.2.3.1 Algorithm Configuration Problem (ACP, see Definition 5.1)

wb-ACP \leq_m ACP: An instance of the wb-ACP $\langle a, \Theta, \mathcal{D}, \text{pr}', p \rangle$ can be formulated as an instance $\langle a', \Theta, \mathcal{D}, \text{pr}', p' \rangle$ of the (black box) ACP, where $p'.\text{eval}(x, \theta)$ is computed as follows:

1. Execute a_θ on x , resulting in y and e .
2. Compute and return $p(e)$.

Unsurprisingly, we can reduce the white box ACP to a black box ACP simply by abstracting/ignoring the additional information. Somewhat more surprisingly perhaps is that the reverse also holds.

ACP \leq_m wb-ACP: An instance of the ACP $\langle a, \Theta, \mathcal{D}, p \rangle$ can be formulated as an instance $\langle a', \Theta, \mathcal{D}, \text{pr}', p' \rangle$ of the wb-ACP, where

a' simulates a and logs $e = (\theta, r_x^\theta)$ with $r_x^\theta = p.\text{eval}(x, \theta)$, where θ is the configuration and x the input. Logging θ uniquely associates execution e with configuration θ .

$$\text{pr}'(e|\theta') = \text{pr}'((\theta, r_x^\theta)|\theta') = [\theta = \theta'].$$

$$p(e) = r_x^\theta.$$

This implies that white box configurators can be used to solve black box configuration problems. In a sense, while the wb-ACP can encode additional information when available, it is optional to do so. For example, in what follows, we consider an alternative, more informative reduction for a variant of the ACP where we can observe that some parameter values were not used during execution (cond-ACP). The most narrow definition of “using a parameter” would be “reading its value”. However, the reduction described below also works for the broader definition “its value affected the desirability of the execution”.

cond-ACP \leq_m wb-ACP: An instance of the cond-ACP $\langle a, \Theta, \mathcal{D}, p \rangle$, where a logs P_{used} the set of parameters which were (possibly) used, can be formulated as an instance $\langle a', \Theta, \mathcal{D}, \text{pr}', p' \rangle$ of the wb-ACP, where

a' simulates a and logs $e = (\bar{\theta}, r_x^\theta)$ with $\bar{\theta}_i = \theta_i$ ($\forall p_i \in P_{\text{used}}$) and $\bar{\theta}_i = *$ ($\forall p_i \notin P_{\text{used}}$) and $r_x^\theta = p.\text{eval}(x, \theta)$, where θ is the configuration and x the input. For instance, if $\theta = (6, 2, 8, 0)$ and $P_{\text{used}} = \{1, 3\}$ then $\bar{\theta} = (6, *, 8, *)$.

$$\text{pr}'(e|\theta') = \prod_{i=1}^k ([\bar{\theta}_i = *] + [\bar{\theta}_i = \theta'_i]).$$

$$p(e) = r_x^\theta.$$

6.2.3.2 Per-subset Algorithm Selection Problem (subset-ASP, see Definition 4.3)

subset-ASP \leq_m wb-ACP: Remark that this follows from subset-ASP \leq_m ACP¹ and ACP \leq_m wb-ACP. However, in what follows, we will consider a direct reduction which retains additional information about the relations between candidate selection mappings, which the ACP cannot capture.² More specifically, each instance of the subset-ASP $\langle A, \mathcal{D}, \phi, p \rangle$ can be formulated as an instance $\langle a', \Theta, \mathcal{D}, \text{pr}', p' \rangle$ of the wb-ACP, where

Θ contains exactly one configuration for every selection mapping of the form $\Phi \rightarrow A$.

We use s_θ to refer to the selection mapping corresponding to θ .

a' is chosen such that $a'(\theta)$ simulates the portfolio solver corresponding to s_θ and logs $e = (\phi_x, a, r_x^a)$, where ϕ_x are the features of input x (i.e. $\phi.\text{extract}(x)$), $a \in A$ is the selected algorithm and $r_x^a = p.\text{eval}(x, a)$ is its observed performance.

$$\text{pr}'(e|\theta) = [s_\theta(\phi_x) = a].$$

$$p(e) = r_x^a.$$

6.2.3.3 Dynamic Algorithm Selection Problem (dynamic ASP, see Definition 4.8)

dynamic ASP \leq_m wb-ACP Again, while this follows from dynamic ASP \leq_m ACP¹ and ACP \leq_m wb-ACP, we will present a direct reduction capturing relations between candidate policies, which the ACP cannot capture. More specifically, each instance of the dynamic ASP $\langle M, \mathcal{D}, \phi \rangle$ can be formulated as an instance $\langle a, \Theta, \mathcal{D}, \text{pr}', p \rangle$ of the wb-ACP, where

Θ contains exactly one configuration for every policy of the form

$$\Delta \times \Omega \times (Q \times \Gamma \times \{R, L\}) \rightarrow [0, 1].$$

We use π_θ to refer to the policy corresponding to θ .

a is chosen such that a_θ simulates both $\langle M, \phi \rangle$ and π_θ and logs for all choice points their identifiers (z), the decisions made by the agent (t) and the contexts in which they were encountered (ω); as well as r_Σ the sum of the rewards accumulated over the course of the execution, i.e. $e = (z, \omega, t, r_\Sigma)$.

$$\text{pr}'(e|\theta) = \prod_{i=1}^{|z|} \pi_\theta(z_i, \omega_i, t_i).$$

$$p(e) = r_\Sigma.$$

¹In Section 4.4.1, we have shown subset-/dynamic ASP \leq_m set-ASP. In Section 5.1, we argued the ACP to be a special case of the set-ASP using parametric representations for algorithm space A . Therefore, we have subset-/dynamic ASP \leq_m ACP, assuming every A can be represented parametrically.

²See Section 6.5.2.1 for an application of this reduction to the sorting subset-ASP (see Section 4.2.2).

6.3 Exploiting White Box Information

In this section, we discuss how the information captured by the wb-ACP can be used in evaluating the relative performance of candidate designs, and the benefits of doing so.

6.3.1 Algorithm Performance Evaluation

We already discussed the problem of evaluating algorithm performance in the context of solving the set-ASP (see Section 4.1.2). Here, we briefly recap this discussion and present a broader perspective.

In the (black box) performance evaluation problem, short PEP, we are given $\langle a, \Theta, \mathcal{D}, p \rangle$ and we would like to know the performance $o(\theta) = \mathbf{E}_{x \sim \mathcal{D}} p(x, \theta)$, $\forall \theta \in \Theta$. Note that in the PEP we are interested in the performance of multiple different configurations (\sim family of related algorithms), rather than that of any particular configuration (\sim algorithm instance). The PEP occurs naturally in

Analysis settings, where we would like to compare the performance of different algorithms.

For instance, parameter sensitivity analysis, where we are interested in the influence of an algorithm's parameter setting on its performance.

Design settings, where we are interested in finding the best algorithm for our target problem. For example, in algorithm configuration where we search a space of alternative configurations for one optimizing performance.

While our focus, in this dissertation, was on the latter, we wish to stress that all the techniques we will discuss here are equally relevant in analysis settings. In the design setting, the PEP is typically solved “online”, i.e. while solving the ADP. Here, we distinguish between two uses of performance evaluations:

1. Determine which of the designs evaluated thus far is best (*choice of incumbent*), i.e. is to be returned as the solution at any time.
2. Determine which design to evaluate next (*guide the search*), e.g. to identify unexplored, potentially interesting areas in the design space.

Remark that the PEP cannot be solved exactly, in general, as \mathcal{D} and p are not known explicitly, but rather are assumed to be given in the form of randomized procedures $\mathcal{D}.\text{sample}$ and $p.\text{eval}$ respectively, satisfying $o(\theta) = \mathbf{E}[p.\text{eval}(\mathcal{D}.\text{sample}(), \theta)]$. Therefore, in practice, we often have to rely on estimates of performance \tilde{o} instead, based on performance observations obtained by executing $p.\text{eval}$ on inputs generated using $\mathcal{D}.\text{sample}$. Furthermore, as executing $p.\text{eval}$ involves executing the target algorithm a , generating these

observations is often highly costly and dominates the overall time the design process takes. In the next section, we will present a technique targeted at reducing the number of evaluations required to get reliable performance estimates. In the remainder of this section, we will discuss how algorithm performance is currently being estimated (in PbC, in particular). Prior art in performance estimation can be roughly subdivided into two categories:

Independent sample averages: This is the predominant approach, used e.g. in ParamLLS [Hutter et al. 2009], iRace [López-Ibáñez et al. 2011], GGA [Ansótegui et al. 2009], and the one we discussed elaborately in Section 4.1.2. In a nutshell: We collect a sample of performance observations R_θ for each configuration θ by repeatedly evaluating it on inputs drawn from \mathcal{D} . The performance of each configuration is then estimated as the average of the performances observed for that configuration (i.e. $\tilde{o}(c) = \bar{o}(c)$, see Equation 4.2).

Regression model predictions: The second is the “model-based” approach. This approach is for instance used in Sequential Model-Based Optimization (SMBO) frameworks (e.g. SMAC [Hutter et al. 2011], GGA++[Ansótegui et al. 2015]). Here, rather than maintaining an independent estimate for each configuration, we maintain an explicit regression model $M : \Theta \rightarrow \mathbb{R}$, trained using all previous performance observations, and use this model’s predictions as an estimate for performance, i.e. $\tilde{o}(\theta) = M(\theta)$. Clearly, this approach is more data efficient than the first, as it generalizes performance observations across the configuration used to obtain it. Note that (in PbC) these predictions are, to the best of our knowledge, only used to guide the search (2). The choice of incumbent (1) is based solely on performance observations obtained using the design itself.

6.3.2 An Importance Sampling Approach

In this section, we present a novel way of estimating algorithm performance in the context of solving the wb-ACP, targeted at reducing the number of evaluations needed, i.e. solving it faster. First, in Section 6.3.2.1, we introduce the observation underlying our approach: Different candidate designs might, with some likelihood, generate the exact same execution. As such, the observed desirability of a *single* execution provides information about the performance of *multiple* different designs. Subsequently, in Section 6.3.2.2, we introduce the general statistical technique known as Importance Sampling (IS) and show how IS can be used to combine all performance observations relevant to a design, into a consistent estimate of its performance, in Section 6.3.2.3. Next, in Section 6.3.2.4, we present an effective procedure for computing this estimate. Finally, in Section 6.3.2.5, we discuss the potential benefits of this approach and illustrate some of these experimentally.

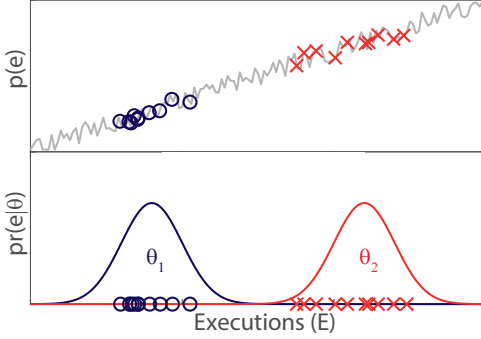


Figure 6.3: Two different designs

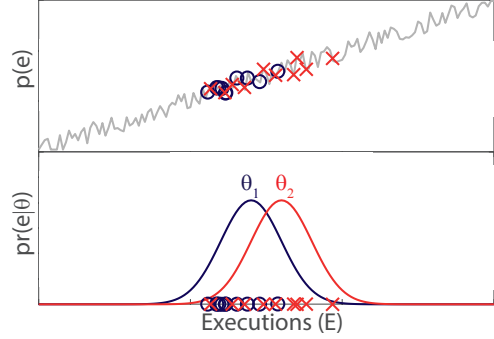


Figure 6.4: Two similar designs

6.3.2.1 Motivation: Similarities in Execution Space

In the wb-ACP, we are faced with a white box variant of the PEP, i.e. given $\langle a, \Theta, \text{pr}, p \rangle$, we are to find $o(\theta) = \mathbb{E}_{e \sim \theta} p(e), \forall \theta \in \Theta$ (wb-PEP). Using the wb-ACP \leq_m ACP reduction described in Section 6.2.3.1, any of the aforementioned (black box) performance evaluation techniques can also be applied to the wb-PEP. In this context, the independent sample average approach can be viewed as drawing a sample of executions E'_θ for each configuration $\theta \in \Theta$ according to its corresponding distribution, and estimating its performance as the average of the associated performance observations, i.e. Equation 4.2 can be rewritten as

$$\bar{o}(\theta) = \frac{1}{|E'_\theta|} \sum_{e \in E'_\theta} p(e) \text{ with } E'_\theta \sim \theta. \quad (6.2)$$

Figures 6.3 and 6.4 illustrate this, where the blue o's and red x's represent the sample of executions (and associated performance observations) for θ_1 and θ_2 , respectively. Note that sample average estimates are fully independent, i.e. $\bar{o}(\theta_1)$ does not depend on observations made using θ_2 (red x's) and vice versa. In what follows, we will argue that while this independence is reasonable when the distributions in the design space are disjoint, as in Figure 6.3, we can do better if two or more distributions overlap, as in Figure 6.4, i.e. if there exists an execution that can be generated by at least two different candidate designs:

$$\exists e \in E, \exists \theta_i, \theta_j \in \Theta : \theta_i \neq \theta_j \wedge \text{pr}(e|\theta_i) * \text{pr}(e|\theta_j) > 0. \quad (6.3)$$

The crux is that even though an execution e might have been obtained using θ_i , it might as well, with some likelihood, be generated using a different θ_j . Put differently, when distributions overlap, evaluations of one design θ_i , potentially provide information about the performance of another θ_j . As these evaluations can be extremely costly, we would like

to maximally utilize the information they provide; e.g. also use the red \times 's to estimate the expectation over the blue distribution and vice versa. In the next section, we will introduce a technique which allows us to do exactly that.

Digression: Does overlap actually occur in practice? Here, it is important to distinguish between overlap in the actual ADP and overlap in our wb-ACP formulation thereof. The assumptions we made about E in Section 6.2 (p. 196) are flexible enough that Equation 6.3 may be satisfied, even if no two candidate designs generate the exact same sequence of instructions, and vice versa. Therefore, rather than whether or not overlap occurs in actual ADPs, we are interested in whether it is possible (and useful?) to formulate any (interesting?) ADP as a wb-ACP where (significant) overlap does occur. In what follows, we will discuss whether and when overlap occurs (Equation 6.3 is satisfied) for the reductions described in Section 5.2.1.1:

ACP \leq_m wb-ACP: Here, we chose $\text{pr}'(e|\theta') = [\theta = \theta']$ with $e = (\theta, r_x^\theta)$, where θ was the configuration used to generate e and r_x^θ the associated performance observation. We can easily show that under this reduction Equation 6.3 can never be satisfied.

cond-ACP \leq_m wb-ACP: Here, we chose $\text{pr}'(e|\theta') = \prod_{i=1}^k ([\bar{\theta}_i = *] + [\bar{\theta}_i = \theta'_i])$ with $e = (\bar{\theta}, r_x^{\bar{\theta}})$, where $\bar{\theta}$ is the “active” configuration, replacing unused parameter values in θ with wildcards: *. Here, an execution e can be generated (with equal positive likelihood) by all configurations differing only in the values chosen for the parameters $p_i : \bar{\theta}_i = *$. Overlap occurs if and only if occasionally not all parameters are used.

subset-ASP \leq_m wb-ACP: Here, we chose $\text{pr}'(e|\theta) = [s_\theta(\phi_x) = a]$, with $e = (\phi_x, a, r_x^a)$, where ϕ_x are the features of the input, a the algorithm which was selected from the portfolio, and r_x^a the performance observation associated with its execution. Here, an execution e can be generated (with equal positive likelihood) by all configurations whose selection mapping selects the same algorithm a for ϕ_x . This implies that every execution e can be generated by exactly $\frac{1}{|A|}$ th of the selection mappings. Overlap occurs if and only if we have more than $|A|$ candidate selection mappings.

dynamic-ASP \leq_m wb-ACP: Here, we chose $\text{pr}'(e|\theta) = \prod_{i=1}^{|z|} \pi_\theta(z_i, \omega_i, t_i)$ with $e = (z, \omega, t, r_\Sigma)$, where t_i is the decision made for the i^{th} choice point z_i encountered in context ω_i and r_Σ is the total sum of rewards accumulated during execution. An execution e can be generated (possibly with different likelihoods) by all configurations whose corresponding policies have a non-zero likelihood of making decision t_i for choice point z_i in context ω_i , for $1 \leq i \leq |z|$. In particular, configurations corresponding to non-degenerate policies, i.e. satisfying $\pi_\theta > 0$, can generate all possible executions, i.e. have $\text{pr}(e|\theta) > 0, \forall e \in E$. Note that there are uncountably infinite times more non-degenerate than degenerate policies.

6.3.2.2 Importance Sampling (IS)

Importance sampling is a general technique for estimating properties of one distribution \mathcal{P} , using samples generated by another \mathcal{Q} ([Rubinstein and Kroese 2016, Section 5.7], [Hesterberg 1988]). By sampling according to \mathcal{Q} , we will oversample some outcomes, while undersampling others. Concretely, we will observe an outcome on average $\frac{q}{p}$ times more often than one would expect under the nominal distribution \mathcal{P} , where p and q are the probability density functions of \mathcal{P} and \mathcal{Q} , respectively. As such, we have to adjust our estimate to account for this, which is done by weighing observations accordingly.

We were unable to determine the exact origin of the technique, but one of the earliest mentions thereof, in published research, by this name, can be found in [Kahn and Marshall 1953]. What is certain is that IS, in one form or another, is used frequently and in a wide variety of applications. In what follows, we will consider the use of IS in the classical setting of estimating the expected value of a random variable with known distribution \mathcal{P} , i.e. $\mathbf{E}_{x \sim \mathcal{P}} f(x)$. Given procedures for computing f , p , q , and for generating (i.i.d.) samples according to \mathcal{Q} :

1) Collect a sample $\Omega' \sim \mathcal{Q}$ (i.i.d.).

2) Estimate $\mathbf{E}_{x \sim \mathcal{P}} f(x)$ as $\frac{1}{|\Omega'|} \sum_{x \in \Omega'} \frac{p(x)}{q(x)} f(x)$. (6.4)

This process produces an unbiased estimate, i.e. $\mathbf{E}_{x \sim \mathcal{Q}} \frac{1}{|\Omega'|} \sum_{x \in \Omega'} \frac{p(x)}{q(x)} f(x) = \mathbf{E}_{x \sim \mathcal{P}} f(x)$, if $p(x) > 0 \implies q(x) > 0$. Conceptually, it suffices that \mathcal{Q} can generate any outcomes possible under \mathcal{P} . Remark that there is no single IS estimate and many variants of Equation 6.4 exist. See [Hesterberg 1988] for some examples and an analysis of their properties.

6.3.2.3 An IS Estimate of Algorithm Performance

Having introduced the general technique, we can now turn towards how it can be used to solve the wb-PEP. In particular, we show how IS can be used to combine all performance observations $p(e)$ relevant for a design into a consistent estimator of its performance.

To this end, let us first consider the simpler case of estimating the performance of one configuration θ_1 using the performance observed when evaluating another θ_2 ; e.g. in Figure 6.4 use the **red** **x**'s to estimate the expectation over the **blue** distribution. Substituting $\mathcal{P} = \theta_1$, $\mathcal{Q} = \theta_2$, $f = p$, $\Omega' = E'_{\theta_2}$, Equation 6.4 becomes

$$\frac{1}{|E'_{\theta_2}|} \sum_{e \in E'_{\theta_2}} \frac{\text{pr}(e|\theta_1)}{\text{pr}(e|\theta_2)} p(e) \text{ with } E'_{\theta_2} \sim \theta_2. \quad (6.5)$$

This estimate is unbiased if $\text{pr}(e|\theta_1) > 0 \implies \text{pr}(e|\theta_2) > 0$. As a concrete example, Equation 6.5 could be used to estimate the performance of a simulated annealing framework (see Section 2.7.2.2) using an arbitrary temperature parameter T (θ_1), based solely on observations obtained using some $T' \in]0, +\infty[$ (θ_2). Remark that $T' = 0$ is excluded as it cannot generate executions accepting worsening proposals.

Clearly, if available, we would also like to incorporate the observations made using θ_1 itself; e.g. use the blue o's as well. More generally, we would like to use all (relevant) observations, independent of which configurations were used to obtain them. Put differently, we wish to estimate a property of \mathcal{P} based on samples generated using multiple different Q 's. Doing so is known as “multiple importance sampling” [Veatch 1997, Chapter 9]. As shown in [Veatch 1997], there are many possible ways of combining the observations that result in consistent (and unbiased) estimates. We will focus on a single one, which [Veatch 1997] refers to as the “balance heuristic”. Let $E' = \bigcup_{\theta' \in \Theta'} E'_{\theta'}$ be the sample of all executions, generated using $\theta' \in \Theta'$ on $x \sim \mathcal{D}$, we can estimate the performance of any $\theta \in \Theta$ as

$$\hat{o}(\theta) = \frac{1}{|E'|} \sum_{e \in E'} w_{\theta}(e) p(e) \quad \text{with} \quad (6.6)$$

$$w_{\theta}(e) = \frac{\text{pr}(e|\theta)}{Q'(e)} \quad \text{and} \quad Q'(e) = \sum_{\theta' \in \Theta'} \frac{|E'_{\theta'}|}{|E'|} \text{pr}(e|\theta')$$

Conceptually, this is the ordinary IS estimate, where we treat E' as if it were generated by a distribution Q' which is a mixture of $\theta' \in \Theta'$, where the weight of θ' in the mixture is given by $\frac{|E'_{\theta'}|}{|E'|}$. Figure 6.5 depicts Q' for our running example where $|E'_{\theta_1}| = |E'_{\theta_2}| = 10$.

In Appendix A.3, we show that $\hat{o}(\theta)$, as in Equation 6.6, indeed combines all observations

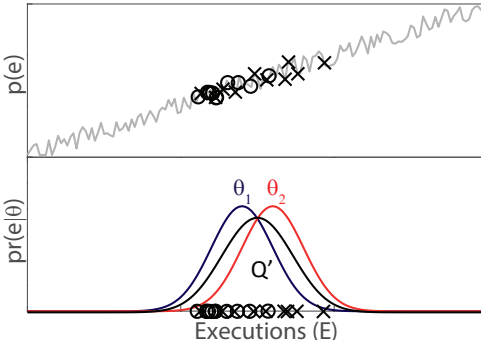
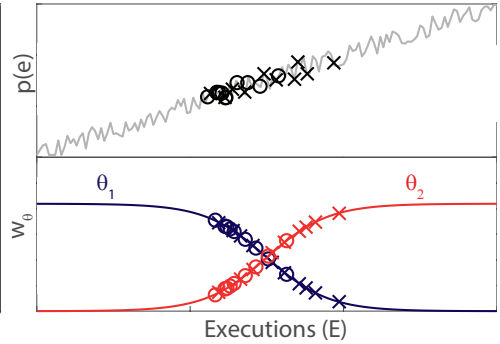
Figure 6.5: Q' as an equal mixture.

Figure 6.6: Design-specific weight functions

relevant to θ (see Theorem A.1) into an unbiased and consistent estimate of its performance, i.e. $o(\theta)$ (see Lemma A.2 and Theorem A.3). We also show that in settings where distributions do not overlap, \hat{o} reduces to an ordinary sample average \bar{o} (see Theorem A.4).

Remark that the performance estimate of each design θ is based on the same sample of (all) executions E' . Obviously, not all of these say as much about the performance of one design than they do about that of another. In IS, we therefore weigh observations according to their relative relevance, using a design dependent weight function w_θ . For instance, in our running example, observations on the left side are more relevant for θ_1 , the ones on the right are more relevant for θ_2 , something which is clearly reflected in their respective weight functions (see Figure 6.6).

Remark that, for any finite E' , the average weight is typically not exactly equal to one. Therefore, it is common practice, sometimes referred to as “weighted” IS [Mahmood et al. 2014], to normalize the weights in an attempt to further reduce the variance; e.g.

$$\hat{o}(\theta) = \frac{\sum_{e \in E'} w_\theta(e) p(e)}{\sum_{e \in E'} w_\theta(e)}. \quad (6.7)$$

In [Hesterberg 1988], the above variant is also referred to as the “ratio” estimate, while the original one listed in Equation 6.4 is called the “integral” estimate. Remark that unlike Equation 6.6, Equation 6.7 represents a biased estimate, i.e. Lemma A.2 does no longer hold. In the remainder of this chapter, we will nonetheless be using the normalized variant, as we did in [Adriaensen et al. 2017]. All other theoretical results we present in Appendix A.3 can be generalized to this normalized variant. In particular, $\hat{o}(\theta)$ is still consistent under the same assumptions we made in Theorem A.3.

6.3.2.4 Computation of an IS estimate

Computability: Something we did not discuss thus far is whether we can actually compute the IS estimate, in practice, in the context of the wb-ACP? Doing so using Equation 6.7 requires us to be able to compute $w_\theta(e), \forall e \in E', \forall \theta \in \Theta$. The latter could be done as in Equation 6.6 which would in turn require us to compute pr . As we discussed in Section 6.2.2, p. 197, pr is typically not known exactly and rather we assumed to be given a function pr' satisfying the criteria in Equation 6.1. As it turns out (see Theorem A.5), we can compute w_θ (and therefore $\hat{o}(\theta)$) simply by using pr' instead of pr . Algorithm 20 lists pseudocode for computing \hat{o} as in Equation 6.7.

Computational Complexity: Obviously, the theoretical possibility of computing \hat{o} , does not imply that it is also tractable in practice. In what follows, we will discuss the computational complexity (see Section 2.5) of our approach. Algorithm 20 has an asymptotic time complexity of $\mathcal{O}(|E'|^2)$ and space complexity of $\mathcal{O}(|E'|)$. In contrast, an ordinary

Algorithm 20 Procedure computing $\hat{o}(\theta)$ (as in Equation 6.7) given a sample of executions E' obtained using configurations Θ' on $x \sim \mathcal{D}$.

```

1: procedure ESTIMATEO( $\theta, \text{pr}', p, \Theta', E' = \bigcup_{\theta' \in \Theta'} E'_{\theta'}$ )
2:    $w_\Sigma, \hat{o}_\theta \leftarrow 0$ 
3:   for  $e \in E'$  do
4:      $Q'_e \leftarrow 0$ 
5:     for  $\theta' \in \Theta'$  do
6:        $Q'_e \leftarrow Q'_e + \frac{|E'_{\theta'}|}{|E'|} \text{pr}'(e|\theta')$ 
7:     end for
8:      $w_e \leftarrow \frac{\text{pr}'(e|\theta)}{Q'_e}$ 
9:      $w_\Sigma \leftarrow w_\Sigma + w_e$ 
10:     $\hat{o}_\theta \leftarrow \hat{o}_\theta + w_e * p(e)$ 
11:   end for
12:   return  $\frac{\hat{o}_\theta}{w_\Sigma}$ 
13: end procedure

```

sample average \bar{o} can be computed in space/time $\mathcal{O}(|E'|)$. Furthermore, \bar{o} can be computed online: Every time we observe a new e we can update \bar{o} in $\mathcal{O}(1)$ time. Computing \hat{o} online is not as straightforward. As every observation possibly changes $Q'(e), \forall e \in E'$, we would naively need to recompute \hat{o} from scratch every time E' changes. One optimization we implemented involves memorizing $Q'(e)$ between calls to Algorithm 20. While the space complexity remains $\mathcal{O}(|E'|)$, since we can update $Q'(e)$ after every run in $\mathcal{O}(|E'|)$ time, we can update \hat{o} online in $\mathcal{O}(|E'|)$ time as opposed to $\mathcal{O}(|E'|^2)$ time. Nonetheless, asymptotically, it will take $|E'|$ times longer to compute \hat{o} than \bar{o} . This dependency on $|E'|$ is undesirable and stems from the fact that we use the “balance heuristic” [Veach 1997] to combine observations generated using different configurations. As mentioned above, many alternatives exist, some of which may have a considerably lower computational complexity.

A hidden constant is the cost of computing pr' , which may be significant in some cases. Since we repeatedly compute pr' for the same inputs one possible optimization would be to memoize pr' . Also, we could further reduce overhead by computing pr' in batch/parallel.

In summary, using \hat{o} will be computationally more expensive than using \bar{o} , increasing the resources required to execute the optimizer's code. However, recall that contemporary configurators typically spend the vast majority of their time evaluating configurations (executing the target algorithm's code) and it is our hope that the use of \hat{o} in place of \bar{o} will allow us to significantly reduce the number of evaluations in settings where Equation 6.3 is satisfied. Therefore, in these settings, assuming the cost of computing \hat{o} is reasonable compared to the cost of an evaluation, potential performance gains may nonetheless be considerable. In other settings, we believe that automatically switching estimates accordingly would be relatively straightforward.

6.3.2.5 Envisioned Benefits of IS Approach

In this section, we will discuss the benefits we envision to be associated with using IS, as opposed to well-established alternatives such as independent sample averages and regression models (see Section 6.3.1). Here, we will illustrate some of these benefits experimentally for the abstract setup shown in Figures 6.1-6.6, where $\langle a, \Theta, \mathcal{D}, \text{pr}, p \rangle$:

a : A framework taking as input an integer x and a single real-valued parameter θ . It uses x to seed a random generator, which is used to generate two samples $y \sim \mathcal{N}(\theta, 1)$ and $z \sim \mathcal{U}(-1, 1)$, before returning $e = (y, z)$.

$\Theta = \{\theta_1, \theta_2\}$: We consider two candidate values for the single parameter.

$\mathcal{D} = \mathcal{U}(\mathbb{Z})$: A uniform distribution over possible integer-valued seeds.

$\text{pr}'(e|\theta) = \phi(y - \theta)$, where ϕ is the density function of the standard normal distribution.

$p(e) = y + z$.

Remark that we have $o(\theta) = \mathbb{E}_{e \sim a(\theta)} p(e) = \theta$ and the best configuration will simply be $\max(\theta_1, \theta_2)$. We will consider the problem of estimating o based on a sample of executions $E' = E'_{\theta_1} \cup E'_{\theta_2}$ with $|E'_{\theta_1}| = |E'_{\theta_2}|$ ($\sim Q'$ in Figure 6.5). We will consider different instances of this problem based on the difference between the two configurations, i.e. $\Delta = \theta_2 - \theta_1$. While Equation 6.3 is satisfied for any Δ , overlap will be larger for smaller Δ values; e.g. Figure 6.3 and Figure 6.4 illustrate different instances of this problem, with Δ equal to 8 and 1, respectively. Obviously, this particular instance is not representative for the wb-ACP's we will encounter in practice. Nonetheless, we would like to argue that the observations in this section can be generalized.

Data efficient performance estimation: $\bar{o}(\theta)$ only uses the observations which were obtained using θ , i.e. E'_θ , to estimate $o(\theta)$. In contrast, by Theorem A.1, $\hat{o}(\theta)$ uses all observations $e \in E'$ relevant for θ , i.e. $\{e \in E' \mid \text{pr}(e|\theta) > 0\}$. In using a single performance observation in the estimation of many designs, we amortize the cost of obtaining it, and will hopefully need less performance evaluations to obtain similarly accurate estimates. Figure 6.7 illustrates this, comparing the average³ estimation errors $|\bar{o}(\theta_2) - o(\theta_2)|$ (dashed line) and $|\hat{o}(\theta_2) - o(\theta_2)|$ (full lines), after x evaluations, for multiple setups with different Δ . Clearly, if Δ is large, the distributions for θ_1 and θ_2 barely overlap, i.e. samples from θ_1 are not particularly relevant for θ_2 and we observe $\bar{o} \simeq \hat{o}$ as suggested by Theorem A.4. However, the lower Δ , the greater the overlap, and as Δ approaches 0, observations become equally relevant for both designs and we see that only half the evaluations are needed to obtain similarly accurate estimates.

³Errors are averaged across 1000 independent runs collecting $|E'| = 1000$ incrementally by alternating between evaluating each of the two configurations.

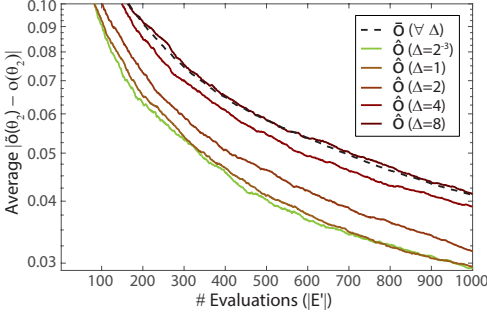


Figure 6.7: Absolute estimation error

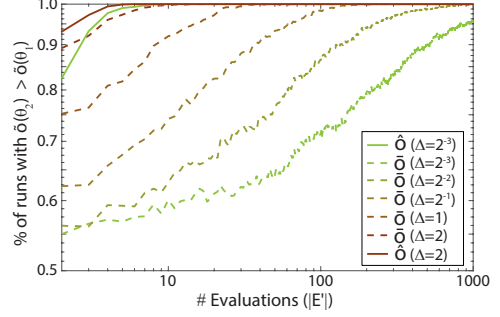


Figure 6.8: Comparison accuracy

Accurate comparisons: In the context of algorithm design, we are primarily concerned with accurate performance estimates because they allow us to more reliably tell which of a set of candidate designs performs better. Independent sample averages give rise to uncorrelated estimates, i.e. $\text{Cov}[\bar{o}(\theta_1), \bar{o}(\theta_2)] = 0$. In contrast, using IS, similar designs will share performance observations and their estimates will as such become positively correlated. In the extreme case where distributions fully overlap (e.g. $\Delta = 0$), performance estimates are the same, i.e. $\text{Cov}[\hat{o}(\theta_1), \hat{o}(\theta_2)] = 1$. As already discussed in Section 4.1.2, p. 109, in the context of “blocking”, positive correlations between estimates further increase comparison accuracy. Therefore, using IS, we expect to be able to more quickly assess which of two similar designs performs best; offering a more reliable gradient to guide the search. Figure 6.8 illustrates this, comparing the fraction of 1000 runs for which $\bar{o}(\theta_1) < \bar{o}(\theta_2)$ (dashed lines) and $\hat{o}(\theta_1) < \hat{o}(\theta_2)$ (full lines) holds after x evaluations, for different $\Delta > 0$. For high Δ values, we observe that both perform good, since $o(\theta_1) \ll o(\theta_2)$. However, for Δ approaching 0, designs become more similar, and we observe that the independent estimate of $o(\theta_1)$ is frequently better, even after many evaluations. However, $\hat{o}(\theta_1) < \hat{o}(\theta_2)$ holds, even for small Δ , after only a few evaluations.

Model and representation-free: Thus far, we have discussed the benefits of IS estimates over independent sample averages. But, how about using regression model predictions instead? Similar to IS, regression models allow one to generalize observations beyond the design used to obtain them, as such improving data efficiency. To do so, these models rely on a priori assumptions about the fitness landscape. Clearly, the exact nature of these assumptions and the flexibility to relax these to fit the data, are dependent on the type of model used. As parametric models typically impose very strict constraints, mainly non-parametric models are used in PbC; e.g. SMAC [Hutter et al. 2011] and GGA++ [Ansótegui et al. 2015] both use Random Forests [Breiman 2001]. While nonparametric models are

more flexible, they typically require more data and still rely on the assumption that similar designs perform similarly (i.e. smoothness). While this assumption is, in essence, reasonable, the key issue lies in that this similarity is measured in configuration (\sim representation) space. Small changes in configuration (\sim representation) might result in large performance differences, while large changes may not. Using IS estimates, similar designs will also have similar estimates, but rather than being based on assumed similarity in representation space, they are based on natural similarity in execution space, i.e. IS is model-free. Finally, the proposed IS approach does not preclude the use of regression models. In fact, we regard both as being complementary. For instance, lacking relevant observations for θ , one could rely on the predictions (assumptions) made by a regression model.

6.4 A Proof of Concept White Box Optimizer

Thus far, we have discussed how to estimate the performance of a given set of designs Θ , given a set of executions E' (and associated performance observations) obtained using $\Theta' \subseteq \Theta$. However, in natural occurrences of the PEP, we are typically not given E' and a matter which we did not address thus far is: *how to collect E'* ? Also, in this dissertation at large, we are concerned with solving the ADP. In particular, in this chapter, we consider doing so by reduction to the wb-ACP (see Definition 6.1). As argued in Section 6.3.1, contemporary configurators base important decisions (e.g. choice incumbent, search guidance) on estimates of performance. However, having maximally accurate information about the performance of candidate designs, i.e. solving the PEP, is only one part of the puzzle and leaves us with a wide array of choices of *how to use this information in solving the ADP*? Therefore, in order to validate the merit of the performance estimation approach presented in Section 6.3.2, we must consider its use in the context of a concrete wb-ACP solver (\sim a “white box” configurator). In this section, we describe the framework we use in Section 6.5. We decompose our description thereof in two logical parts:

Section 6.4.1: Which of the designs explored thus far do we return as anytime solution?

Section 6.4.2: In which order do we explore/evaluate the candidate designs?

6.4.1 Choice of Incumbent

Which of the designs explored thus far do we return as the solution at any time? Obviously, we would like our incumbent θ_{inc} to be the best design encountered thus far. One strategy would therefore update the incumbent whenever we encounter a design which is estimated to have superior performance. However, the reliability of performance estimates may vary strongly, and should therefore also be taken into account in choosing our incumbent.

6.4.1.1 Measuring the Reliability of an IS Estimate

Traditional black box configurators (e.g. ParamILS, SMAC iRace, GGA, etc.) use “sample size” (i.e. $|E'_\theta|$) as a measure of reliability of $\bar{o}(\theta)$, and only update the incumbent if the candidate incumbent has a superior estimated performance, *and* if this estimate is based on at least as many evaluations. The use of sample size in this manner can be motivated by the fact that the MSE decreases linearly with $|E'_\theta|$, i.e. $\mathbf{E}[(\bar{o}(\theta) - o(\theta))^2] = \frac{\mathbf{Var}_{e \sim \theta}[p(e)]}{|E'_\theta|}$.

However, what is the sample size of an IS estimate $\hat{o}(\theta)$? The concept “effective sample size” generalizes the notion of sample size to arbitrary estimators.

Definition 6.2: Effective Sample Size [Leister 2014]

The effective sample size of an estimator $\tilde{o}(\theta)$ for $\mathbf{E}_{e \sim \theta}[p(e)]$ is the unique number $n_{\text{eff}} \in \mathbb{R}$ satisfying $\mathbf{Var}[\tilde{o}(\theta)] = \frac{\mathbf{Var}_{e \sim \theta}[p(e)]}{n_{\text{eff}}}$.

Clearly, for $\tilde{o} = \bar{o}$ we have $n_{\text{eff}} = |E'_\theta|$. As it turns out, it is non-trivial to compute the actual n_{eff} for \hat{o} in Equation 6.7. Instead, we used a variant of the widely adopted approximate formula developed by Leslie Kish ([Kish 1965]):

$$\widehat{n}_{\text{eff}}^{\text{Kish}}(\theta) = \frac{(\sum_{e \in E'} w_\theta(e))^2}{\sum_{e \in E'} w_\theta(e)^2}.$$

Conceptually, while all IS estimates $\hat{o}(\theta)$ are based on the same sample of all executions E' , many of these may have relatively small weights w_θ . $\widehat{n}_{\text{eff}}^{\text{Kish}}(\theta)$ estimates how many observations effectively contribute to $\hat{o}(\theta)$. However, as a measure of reliability of an IS estimate, $\widehat{n}_{\text{eff}}^{\text{Kish}}$ has an important shortcoming: In settings where we do not have any relevant observations for θ , $\widehat{n}_{\text{eff}}^{\text{Kish}}(\theta)$ may nonetheless be high because many observations are similarly irrelevant for θ . In this setting, adding a single observations relevant for θ to E' will cause $\widehat{n}_{\text{eff}}^{\text{Kish}}(\theta)$ to suddenly drop to one. [Owen 2014, Section 9.3] suggested the use of $\bar{w}(\theta) = \frac{1}{|E'|} \sum_{e \in E'} w_\theta(e)$ as a complementary diagnostic. While $\bar{w}(\theta)$ should be roughly equal to one, $\bar{w}(\theta)$ will be very low exactly in those settings where $\widehat{n}_{\text{eff}}^{\text{Kish}}$ overestimates n_{eff} . In our PoC, we adapted Kish’s estimator accordingly:

$$\hat{N}(\theta) = \widehat{n}_{\text{eff}}^{\text{Kish}}(\theta) * \min(\bar{w}(\theta), \bar{w}(\theta))^{-1}. \quad (6.8)$$

It can be shown that $\hat{N}(\theta) = |E'_\theta|$ under the conditions (A) of Theorem A.4.

6.4.1.2 A Similarity-Aware Update Criterion

As discussed before, contemporary configurators require a candidate incumbent θ to satisfy (1) $\bar{o}(\theta) > \bar{o}(\theta_{\text{inc}})$ and (2) $|E'_\theta| \geq |E'_{\theta_{\text{inc}}}|$. If a design does not satisfy both criteria, but nonetheless looks promising (e.g. it may satisfy (1) but not (2)) they will collect additional performance observations in a procedure which is commonly referred to as a “race” (see also Section 4.1.2, p. 111), until it either no longer looks promising (e.g. violates (1)) or satisfies both criteria required to become the new incumbent (e.g. also satisfies (2)). In essence, we could do the same, substituting $\hat{o}(\theta)$ for $\bar{o}(\theta)$ and $\hat{N}(\theta)$ for $|E'_\theta|$.

However, we found that doing so can sporadically result in extremely long races when considering designs which are highly similar (due to correlations between estimates, see Section 6.3.2.5). This while the user is unlikely to actually care whether the optimizer returns contender θ or incumbent θ_{inc} as both designs likely perform similarly. To avoid such tragic waste of resources, we decided to relax (2) in such comparisons. Concretely, we accept θ as new incumbent whenever the following criteria are satisfied:

$$\begin{aligned} \hat{o}(\theta) &\geq \hat{o}(\theta_{\text{inc}}) \\ \text{and} \end{aligned} \tag{6.9}$$

$$\widehat{\text{sim}}(\theta, \theta_{\text{inc}})^K * (\hat{o}(\theta) - \hat{o}(\theta_{\text{inc}})) \geq (1 - \widehat{\text{sim}}(\theta, \theta_{\text{inc}})^K) * (\widehat{\text{unc}}(\theta) - \widehat{\text{unc}}(\theta_{\text{inc}}))$$

with

$$\widehat{\text{unc}}(\theta) = \sqrt{\frac{\text{VAR}}{\hat{N}(\theta)}} \quad \text{VAR} = \left(\frac{1}{|E'|} \sum_{e \in E'} p(e)^2 \right) - \left(\frac{1}{|E'|} \sum_{e \in E'} p(e) \right)^2 \tag{6.10}$$

$$\widehat{\text{sim}}(\theta, \theta') = \frac{\sum_{e \in E'} \min(w_\theta(e), w_{\theta'}(e))}{|E'| \max(\bar{w}(\theta), \bar{w}(\theta'))} \tag{6.11}$$

Here, $\widehat{\text{unc}}(\theta)$ is a rough estimate of the standard error on $\hat{o}(\theta)$. $\widehat{\text{sim}}(\theta, \theta')$ estimates the similarity between two designs θ and θ' .⁴ Finally, K is a non-negative constant which is passed as a parameter to the framework (default value of 3).

⁴In Equation 6.11, we used an IS estimate for the overlap between their corresponding distributions over executions: $\frac{1}{|E|} \sum_{e \in E} \min(\text{pr}(e|\theta), \text{pr}(e|\theta'))$. However, other statistical distance measures (e.g. Kullback-Leibler divergence) could be used as well. Remark that $\widehat{\text{sim}}$ satisfies

1. $\widehat{\text{sim}}(\theta, \theta') = \widehat{\text{sim}}(\theta', \theta)$
2. $0 \leq \widehat{\text{sim}}(\theta, \theta') \leq 1$.
3. $\widehat{\text{sim}}(\theta, \theta') = 0 \iff \forall e \in E' : \text{pr}(e|\theta) * \text{pr}(e|\theta') = 0$.
4. $\widehat{\text{sim}}(\theta, \theta') = 1 \iff \forall e \in E' : \text{pr}(e|\theta) = \text{pr}(e|\theta')$.

Also, under the conditions (A) of Theorem A.4 we have $\widehat{\text{sim}}(\theta, \theta') = \begin{cases} 1 & \theta = \theta' \\ 0 & \theta \neq \theta' \end{cases}$.

While seemingly complex, the intuition behind the second criterion is relatively straightforward. Ignoring the $\widehat{\text{sim}}$ factors, the left-hand side of the inequality corresponds to the estimated improvement in performance (1), while the right-hand side corresponds to the estimated loss in reliability (2), associated with choosing θ , rather than θ_{inc} , as incumbent. $\widehat{\text{sim}}$ can be viewed to determine the trade-off between (1) and (2): When $\widehat{\text{sim}}$ is 0, the second criterion reduces to $\hat{N}(\theta) \geq \hat{N}(\theta_{\text{inc}})$, i.e. requires both estimates to be at least equally reliable, when $\widehat{\text{sim}}$ is 1, it reduces to $\hat{o}(\theta) \geq \hat{o}(\theta_{\text{inc}})$. Finally, the constant K determines for what value of $\widehat{\text{sim}}$, (1) and (2) are equally important, i.e. $2^{-\frac{1}{K}}$. For $K = 1$, this would be 0.5. We choose $K > 1$, as we wish to relax criterion (2) only if designs are highly similar. For our default choice of $K = 3$, break-even occurs at similarity 0.8.

6.4.2 High-level Search Strategy

In this section, we describe the high-level search strategy followed by our PoC (see Algorithm 21), which determines the order in which candidate designs are explored and evaluated. Our strategy closely resembles that of a traditional SMBO framework, differing in that we use IS estimates, rather than regression model predictions to guide our search.

First, we perform some initialization. At line 2, we choose θ_{inc} , the incumbent, to be θ_{init} , the design which is passed as an (optional⁵) argument to the framework. At lines 3 and 4, we initialize the IS “model”, denoted by \hat{M} , and defined as a 4-tuple $\langle \text{pr}', p, E', \Theta' \rangle$, where E' is the sample of executions generated thus far and Θ' the bag of designs used to generate them. Initially $E' = \Theta' = \emptyset$. Remark that \hat{M} encapsulates all information required to estimate \hat{o} (performance, see Equation 6.7), \widehat{unc} (standard error, see Equation 6.10) and $\widehat{\text{sim}}$ (similarity, see Equation 6.11). At line 5, we initialize counters for the number of “proposals generated” ($\# \text{prop}$) and “iterations performed” ($\# \text{it}$) thus far to 0. At line 6, we initialize “the number of proposals to generate next iteration” (m) to one for the first iteration. Subsequently, we iteratively

1. explore a sample of candidate designs $\Theta_{\text{prop}} \subset \Theta$ (line 8)
2. select one of these as a contender θ_{prop} (line 9)
3. evaluate both θ_{prop} and θ_{inc} in the context of a race (line 10).

This process continues until “the number of candidate design executions performed” reaches N , a preset limit, after which we return θ_{inc} . Note that $N = +\infty$ is a valid configuration, as our PoC is a proper anytime framework. In what follows, we describe the procedures we used for (1), (2) and (3) in more detail.

⁵If omitted, a design is selected randomly according to $\Theta.\mathcal{GP}$, see Section 6.4.2.1.

Algorithm 21 High-level search strategy for our PoC.

```

1: function SOLVE( $\langle a, \Theta, \mathcal{D}, \text{pr}', \text{p} \rangle, K, L, N, \theta_{\text{init}}$ )
2:    $\theta_{\text{inc}} \leftarrow \theta_{\text{init}}$ 
3:    $\Theta', E' \leftarrow \emptyset$ 
4:    $\hat{M} \leftarrow \langle \text{pr}', \text{p}, E', \Theta' \rangle$ 
5:    $\#it, \#prop \leftarrow 0$ 
6:    $m \leftarrow 1$ 
7:   while  $|E'| < N$  do
8:      $\langle \Theta_{\text{prop}}, \theta_{\text{inc}}, \#prop \rangle \leftarrow \text{EXPLORE}(\Theta, K, m, \#prop, \hat{M})$ 
9:      $\theta_{\text{prop}} \leftarrow \arg \max_{\theta \in \Theta_{\text{prop}}} \frac{\hat{M}.o(\theta) - \hat{M}.o(\theta_{\text{inc}})}{\hat{M}.unc(\theta) + \hat{M}.unc(\theta_{\text{inc}})} - \frac{\hat{M}.sim(\theta, \theta_{\text{inc}})^K}{1 - \hat{M}.sim(\theta, \theta_{\text{inc}})^K}$ 
10:     $\langle E', \Theta', \theta_{\text{inc}}, \hat{M} \rangle \leftarrow \text{RACE}(a, \mathcal{D}, K, N, \theta_{\text{prop}}, \theta_{\text{inc}}, \hat{M})$ 
11:     $\#it \leftarrow \#it + 1$ 
12:     $m \leftarrow \min\left(\frac{|E'|}{\#it} * \frac{L - \frac{\#prop}{N}}{1 - \frac{|E'|}{N}}, (|E'| + \min(2, N - |E'|)) * L - \#prop\right)$ 
13:  end while
14:   $\langle \_, \theta_{\text{inc}}, \_ \rangle \leftarrow \text{EXPLORE}(\Theta, K, L * N - \#prop, \#prop, \hat{M})$ 
15:  return  $\theta_{\text{inc}}$ 
16: end function

17: function EXPLORE( $\Theta, K, m, \#prop, \hat{M}$ )
18:    $\Theta_{\text{prop}} \leftarrow \emptyset$ 
19:   for  $i=1:m$  do
20:      $\mathcal{P}_{\theta_{\text{inc}}} \leftarrow \text{equal mixture of } \Theta.\mathcal{GP} \text{ and } \Theta.\mathcal{LP}(\theta_{\text{inc}})$ 
21:      $\theta_i \sim \mathcal{P}_{\theta_{\text{inc}}}$ 
22:      $\Theta_{\text{prop}} \leftarrow \Theta_{\text{prop}} \cup \{\theta_i\}$ 
23:      $\theta_{\text{inc}} \leftarrow \text{UPDATEINCUMBENT}(K, \theta_i, \theta_{\text{inc}}, \hat{M})$ 
24:   end for
25:   return  $\langle \Theta_{\text{prop}}, \theta_{\text{inc}}, \#prop + m \rangle$ 
26: end function

27: function UPDATEINCUMBENT( $K, \theta, \theta_{\text{inc}}, \hat{M}$ )
28:   if  $\hat{M}.o(\theta) \geq \hat{M}.o(\theta_{\text{inc}})$  then
29:      $s \leftarrow \hat{M}.sim(\theta, \theta_{\text{inc}})$ 
30:     if  $s^K * (\hat{M}.o(\theta) - \hat{M}.o(\theta_{\text{inc}})) \geq (1 - s^K) * (\hat{M}.unc(\theta) - \hat{M}.unc(\theta_{\text{inc}}))$  then
31:       return  $\theta$ 
32:     end if
33:   end if
34:   return  $\theta_{\text{inc}}$ 
35: end function

```

```

36: function RACE( $a, \mathcal{D}, K, N, \theta_{\text{prop}}, \theta_{\text{inc}}, \hat{M}$ )
37:    $\langle E', \Theta', \hat{M} \rangle \leftarrow \text{TEST}(a, \mathcal{D}, \theta_{\text{inc}}, \hat{M})$ 
38:   repeat
39:      $\langle E', \Theta', \hat{M} \rangle \leftarrow \text{TEST}(a, \mathcal{D}, \theta_{\text{prop}}, \hat{M})$ 
40:      $\theta_{\text{inc}} \leftarrow \text{UPDATEINCUMBENT}(K, \theta_{\text{prop}}, \theta_{\text{inc}}, \hat{M})$ 
41:   until  $|E'| = N \vee \theta_{\text{inc}} = \theta \vee \hat{M}.o(\theta_{\text{prop}}) < \hat{M}.o(\theta_{\text{inc}})$ 
42:   return  $\langle E', \Theta', \theta_{\text{inc}}, \hat{M} \rangle$ 
43: end function

44: function TEST( $a, \mathcal{D}, \theta, \langle \text{pr}', p, E', \Theta' \rangle$ )
45:   if  $|E'| < N$  then
46:      $x \leftarrow \mathcal{D}.\text{sample}()$ 
47:      $e \leftarrow a_{\theta}(x)$ 
48:      $E' \leftarrow E' \cup \{e\}$ 
49:      $\Theta' \leftarrow \Theta' \cup \{\theta\}$ 
50:      $\hat{M} \leftarrow \langle \text{pr}', p, E', \Theta' \rangle$ 
51:   end if
52:   return  $\langle E', \Theta', \hat{M} \rangle$ 
53: end function

```

6.4.2.1 Design Space Exploration

In this section, we describe the procedure used to search the design space for designs which are (potentially) better than our incumbent. Each iteration, we generate a sample of m designs (Θ_{prop}) according to $\mathcal{P}_{\theta_{\text{inc}}}$, a distribution over Θ , dependent on θ_{inc} .

How many proposals do we generate (m)? In the first iteration, we generate only a single proposal which is also the first contender. In subsequent iterations, m is chosen such that we on average explore L proposals for every candidate design execution, where L is passed as an argument to the framework (default value 10). Since we must calculate \hat{o} , $\widehat{\text{unc}}$ and $\widehat{\text{sim}}$ for every proposal, generating a large number of proposals will increase the overhead and therefore the total runtime. For $m = 10$, we found the total runtime of our framework, on our platform (see Appendix A.1), to be similar or lower than that of the other configurators for the scenarios we considered in Section 6.5. One challenge in choosing m is that the total number of iterations is not known a priori since the candidate design executions per iteration (\sim duration of the race) varies. We used the formula at line 12, which allocates the total of $L * N$ proposals roughly uniformly across iterations while guaranteeing that at least $2L$ candidates are explored per iteration. Remark that if the final race takes $2 + x$ executions, this will cause us to explore only $L * (N - x)$ designs. Something which we correct for at line 14, where we explore the remaining $L * x$ designs.

How do we generate these proposals (lines 17-26)? We draw each proposal θ_i from a distribution $\mathcal{P}_{\theta_{\text{inc}}}$, which is an equal mixture of two distributions which are passed as arguments to the framework and which we can generate samples according:

Global Prior (Θ, \mathcal{GP}): A distribution over Θ , allowing the user to encode prior knowledge about where good designs are most likely located in Θ .

Local Prior (Θ, \mathcal{LP}): A distribution over Θ , conditioned on θ_{inc} , allowing the user to encode prior knowledge of how θ_{inc} can be best improved.

Note that our search strategy only interacts with Θ indirectly, through these distributions, making it design representation independent. In particular, it does not assume them to be configurations, let alone makes any assumptions about the type of parameters.

Remark that after generating a proposal θ_i (line 23), we check whether it satisfies the criteria in Equation 6.9 and update the incumbent accordingly (lines 27-35), as described in Section 6.4.1. This implies that our incumbent is potentially a design which itself has never been executed. This is a key difference with contemporary SMBO frameworks, which do not base the choice of incumbent on the predictions made by the regression model.

6.4.2.2 Selecting a Contender

In this section, we discuss which of the candidate designs Θ_{prop} we select to be evaluated in a race against the incumbent. Here, we wish to select the design which we believe to be most likely better than our incumbent based on \hat{o} and $\widehat{\text{unc}}$. Also, as discussed in Section 6.4.1, we wish to avoid selecting θ with high $\widehat{\text{sim}}(\theta, \theta_{\text{inc}})$, as doing so could result in extremely long races. To this end, we select the design θ maximizing (line 9):

$$\frac{\hat{o}(\theta) - \hat{o}(\theta_{\text{inc}})}{\widehat{\text{unc}}(\theta) + \widehat{\text{unc}}(\theta_{\text{inc}})} - \frac{\widehat{\text{sim}}(\theta, \theta_{\text{inc}})^K}{1 - \widehat{\text{sim}}(\theta, \theta_{\text{inc}})^K}.$$

Here, the first term is an approximate, signed, non-parametric measure of the significance of the estimated performance difference, roughly imposing the following order

1. θ with $\hat{o}(\theta) > \hat{o}(\theta_{\text{inc}})$, where lower $\widehat{\text{unc}}$ are better.
2. θ with $\hat{o}(\theta) \leq \hat{o}(\theta_{\text{inc}})$, where higher $\widehat{\text{unc}}$ are better.

The second term penalizes designs which are too similar to θ_{inc} .

Why not the EI criterion? We have considered using the Expected Improvement (EI) criterion [Jones et al. 1998], also used in SMAC (see Section 4.1.2, p. 116). However, EI assumes the error on \hat{o} to be approximately normally distributed with standard deviation $\widehat{\text{unc}}$. When $\widehat{\text{unc}}$ is high, we observed this not to be the case, which led to an overestimation of the actual expected improvement and designs with high uncertainty being selected at the cost of those which highly likely have superior performance.

6.4.2.3 Incremental Evaluation Mechanism

The contender θ_{prop} is a design which we believe, based on \hat{M} , to be potentially better than the incumbent, but that does not satisfy the criteria required to become the new incumbent. In this section, we describe the mechanism we used to collect additional performance observations, update \hat{M} , and potentially the incumbent. The procedure used resembles the aggressive racing scheme used in ParamILS, ROAR and SMAC, described in Section 4.1.2, p. 113. Here, we first execute θ_{inc} (line 37) to avoid that an overly optimistic estimate of $o(\theta_{\text{inc}})$ would cause us to fail to identify better designs. Subsequently, we execute the contender θ_{prop} once, and continue to do so until either $\hat{o}(\theta_{\text{prop}}) < \hat{o}(\theta_{\text{inc}})$ holds or it satisfies all criteria required to become the new incumbent (lines 38-41).

6.5 Experimental Validation

In this section, we validate the benefits of the novel performance estimation technique using Importance Sampling (IS), which we introduced in Section 6.3.2, experimentally. To this end, we consider three⁶ different scenarios where we formulate an ADP as a wb-ACP which we subsequently solve using the (white box) configurator described in Section 6.4 implementing this technique (PoC-IS). We compare its anytime performance, i.e. the expected average-case performance of θ_{inc} at any point in time, to that of

- A variant using sample average (instead of IS) estimates for $o(\theta)$ (PoC-SA).⁷
- Various state-of-the-art (black box) configurators: ParamILS,⁸ iRace⁹ and SMAC.¹⁰

Disclaimer: The scenarios we consider here were not chosen arbitrarily, nor did we choose them to be maximally representative for the ADPs one may encounter in practice. Rather, we have intentionally chosen settings where we conjectured the wb-ACP form of the problem to be considerably easier to solve than the ACP form. That being said, with exception of the first scenario which is a “toy” problem (see Section 6.5.1), all problems considered correspond to relevant instances of the ADP which are part of a larger family of problems which we expect the observations made in this chapter to generalize to.

In what follows, we study each scenario in a separate subsection. Here, we first briefly introduce the ADP considered. Subsequently, we describe the wb-ACP reduction (i.e. our choices for a , Θ , \mathcal{D} , pr' and p), our experimental setup (e.g. parameters choices for the frameworks compared). Finally, we present and discuss the results.

⁶In [Adriaensen et al. 2017], we only considered the first of these scenarios (see Section 6.5.1).

⁷The code for PoC-IS/SA is available at <http://github.com/Steven-Adriaensen/IS4APE>.

⁸<http://www.cs.ubc.ca/labs/beta/Projects/ParamILS/> (version 2.3.8)

⁹<http://iridia.ulb.ac.be/irace/> (version 2.4.1844)

¹⁰<http://www.cs.ubc.ca/labs/beta/Projects/SMAC/> (version 2.10.03)

6.5.1 The Looping Problem: Break or Continue

The first scenario we consider is a “toy” dynamic ASP where we are to execute a loop for at most H iterations. Each iteration i , we encounter a choice point and have to decide whether to continue or break the loop, if we continue, we receive a reward $r_i \sim \mathcal{N}(1, 4)$, otherwise execution terminates. As in any dynamic ASP, the goal is to make decisions at these choice points as to maximize the total expected reward collected. As the rewards received are assumed to be a priori unpredictable (truly random), the optimal policy is to continue the loop till the end, receiving an expected reward of H . The toy problem described above is the one we considered in [Adriaensen et al. 2017] (with $H = 20$) and which was introduced in [Adriaensen and Nowé 2016b] (with $H = 10$) under the rather cryptic name “benchmark 1”. Here, we will refer to it as “the looping problem”.

Toy problems are problems whose solutions themselves are not of any practical interest, but which are nonetheless considered in the analysis of algorithms because they are “convenient” (e.g. conceptually simple, facilitate replication, isolate some property, etc.), while exhibiting features also found in complex real-world problems. The use of toy problems is common practice in many fields; e.g. the grid worlds in RL, the One Max problem in combinatorial optimization, the prisoner’s dilemma in game theory, etc.

6.5.1.1 Formulation as a wb-ACP

We now formulate the looping problem with $H = 20$ as a wb-ACP:

a is the framework listed in Algorithm 22 executing the interruptible loop described above. It takes 20 numerical parameters as input ($|\theta| = 20$), where the value of the i^{th} parameter (θ_i) determines the likelihood of continuing in the i^{th} iteration. It returns $e = (\#it, r_\Sigma)$, where $\#it$ is the number of iterations performed and r_Σ the sum of rewards received in each iteration.

Θ : We consider two alternative configuration spaces in this scenario:

Θ_{cont} : Each parameter p_i can take any real value in $[0, 1]$, i.e. $\Theta_i = [0, 1]$.

Θ_{discr} : Each parameter p_i can take 11 different values: $\Theta_i = \{0, 0.1, \dots, 0.9, 1\}$.

\mathcal{D} : Note that a does not take any inputs beyond θ . For completeness, we let the empty string ϵ be the only possible input ($X = \{\epsilon\}$) and choose $\mathcal{D}(\epsilon) = 1$.

$$\text{pr}'(e|\theta) = \begin{cases} (1 - \theta_{\#it+1}) \prod_{i=1}^{\#it} \theta_i & 0 \leq \#it \leq 19 \\ \prod_{i=1}^{\#it} \theta_i & \#it = 20 \end{cases}.$$

$$p(e) = r_\Sigma.$$

Remark that $o(\theta) = \sum_{i=1}^{20} \prod_{j=1}^i \theta_j$ and the solution is $\theta^* : \theta_i^* = 1, \forall i$ with $o(\theta^*) = 20$.

Algorithm 22 Target algorithm used in the wb-ACP reduction of the looping problem.

```

1: procedure LOOP( $\theta$ )
2:    $\#it, r_\Sigma \leftarrow 0$ 
3:   for  $i = 1 : 20$  do
4:      $\text{pivot} \sim \mathcal{U}(0, 1)$ 
5:     if  $\theta_i < \text{pivot}$  then
6:       goto line 12 (break)
7:     end if
8:      $\#it \leftarrow i$ 
9:      $r_i \sim \mathcal{N}(1, 4)$ 
10:     $r_\Sigma = r_\Sigma + r_i$ 
11:   end for
12:   return  $\langle \#it, r_\Sigma \rangle$ 
13: end procedure

```

6.5.1.2 Experiment

Setup: In the previous section, we formulated two wb-ACPs, i.e. $\langle a, \Theta_{\text{cont}}, \mathcal{D}, \text{pr}', p \rangle$ and a discretized variant thereof $\langle a, \Theta_{\text{disc}}, \mathcal{D}, \text{pr}', p \rangle$. On the discretized variant, we compare PoC-IS to PoC-SA, (focused) ParamLLS, iRace and SMAC. On the continuous variant, we only consider PoC-SA, iRace and SMAC as a baseline, since ParamLLS does not support parameters with infinite domains. We performed 1000 independent runs of 10000 evaluations for every framework, using their default parameter settings, all starting from the same initial configuration $\theta_{\text{init}} : \theta_i = 0.5, \forall i$. For fair comparison, we chose the global and local prior used in PoC-IS and PoC-SA to be uninformative: $\Theta.\mathcal{GP}$ chooses $\theta \sim \mathcal{U}(\Theta)$ (uniform), $\Theta.\mathcal{LP}$ chooses $\theta_j \sim \mathcal{U}(\Theta_j)$, for some randomly selected j , and $\theta_i = \theta_{\text{inc},i}$ otherwise (random one-exchange). Remark that these are the same local and global search operators as are used in ParamLLS. We did not provide any of the frameworks with a priori information about the conditionalities between parameters.

Results: Figures 6.9 and 6.10 show the (actual) performance of the incumbent design obtained by each framework, after x evaluations, averaged over 1000 runs. For the discretized setting, we find that none of the configurators in our baseline (e.g. PoC-SA, ParamLLS, iRace and SMAC) found the optimal solution (θ^*) within 10000 evaluations. ParamLLS performed worst, returning a design with an average performance of 2. PoC-SA and iRace both returned a design with an average performance of 5. Of our baseline, SMAC performed best, returning a design with an average performance of roughly 6 after 10000 evaluations. PoC-IS needed only 100 evaluations to obtain a similar performance as SMAC, and all of the runs found θ^* within 10000 evaluations. Our observations in the continuous setting are similar, except that each framework performed slightly worse.

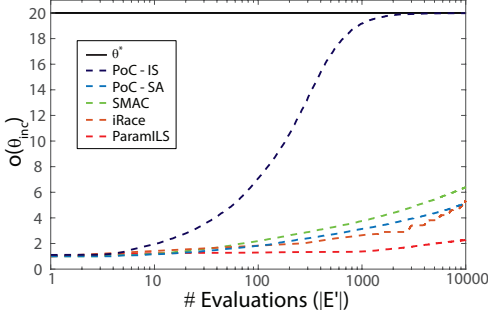


Figure 6.9: Results for the looping problem (wb-ACP with discretized Θ)

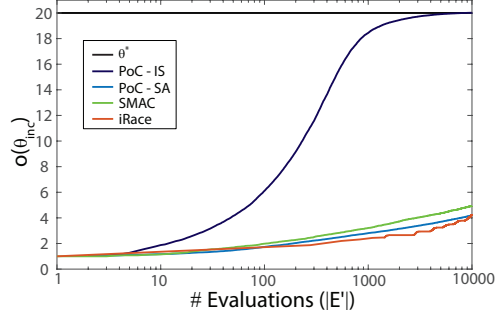


Figure 6.10: Results for the looping problem (wb-ACP with continuous Θ)

6.5.2 Static Sorting Portfolio

In the second scenario, we consider the problem of designing an algorithm to sort a sequence of integers in ascending order. In particular, we look at solving the sorting subset-ASP which we introduced in Section 4.2.2. Here, we considered the design of a *static* portfolio-solver which applies one of eight canonical sorting algorithms depending on features of the sequence to be sorted. Remark that the experiments performed in this section can be viewed as an illustration of the “solving subset-ASPs by set-ASP reduction” approach which we discussed in Section 4.4.1, p. 148.

6.5.2.1 Formulation as a wb-ACP

We reduce the sorting subset-ASP to the following wb-ACP:¹¹

a is a parametrized static portfolio-solver (see Algorithm 23). Next to a sequence of integers to be sorted (l), a takes 40 real-valued parameters as input which determine the selection mapping. First, Algorithm 13 is used to extract the four features of the input sequence l . Subsequently, at lines 3-7, these features are transformed (log-scaled/normalized) as in the clustering approach described in 4.2.3, p. 122. Next, the selection mapping s_θ is computed for these features, as follows: At line 15, we compute an affinity for each of the eight sorting algorithms (A) as a linear combination of the feature vector, such that the affinity of the j^{th} algorithm is given by $\theta_j + \sum_{i=1}^4 \theta_{8i+j} * f'_i$. Afterwards, the algorithm which has the highest affinity (a_{sel}) is returned and used to sort l . Finally, next to the sorted sequence l , a returns $e = (f', a_{\text{sel}}, \frac{-t}{|l|})$, where t is the time a_{sel} took to sort l .

¹¹This can be viewed as an instance of the general reduction described in Section 6.2.3.2.

Algorithm 23 Target algorithm used in the wb-ACP reduction of the sorting subset-ASP.

```

1: procedure SORT( $\theta, l$ )
2:    $f = \langle f_{\text{length}}, f_{\text{range}}, f_{\text{sorted}}, f_{\text{equal}} \rangle \leftarrow \text{EXTRACTFEATURES}(l)$ 
3:    $f^{\min} \leftarrow \langle 2, -10^9, 0, 0 \rangle$ 
4:    $f^{\max} \leftarrow \langle 10^5, 10^9, 1, 1 \rangle$ 
5:   for  $i = 1 : 4$  do ▷ Normalize and log-scale features.
6:      $f_i \leftarrow \frac{\log(f_i - f_i^{\min} + 1)}{\log(f_i^{\max} - f_i^{\min} + 1)}$ 
7:   end for
8:    $a_{\text{sel}} \leftarrow s(\theta, f)$ 
9:    $\text{timelapse}()$  ▷ Reset the timer to only time Line 10.
10:   $l \leftarrow a_{\text{sel}}(l)$ 
11:  return  $\langle l, \langle f, a_{\text{sel}}, \frac{-\text{timelapse}()}{|l|} \rangle \rangle$ 
12: end procedure

13: procedure  $s(\theta, f')$  ▷ Compute  $s_\theta$  for an input with features  $f'$ .
14:   $A \leftarrow \langle \text{bubble\_sort, selection\_sort, insertion\_sort, merge\_sort, quicksort,} \\ \text{heapsort, radix\_sort, tim\_sort} \rangle$ 
15:   $q \leftarrow \{(a_j, v_j) \mid 1 \leq j \leq 8 \wedge (a_j \text{ is the } j^{\text{th}} \text{ element of } A) \wedge v_j = \theta_j + \sum_{i=1}^4 \theta_{8i+j} * f'_i\}$ 
16:  return  $\arg \max_{a_j} q(a_j)$ 
17: end procedure

```

Θ : Each parameter p_i can take any real value in $[-1, 1]$, i.e. $\Theta_i = [-1, 1]$.

\mathcal{D} is the same as in the sorting subset-ASP (see Section 4.2.2).

$\text{pr}'(e|\theta) = [s(\theta, f') = a_{\text{sel}}]$, i.e. 1 iff s_θ selects a_{sel} , 0 otherwise.

$p(e) = \frac{-t}{|l|}$, i.e. time it took per element sorted (\sim Section 4.2.2).

6.5.2.2 Experiment

setup: For this scenario, we consider PoC-SA, iRace and SMAC as a baseline. Since the wb-ACP formulated in the previous section has real-valued parameters, we did not include ParamILS in our comparison. We performed 1000 independent runs of 100000 evaluations for every framework, using their default parameter settings. Apart from the default configuration, we also considered a “customized” configuration for iRace, setting the “elitist limit” parameter to zero (default two. According to [López-Ibáñez et al. 2016], this parameter determines the “*maximum number of statistical tests performed without successful elimination after all instances from the previous race have been evaluated. If the limit is reached, the current race is stopped.*” In high-variance settings, the default

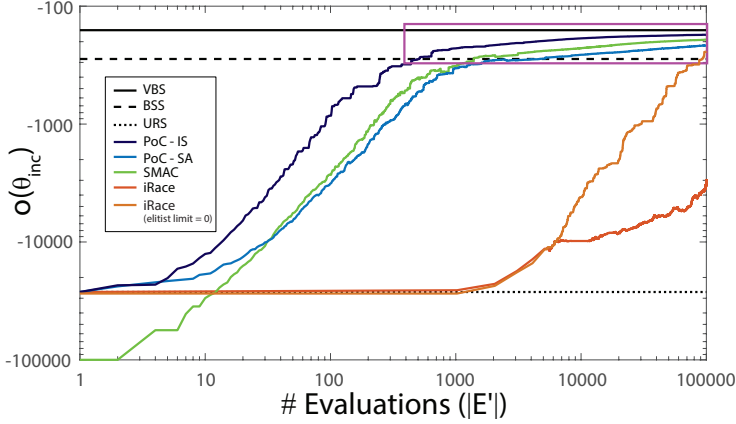


Figure 6.11: Results for the static sorting portfolio scenario.

“elitist limit” of two, will terminate races early. While this allows iRace to perform more iterations and explore more different configurations, it will evaluate these on only a few inputs (± 40), too little to significantly reduce the variance in performance estimates and reliably determine which of these are actually best. From an ML perspective, $s_{\theta_{\text{inc}}}$ will likely overfit such a small subset of training inputs. In setting this parameter to 0, we effectively disable this mechanism, and found that iRace (\sim PoC and SMAC), evaluates θ_{inc} on more than a thousand inputs. For PoC-IS and PoC-SA, we choose $\Theta.\mathcal{GP} : \theta \sim \mathcal{U}(\Theta)$ (uniform) and $\Theta.\mathcal{LP} : \theta_j = \max(-1, \min(v, 1))$ with $v \sim \mathcal{N}(\theta_{\text{inc},j}, \frac{1}{40})$ (truncated multivariate Gaussian centered at θ_{inc}). We did not specify an initial configuration for iRace nor PoC, causing the configurators to select one uniformly at random from Θ . For SMAC, θ_{init} is to the best of our knowledge not optional and was set to $\theta_{\text{init}} : \theta_i = 0, \forall i$.

Results: Figure 6.11 shows the (actual) performance of the incumbent design obtained by each framework, after x evaluations, averaged over 1000 runs. As a reference, we also show the performance of the following selection mappings: the Virtual Best Solver (VBS, an oracle selecting the best algorithm for each input), the Best Single Solver (BSS, i.e. tim sort) and uniform random selection (URS).

We observe that PoC-IS needs only 500 evaluations to obtain a design which performs as good as the BSS, which is roughly three times less than SMAC, 10x less than PoC-SA and nearly 200x less than iRace (elitist limit disabled). We observe that iRace, using its default parameter setting, performs poorly. It does not obtain BSS performance, even after 100000 evaluations. In general, compared to other configurators considered, we find that the anytime performance of iRace is clearly inferior in this scenario.

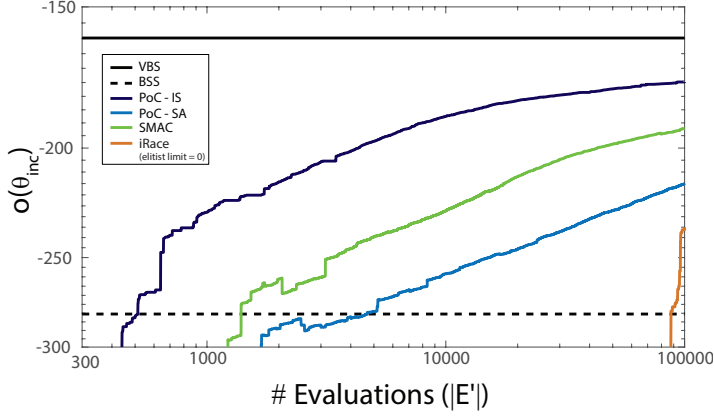


Figure 6.12: Detail of Figure 6.11 (purple frame)

Figure 6.12 zooms in on an area of interest in Figure 6.11 (indicated by the purple frame), allowing us to compare the configurator's ability to close the gap with the VBS (see p. 124, footnote). We observe that after 100000 evaluations, iRace (elitist limit disabled), PoC-SA and SMAC close the gap with the VBS with 35%, 55% and 80%, respectively. PoC-IS requires 100x, 50x and 15x less evaluations, respectively, to obtain a design with similar performance, eventually closing the gap with 87.5% after 100000 evaluations. The best selection mapping encountered in any run of any configurator closes the gap with 90%.

6.5.3 Dynamic Metaheuristic Scheduler

As a final scenario, we consider the problem of designing a scheduler allocating resources dynamically across two given metaheuristic methods. While our focus is on selection hyper-heuristics implemented using HyFlex (see Section 5.2.1.2), the same methodology could be applied to anytime optimizers in general. Also, in the spirit of reuse, we consider the design of a generic policy for scheduling any two, rather than two specific, optimizers.

Motivation: In Section 5.3.2.4, we have compared the performance of different selection hyper-heuristics across 98 instances taken from nine different domains. While some performed, on average, better than others, we found that none performed consistently best across all instances. Therefore, it makes sense to apply a “portfolio approach” where we attempt to automatically apply the most appropriate method for each instance. We consider dynamic scheduling, rather than (static) algorithm selection, since the information that is available (or that can be derived efficiently) a priori about the input (i.e. the problem instance to be solved and the two candidate solvers) is insufficient to permit a reliable prediction of which method will perform best.

The candidate schedulers considered subdivide the computational budget t_{allowed} in N “slots” of $\frac{t_{\text{allowed}}}{N}$ seconds and iteratively allocate these slots to the two methods based on their anytime performance in previously assigned slots. Below, we consider a computational budget of 10 minutes, subdivided into 100 slots of six seconds each.

6.5.3.1 Formulation as a wb-ACP

Remark that the problem characterized above can be viewed as a dynamic ASP, where we encounter 100 binary choice points in which we are to choose to which of the two methods we assign the next slot. As such, we could essentially apply the general dynamic-ASP reduction described in Section 6.2.3.3. However, we considered a reduction specific to the dynamic scheduling problem instead. In what follows, we will restrict ourselves to a high-level description thereof, as its specifics are non-trivial and not essential for interpreting our experimental results. Details can be found in Appendix A.4.

a : Our target algorithm (see Algorithm 24, Appendix A.4) takes x , the minimization problem instance to be solved, hh_1 and hh_2 , two preemptable anytime optimizers that can be used for doing so, and a configuration θ consisting of 22 real-valued parameter values determining how resources are dynamically allocated to hh_1 and hh_2 (\sim scheduling policy π_θ). For each slot, a determines pr_1 , the likelihood of assigning it to hh_1 (instead of hh_2), as follows: First, a derives two sets of five features ω_1 and ω_2 , characteristic of “the anytime performance” of hh_1 and hh_2 , respectively, during “previously allocated slots”. Subsequently, a passes ω_1 and ω_2 one-by-one as input to a neural network, whose weights are given by θ . This network’s outputs v_1 and v_2 are used to compute pr_1 as $\frac{v_1}{v_1+v_2}$. When all 100 slots are allocated, a assesses p_e the desirability of the allocation it made. To do so, it compares the quality of the best solution obtained, to those obtained by all 100 other hypothetical allocations, such that $p_e \in [-1, 0]$ with -1 for the unique worst, and 0 for the unique best possible allocation. Finally, it returns $e = \langle T_1, T_2, n_1, p_e \rangle$, where T_j is the solution quality trace, i.e. quality of the anytime solution at any time, of hh_j , and n_1 the number of slots allocated to hh_1 . See Appendix A.4 for more details.

Θ : Each of the 22 parameters can take any real value in $[-100, 100]$, i.e. $\Theta_i = [-100, 100]$.

\mathcal{D} : a takes a minimization problem x and two (anytime) optimizers hh_1 and hh_2 as input. As possible x , we consider the 98 instances in the (extended) HyFlex benchmark (X_{all} , see Appendix A.2). As possible hh_1, hh_2 , we consider any pair of the eight ASAP hyper-heuristics (see Section 5.3.2.4). This results in a total of 2744 inputs, where \mathcal{D} .sample samples uniformly at random from.

$\text{pr}'(e|\theta)$ is the likelihood that the scheduling policy π_θ allocates $(n_1, 100 - n_1)$ slots to (hh_1, hh_2) , given their traces (T_1, T_2) . This can be computed efficiently/elegantly using dynamic programming (see Algorithm 25, Appendix A.4).

$$p(e) = p_e.$$

6.5.3.2 Experiments

Setup: For this scenario, we consider PoC-SA, iRace and SMAC as a baseline. We performed 1000 independent runs of 1000 evaluations for every framework, using their default parameter settings. For PoC-IS and PoC-SA, we choose $\Theta.\mathcal{GP}$: $\theta \sim \mathcal{U}(\Theta)$ (uniform) and $\Theta.\mathcal{LP}$: $\theta_j = \max(-100, \min(v, 100))$ with $v \sim \mathcal{N}(\theta_{\text{inc},j}, 5)$ (truncated multivariate Gaussian centered at θ_{inc}). We did not specify an initial configuration for iRace nor PoC, causing the configurators to select one uniformly at random from Θ . For SMAC, θ_{init} was set to $\theta_{\text{init}} : \theta_i = 0, \forall i$.

Results: Figure 6.13 shows the (actual) performance of the incumbent design obtained by each framework, after x evaluations, averaged over 1000 runs. As a reference, schedulers allocating all slots to an arbitrary method have $o = -0.524$. An oracle scheduler, allocating all slots to the method which obtains the best solution, has $o = -0.190$.

We observe that after 1000 evaluations, iRace, PoC-SA and SMAC obtain a design which has an average-case performance o of -0.407 , -0.374 and -0.364 , respectively. PoC-IS requires roughly 40x, 10x and 5x less evaluations, respectively, to obtain a design with similar performance, eventually obtaining one with $o = -0.346$ on average.

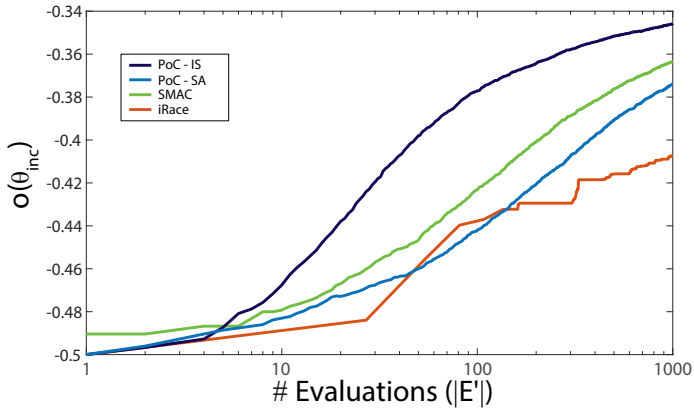


Figure 6.13: Results for the dynamic metaheuristic scheduling scenario.

6.6 Limitations and Future Research

In Section 6.1, we argued for the importance of using all available information to solve the ADP (see also Section 3.4). In particular, we entertained the possibility of improving the data-efficiency of PbC, without sacrificing its “off-the-shelfness”. Subsequently, in Sections 6.2-6.5, we explored *one* possible way of doing so, i.e. our IS approach. In this section, we will discuss the limitations of the latter and how they can be addressed.

Increased/ing overhead: In Section 6.5, we showed experimentally that our IS approach improves data efficiency. As ordinary configurators spend the vast majority of the time collecting experimental data, their data efficiency directly translates to time efficiency (unless computing α is extremely cheap). However, in Section 6.3.2.4, we also discussed that calculating IS estimates (\hat{o}) is computationally more expensive than ordinary sample averages (\bar{o}). To avoid that both notions of efficiency diverge, we guaranteed (by tuning parameter L and N) that the total runtime of PoC-IS, on our platform (see Appendix A.1), is similar/lower than that of the other configurators for the scenarios we considered. However, in other settings, overhead may grow so large that it eventually dominates runtime. In particular, as overhead increases over time ($\mathcal{O}(|E'|)$), this will eventually happen in all lifelong learning settings (e.g. semi-online setting discussed in Section 3.5.3). Noteworthy is that when taking into account advances in computing infrastructure this is not necessarily true. Also, we believe that there is considerable potential to reduce this overhead.

Reliance on overlap: Using IS, we only generalize observations across designs which are “actually” similar (vs. regression models, see Section 6.3.2.5). As a consequence, if candidate designs are not similar (\sim overlap, see Equation 6.3), using \hat{o} will not improve data efficiency w.r.t. \bar{o} (see also Theorem A.4), but only increase overhead (see above). As such, for our IS approach to be beneficial, data-efficiency gains (i.e. overlap) must be sufficient to compensate for the increase in overhead. The three scenarios considered in Section 6.5 fall into this category. However, in other settings, PoC-IS may require considerably more resources than PoC-SA. This forces the user of our PoC to manually distinguish between both settings. Also, as overhead increases over time, PoC-IS may be only beneficial initially. Therefore, a variant switching dynamically between \hat{o} and \bar{o} is interesting future research. As is hybridizing IS with regression models.

Does PoC-IS outperform SMAC? In our experiments, we did not aim to show the superiority of PoC-IS over state-of-the-art black box configurators. Our goal here was merely test whether using IS, instead of SA, indeed results in improved data efficiency. As such, we could have restricted our comparison to PoC-IS vs. PoC-SA. State-of-the-art black box optimizers (FocusedILS, iRace and SMAC) were merely included as a reference,

e.g. to show that PoC-SA is sufficiently competitive that “improving its data efficiency” has practical relevance. While PoC-IS performs clearly better in the selected scenarios, as by our discussion above, it will perform much worse in certain other settings. Also, PoC-IS does not support many of the other features (e.g. parallel execution support) provided by the state-of-the-art. That being said, we believe that none of these limitations are inherent, and that state-of-the-art white box configurators are a realistic goal.

Not very white box... wb-ACP only captures (our IS approach only uses) a small fraction of the information that we believe could, in principle, be usefully exploited. However, we hope that the research described in this chapter, limited as it is, convinces the reader that this is a research track worth exploring.

6.7 Summary

In this chapter, we presented a critical reflection on Programming by Configuration (PbC). In particular, we argued that contemporary configurators limit themselves by abstracting information that can otherwise be exploited to speed up the design process, and described our own work towards addressing this shortcoming.

In Section 6.1, we presented a high-level discussion of the limitations of PbC and how these could be addressed. Here, we argued the main limitation of PbC to be the large computational cost associated with collecting algorithm performance data. We continue to discuss two more specific attributes of PbC that further aggravate this issue: (1) “it requires us to pay this cost up front” since PbC solves the ADP offline. Here, we briefly recapped our general discussion of this issue in Section 3.5 and its relevance in the context of “using PbC to design reusable algorithms” in particular. (2) “it uses the collected data inefficiently”. Here, we argue that in reducing the ADP to an ACP, PbC abstracts information that can be usefully exploited to solve the ADP faster.

In Section 6.2 and onwards, we presented our own work towards addressing (2). Here, we considered solving the ADP by reducing it to a more informative variant of the ACP: the “white box” ACP (wb-ACP). In Section 6.2.1, we put this attempt into a wider perspective: While “white box” approaches to the ADP exist, none offer similar generality, ease of use, control, transparency, etc., i.e. are equally “off-the-shelf”, as “black box” approaches (e.g. PbC). Our interest lies in approaches which are both “white box” and “off-the-shelf”. To this end, we aimed to improve the data-efficiency of existing configurators, while retaining the properties that make them attractive in the first place. In Section 6.2.2, we defined the wb-ACP and discussed the reduction. Finally, in Section 6.2.3, we discussed the relations between the wb-ACP and previously examined formulations of the ADP: the

ACP, subset-ASP and dynamic ASP. In particular, we described generic reductions from these problems to the wb-ACP.

In Section 6.3, we discussed how the information captured by the wb-ACP can be exploited in evaluating the relative performance of candidate designs. First, in Section 6.3.1, we characterized the performance evaluation problem (PEP) and recapped how it is currently being solved in the context of the PbC. Subsequently, in Section 6.3.2, we presented a novel performance estimation technique, loosely inspired by the practice of “off-policy policy evaluation” in the RL community. In Section 6.3.2.1, we introduced the observation underlying our approach: Different candidate designs might, with some likelihood, generate the exact same execution e . The crux is that even though an execution e might have been generated using θ , it might as well (with some likelihood) have been generated using a different θ' . As such, the observed desirability of e does not only provide information about the performance of θ , but also about that of θ' . Subsequently, in Section 6.3.2.2, we introduced the general statistical technique known as Importance Sampling (IS) and showed, in Section 6.3.2.3, how IS can be used to combine all performance observations relevant to a design θ , into a consistent estimator of its performance. In Section 6.3.2.4, we presented an effective procedure for computing this estimate and discussed its computational complexity. Finally, in Section 6.3.2.5, we discussed the merits of using this technique to solve the PEP (vs. alternatives) and illustrated some of these experimentally.

In Section 6.4, we described a white box configurator implementing our novel performance estimation approach. Here, we distilled two top-level design choices around which we structured our description. In Section 6.4.1, we discussed the *choice of incumbent*: Which of the designs explored thus far do we return as the solution at any time? Here, we found that performance estimates alone are insufficient to make this decision. The reliabilities of these estimates, since they may vary strongly, must also be taken into account. In section 6.4.1.1, we discussed measuring the reliability of an IS estimate. In Section 6.4.2, we described the *search strategy* implemented, determining the order in which we explore/evaluate candidate designs.

In Section 6.5, we validated our approach experimentally. To this end, we compared the performance of our white box configurator with that of a baseline of black box configurators, in three different scenarios. In each scenario, we considered a different instance of the ADP and described its reduction to the wb-ACP. In summary, we found that, for the scenarios considered, our configurator required (up to several orders of magnitude) less evaluations than its black box counterparts to obtain similar quality designs.

Finally, in Section 6.6, we discussed the limitations of our IS approach, and how these could be addressed in future research. Here, we, for instance, explained that using our IS approach, in its current realization, is only beneficial in specific settings and that identifying these automatically is crucial to achieve the desired level of off-the-shelfness.

7 | Accidental Complexity Analysis

In Section 1.1, we have argued that, in our opinion, the contemporary practice of designing, analyzing and presenting algorithmic contributions is lacking. In this dissertation, we set out to investigate *how we can do better*. Thus far, we have looked at alternative ways of designing algorithms, in particular, the possibility of (partially) automating this process (answering Q.A), which we have argued to have the potential to not only reduce the manual effort involved in the process, but also to offer more control and transparency about the quality attributes of the resulting design.

In this chapter, we will take a different perspective, exploring a less disruptive angle, where rather than trying to replace the design process, we will attempt to improve the quality (\sim reusability) of the resulting design *post hoc* (answer Q.B). Here, we focus on “simplicity”, which we argue to be an important factor often overlooked in algorithm design, which has a tendency to produce overly complex methods. This chapter is structured as follows: In Section 7.1, we provide some context and motivate our focus on simplicity. In Section 7.2, we introduce Accidental Complexity Analysis (ACA), a research practice targeted at identifying needless complexity. In Section 7.3, we demonstrate this practice, applying it to analyze the presence of accidental complexity in AdapHH, a state-of-the-art selection hyper-heuristic for HyFlex. In Section 7.4, we present a critical reflection on this research. Finally, in Section 7.5, we summarize the content covered in each section.

7.1 Simplicity in Algorithm Design

Often, what makes designing an algorithm worthwhile, is its ability of being reused. Important factors affecting algorithm reusability (see also Section 3.2) are its ability to solve many different problem instances (**generality**) and to do so better, using less resources than other algorithms (**performance**). Therefore, traditionally, algorithm design has (rightfully) focused on the performance and generality of algorithms.

In what follows, we argue for **simplicity** as another important factor often overlooked when designing algorithms. In daily life, simpler “algorithms” (recipes, manuals, etc.) are preferred for many reasons, e.g. they are less error-prone and take less time to understand and execute. One might argue that complexity is not an issue in computer science, as the computer executes the algorithm, not a human. For software in general, this argument can be rephrased as “It does not matter what my code looks like, as the user does not see it anyway”. This is a known fallacy in Software Engineering, as software must be maintained and therefore be maintainable [Lientz and Swanson 1981]. In analogy, algorithms will unavoidably need revision at some point (e.g. to solve other problems, more efficiently, in a slightly different setting, etc.), especially in a research setting. Complex algorithms are difficult to understand, implement, analyze, extend and improve... , i.e. reuse.

Yet, when looking at state-of-the-art algorithmic solutions to various problems, we often find them to be rather complex. The question we ask ourselves in this chapter is: “*Do they need to be?*”. That is, we wish to distinguish between *essential* complexity, which contributes to an algorithm’s ability to solve multiple problems efficiently, and *accidental* complexity which can be eliminated without loss of performance.

In the remainder of this section, we will argue that the complexity found in state-of-the-art algorithmic (especially heuristic) solutions is often not all essential, i.e. simpler alternatives exist, but these are simply not considered in the manual trial & error modus operandi by which these methods are typically designed (graduate student search, see Section 3.3.1). Here, we often start from a vague and inexact solution approach. Implementations on the other hand must be exact. Therefore, when trying to implement this algorithmic concept, we are forced to make further design decisions. We often lack the expert knowledge to do so, prompting us to make these ad hoc, adding accidental complexity. When testing a method, we typically observe various issues in performance, which we subsequently patch. These patches often solve problems only partially, or cause new problems, incrementally adding further complexity. This *spiral of added complexity*, results in methods which might perform well, but have a tendency of being overly complicated.

Remark that some automated approaches (see Section 3.3.2) also have a tendency to produce overly complex designs. A general issue is that complex algorithms tend to greatly outnumber simpler ones; e.g. there are twice as many n -bit programs as $n-1$ -bit programs.

7.2 Identifying Accidental Complexity

In the previous section, we have presupposed that performant algorithmic solutions, especially those of a heuristic nature, have a tendency to be overly complex. In this section, we will argue that it is therefore important to counteract this effect by making a dedicated effort to reduce algorithmic complexity (i.e. simplify the solution approach) post hoc, similar to the process of refactoring in software engineering [Fowler and Beck 1999], which reduces code complexity (i.e. simplifies/improves the program structurally).

Accidental Complexity Analysis (ACA) is the term which we will use to collectively refer to research practices targeted at identifying and eliminating accidental complexity in algorithms. We wish to stress that while this term is novel, ACA itself surely is not. Rather, it is something we argue, we should all be doing, yet rarely gets done thoroughly. In ACA, we are to discriminate accidental from essential complexity, i.e. we are faced with a structural “credit assignment problem” [Minsky 1961]: How is the success of a system due to the contributions of the system’s various components?

Ablation analysis: The specific ACA technique we demonstrate in the next section can be seen as an extensive comparison with a naive baseline. This practice has, in the past, by some authors, been referred to as “ablation analysis”. This due to its similarity to the ablation experiments performed in neurobiology [Flourens 1842], where one removes or deactivates specific brain regions and subsequently observes differences in animals subjected to behavioral tests, allowing one to infer the functions of the removed areas. Similarly, in this ACA technique, the contribution of algorithmic complexity is measured based on the impact of its absence, i.e. we strip an algorithm from its complexity and determine whether and how this impacts performance. We avoid using this “ablation analysis” terminology as to elude confusion with the recent use of the term in the context of parameter sensitivity analysis, targeted at deducing the importance of parameters [Fawcett and Hoos 2016, Biedenkapp et al. 2017]. Instead, lacking a better term, we will just generically refer to it as “ACA”. However, we wish to stress again that ACA is a more general concept, encompassing the gamut of techniques to identify/eliminate accidental complexity.

7.3 Case Study: Accidental Complexity in Reusable Metaheuristics

In this section, we describe how we used ACA, in [Adriaensen and Nowé 2016a], to analyze the presence of accidental complexity in AdapHH, a state-of-the-art selection hyperheuristic for the HyFlex framework.

Table 7.1: The top eight contestants of the CheSC 2011 competition

rank	method	author	score	loc
1	AdapHH	Mustafa Misir	181	2324
2	VNS-TW	Ping-Che Hsiao	134	791
3	ML	Mathieu Larose	131.5	323
4	PHUNTER	Fan Xue	93.25	2849
5	EPH	David Meignan	89.75	957
6	HAHA	Andreas Lehrbaum	75.75	399
7	NAHH	Franco Mascia	75	1089
8	ISEA	Jiri Kubalik	71	1511

7.3.1 Context and Motivation

First, we provide some background and motivation for our case study. The objective of this study was twofold: (1) illustrate ACA experimentally,¹ and (2) provide evidence of accidental complexity in prior art.

Reusable Metaheuristics: In our study, we investigate the presence of accidental complexity in state-of-the-art “off-the-shelf” metaheuristics.² Despite the numerous successful applications of metaheuristics to a wide variety of problems, applying metaheuristics to newly encountered problems remains notoriously challenging. This has prompted research into more readily reusable off-the-shelf metaheuristics. As discussed earlier, simplicity is an important factor contributing to the reusability of an algorithm. Yet, we find that many of these off-the-shelf metaheuristics tend to be highly complex [Mladenović et al. 2016]. A complexity we conjecture to be often accidental, rather than essential because:

- They are designed predominantly manually and therefore susceptible to *the spiral of added complexity* we characterized in Section 7.1.
- To achieve a higher level of generality these methods often resort to metaphorical inspiration (see Section 2.7.2.4) and learning mechanisms (see Section 3.5.2.1), both of which “encourage” the introduction of complexity whose contribution to performance is rarely validated thoroughly.

AdapHH: We further motivate our choice for AdapHH, the selection hyper-heuristic for HyFlex that won the CheSC 2011 competition (see Section 5.2.1.2). Table 7.1 shows the score obtained by, and the complexity³ of each of the top eight contestants. Looking at these results we observe a lack of correlation between complexity and performance. Simpler

¹In Section 5.3.2.2, we already applied ACA to FS-ILS.

²For more information about metaheuristics we refer the reader to Section 2.7.2. Off-the-shelf metaheuristics, selection hyperheuristics in particular, are discussed in Section 2.7.2.5.

³The lines of code (loc) metric was used as a rough estimate and was measured using the cloc tool <http://cloc.sourceforge.net/> (version 1.62).

methods outperform more complex ones, in 15 out of 28 pairwise comparisons, suggesting that high complexity is not essential to perform well on this benchmark. This is further supported by our observations in Section 5.2, where, following a semi-automated design approach, we obtained a simple⁴ selection hyper-heuristic (FS-ILS), which would have won the CheSC 2011 competition. On the other hand, in Section 5.3.2.4, we compared amongst others AdapHH and FS-ILS on the extended HyFlex benchmark set. While overall AdapHH and FS-ILS (without restart) performed similar, AdapHH was shown to generalize better to the three new domains. The main downside of AdapHH is that it is rather complex, which is why we have selected it as the subject of our study.

AdapHH, was previously described in various articles:

- [Misir et al. 2011]: Extended abstract for CheSC 2011, describing the contestant.
- [Misir et al. 2012b]: Full paper, describing and analyzing AdapHH in more detail.
- [Misir et al. 2012a]: Mustafa Misir's doctoral dissertation.

During our literature survey, we have noted inconsistencies in these descriptions, suggesting they consider slightly different versions. The version used in our study is the one which is publicly available.⁵ We also used this version in Section 5.3.2.4, where a brief description of this method can be found. More detailed descriptions of AdapHH's sub-mechanisms can be found in Appendix A.5.

Disclaimer: We are not the first to analyze AdapHH. In prior art, besides comparisons with the state-of-the-art, various mechanisms were shown to influence its behavior (e.g. heuristics are excluded/reincluded, parameter values are updated dynamically, etc.). However, insufficient proof was presented that this behavior is *actually beneficial*. Noteworthy is that [Misir et al. 2012a] did validate the two main mechanisms ADHS and AILLA, replacing them with uniform random selection and various commonly used acceptance criteria, respectively. To some extent, the study presented in the next section reproduces these results, but goes well beyond this rather coarse-grained comparison, looking at the contributions of its various sub-mechanisms.

7.3.2 Empirical Study

In what follows, we apply accidental complexity analysis to AdapHH. Table 7.2 describes the 39 simplifications considered in our experiments. In Section 7.3.2.1, we consider these in isolation, and use experimental results to assess and discuss the contribution of the various sub-mechanisms to AdapHH's overall performance. Subsequently, in Section 7.3.2.2, we discuss the possibility of combining these and present Lean-AdapHH, an implementation combining a subset of these simplifications and show it to be competitive with AdapHH.

⁴216/166 lines of code with/without restart mechanism.

⁵Called GIHH: <https://code.google.com/archive/p/generic-intelligent-hyper-heuristic/>

Table 7.2: The 39 simplifications considered in our experiments

Phase-based Heuristic Set Selection (ADHS)	
noxo:	Crossover heuristics are never selected.
runbestxo:	Perform crossover with best since last reinit.
ph1:	One phase per run (no phase-based updates).
fixpl:	The phase length is constant $pl = 500 \lfloor \sqrt{2N} \rfloor$.
ph1types:	One phase per run (ph1) + utdtypes.
utdtypes:	Update heuristic types after each application.
notypes:	Do not update heuristic types.
notabu:	Heuristics are never excluded.
noqitabu:	No exclusions by the QI tabu criterion.
noextreme:	No extreme exclusion.
notabur:	Relay hybridization is never tabu.
fixd:	Tabu durations d_i are fixed to $D = \lfloor \sqrt{2N} \rfloor$.
noubd:	No upper bound for d_i (also noinfex).
noinfex:	Heuristics are never excluded indefinitely.
Heuristic Selection	
ursall:	Heuristics are selected uniformly from the full heuristic set.
urs:	Heuristics are selected uniformly from the elitist set.
norelay:	Relay Hybridization (RH) is never used.
ronly:	Always use relay hybridization.
urssingle:	Single heuristics are selected as in <i>urs</i> .
urs1relay:	Select the first heuristic in relay hybridization uniformly.
fixa:	Fixed learning rates: $\alpha_r(i) = 0.2, \forall i$
r2impr:	$P_{12} = 0$, i.e. do not select second heuristic in RH from list.
r2list:	$P_{12} = \frac{l_{12}}{l_{12}+1}$, where l_{12} is the number of entries in the list. Otherwise a heuristic is selected as in <i>ursall</i> .
Dynamic Parameter Adaptation	
fixloc:	Fix v_i values to 0.2 (also <i>noosc</i>).
randloc:	Select v_i uniformly from $[0.2, 1.0]$ (also <i>noosc</i>).
osconly:	Modify v_i only using parameter oscillation.
noosc:	No parameter oscillation.
eloc:	u is no random variable, instead $u = \bar{u}$ (mean).
Acceptance and Reinitialization Mechanism	
acceptall:	Accept any proposal (also <i>nostuck</i>).
anw:	Reject all worsening proposals (also <i>nostuck</i>).
fixl:	Fix l to 6.
fixk:	Fix k to 20 ($K = 100$).
nok:	Fix k to 0 ($K = 0$).
nostuck:	The acceptance mechanism never signals stuck.
norestart:	Reinitialization is disabled.
srestart:	Use simplified reinitialization rules described in Appendix A.5.
torestart:	Reinitialize iff stuck during the 1 st half of the run.
contrestart:	Reinitialize whenever stuck is signaled.
nobinit:	Do not perform best-initialization.

7.3.2.1 Single Simplifications

In this section, we assess the contribution of individual mechanisms to AdapHH's performance and identify accidental complexity in the process. To this end, we evaluate the performance of 39 simpler variants of AdapHH. Each variant applying one of the simplifications described in Table 7.2 (\sim eliminating a particular sub-mechanism).

In this study, we are interested in estimating how likely a simplified method is to produce a result worse/better than AdapHH after 10 minutes on the machine used during the CHeSC competition (which is the de facto standard on HyFlex). In addition, we would also like to take into account *how much* better/worse these results are. Therefore, we used the average rank (amongst all 40 methods) of the solution quality obtained, per test on the same instance (μ_{rank}). In our experiments, we run each of these 40 methods, 31 times, for 10 minutes, on all 98 instances of the 9 domains in the extended HyFlex benchmark set (X_{all} , see Appendix A.2). All experiments were performed on the same machine (see Appendix A.1). To avoid bias due to temporal variations in system performance, we ran each batch of 40 tests consecutively in the same process, in a variable random order (to avoid dependencies between runs). Table 7.3 shows the outcome of these experiments. It ranks each method by μ_{rank} . It also provides the p -value obtained by a paired Wilcoxon signed rank test in a pairwise comparison with (unsimplified) AdapHH as a measure of significance and the associated rank correlation $\frac{W}{S}$ as a measure of effect size.⁶ In what follows, we discuss these results in detail, considering a significance level of 1%. Note that many differences are not significant, suggesting our sample size was too small. However, insignificant differences are extremely small ($\frac{W}{S} < 0.05$) and therefore unlikely to be reproducible in slightly different settings (e.g. different platform) anyway.

Phase-based Heuristic Set Selection (ADHS): AdapHH, unlike many other single point hyper-heuristics, uses crossover heuristics. In comparing with **noxo**, we find this to be beneficial. Also, the mechanism managing five candidate solutions to be used as second input in crossover outperforms the straightforward alternative **runbestxo**. AdapHH subdivides a run into multiple phases, each lasting a varying # iterations (pl). At the end of each phase, the elitist set is revised, heuristic types and pl are updated. Surprisingly, **ph1** performed slightly better on average in our experiments, even though this difference was not found to be significant, it leads us to question the contribution of pl adaptation, heuristic types and the tabu mechanism used. **fixpl** performed very similar to AdapHH, which further suggests that the phase length adaptation mechanism does not contribute to performance. **ph1types** and **utdypes** did perform (significantly) better, while **notypes** performed worse. This suggests the use of dynamic types in AdapHH is actually beneficial, however, by updating types only at the end it fails to exploit these benefits fully. Finally, none of the

⁶ W : the test statistic; S : the total sum of ranks

Table 7.3: Comparison of AdapHH to 39 simpler variants

rank	simplification	μ_{rank}	$\frac{W}{S}$	p
1	r2list	17.2964	0.0786	0.0003
2	utdtypes	17.4821	0.065	0.0027
3	torestart	17.5871	0.0617	0.0044
4	ph1types	17.6371	0.0501	0.0206
5	noqitabu	17.7956	0.0385	0.0756
6	fixd	17.8863	0.0324	0.1349
7	eloc	17.9249	0.0355	0.1017
8	noinfex	17.9279	0.0308	0.1555
9	noubd	17.9284	0.0368	0.0894
10	fixa	17.956	0.0316	0.144
11	srestart	17.9865	0.0303	0.1616
12	noosc	18.0276	0.0285	0.1887
13	notabu	18.1423	0.0117	0.589
14	fixl	18.2217	0.0109	0.6148
15	ph1	18.2625	0.0064	0.7676
16	fixk	18.2702	0.0022	0.9182
17	fixpl	18.3151	0.0036	0.8668
18	unsimplified AdapHH	18.3987	1	1
19	notabur	18.4451	0.0104	0.6301
20	nok	18.757	0.0302	0.1626
21	notypes	18.8165	0.0376	0.0829
22	noextreme	18.8367	0.0356	0.1003
23	osconly	19.4689	0.0833	0.0001
24	r2impr	19.564	0.101	0
25	fixloc	19.592	0.0932	0
26	noxo	19.6545	0.0881	0
27	randloc	20.5053	0.1646	0
28	runbestxo	20.8638	0.1923	0
29	nobinit	20.977	0.211	0
30	urs1relay	21.5767	0.246	0
31	contrestart	21.8003	0.2694	0
32	ronly	21.8183	0.2442	0
33	nostuck	23.1508	0.3283	0
34	norestart	23.4382	0.3551	0
35	urssingle	23.8746	0.3949	0
36	anw	25.6425	0.4733	0
37	norelay	26.2887	0.5122	0
38	urs	29.7704	0.7033	0
39	ursall	31.6582	0.7763	0
40	acceptall	32.4543	0.7956	0

simplifications to the tabu mechanism (**notabu**, **noqitabu**, **noextreme**, **notabur**, **fixd**, **noubd**, **noinfex**) were found to have a significant impact on performance, which strongly suggests that the (highly complex) tabu mechanism does not add any value.

Heuristic Selection: Both **ursall** and **urs** perform categorically worse than AdapHH, implying AdapHH's heuristic selection mechanism has a significant contribution to its performance. AdapHH combines two mechanisms for selection heuristics: Single Selection (SS) and Relay Hybridization (RH). Which is to be credited for this performance? The fact that **norelay** and **ronly** perform significantly worse, suggests both. Do note **ronly** clearly outperforms **norelay**, suggesting RH deserves credit in particular. Next, we take a look at both mechanisms individually. SS is already rather straightforward and the even simpler alternative **urssingle** performed significantly worse. RH, on the other hand, is rather complex, and different mechanisms are used to select the first and (possibly) second heuristic. The learning automaton used to update the selection probabilities of the first heuristic contributes to performance, as **urs1relay** performs clearly worse. However, using heuristic specific, adaptive learning rates $\alpha_r(i)$ in this scheme does not (**fixa**). For selecting the second heuristic, AdapHH combines list-based selection, used 25% (P_2) of the time, and a greedy selection scheme, used otherwise. Which of these two contribute to performance? Our results suggest the former definitely does as **r2impr** performs significantly worse. However, **r2list** performed significantly better, suggesting that the somewhat cumbersome greedy selection scheme is not desirable.

Dynamic Parameter Adaptation: AdapHH is one of the few hyper-heuristics for Hy-Flex modifying the α and β values for low-level heuristics. Most retain the default (0.2) values. **fixloc** performs significantly worse, implying modifying these parameters is beneficial. Furthermore, simple randomization of these values (**randloc**) performs worse as well. In AdapHH two mechanisms update these parameters: One mechanism updates α and β after each application, the other oscillates these parameters when the search is stuck. Which of these contribute to performance? Our results suggest the former does as **osconly** performs significantly worse. On the other hand, **noosc** seems competitive with AdapHH, suggesting the contribution of parameter oscillation is minor (if any). Parameter updates after each application are stochastic. **eloc** is competitive with AdapHH suggesting that the stochasticity of these updates does not contribute to performance.

Acceptance and Reinitialization: AdapHH uses a relatively complex acceptance mechanism (AILLA) to decide whether or not to accept a proposal. The necessity of an acceptance mechanism to control diversification in AdapHH clearly shows in our experiments (**acceptall**). An extensive comparison using alternative acceptance criteria was already performed in [Misir et al. 2012a] suggesting AILLA is an important factor in the perform-

ance of AdapHH. We only include the most straightforward one **anw** in our study, which was outperformed as expected. AILLA maintains a list of l threshold levels and uses two patience thresholds, one to decide when to accept worsening solutions (k), another to decide when to increase the threshold level (K). Both l , k and K are adapted dynamically. In our experiments **fixl** and **fixk** (and **nok**) did not perform significantly worse, suggesting the dynamic adaptation of these does not add much value and reasonable defaults can be chosen in a general setting.

A unique feature of AdapHH is that the acceptance mechanism informs the reinitialization mechanism. If the threshold exceeds its maximum value, the acceptance mechanism signals the search is stuck, activating the reinitialization or parameter oscillation mechanism. The fact that **nostuck** performs significantly worse suggests the importance of this mechanism. Combining this with our earlier observation that parameter oscillation adds little value, our results suggest the reinitialization mechanism contributes significantly to AdapHH's overall performance. The similarly poor performance of **norestart** confirms this. The exact restart criterion was difficult to deduce as various (superfluous?) conditions were checked throughout the code. **srestart** replaces all of these with a combination of rules we believe to be functionally equivalent. The similar performance of **srestart** seems to confirm this belief. Furthermore, our results for **torestart** suggest potential further simplification. All these mechanisms disable restart in the second half of the run (perhaps earlier). When restart is disabled, best-initialization is performed. The significantly worse performance of **contrestart** and **nobinit** suggests it is important to do so.

7.3.2.2 Combining Simplifications

In the previous section, we have seen that AdapHH can be simplified in various ways without loss of performance. In this section, we consider the potential of combining these simplifications. Not all simplifications can be combined in a meaningful way. In our study, we consider two types of (binary) relationships between simplifications prohibiting this:

mutual exclusiveness ($a \leftrightarrow b$): If simplifications a and b are mutually exclusive, a combination cannot have both a and b . This relation is irreflexive and symmetric.

subsumption ($a \leftarrow b$): If a simplification a subsumes b , the presence of a renders b redundant, e.g. b eliminates the mechanism a simplifies. This relation is reflexive, antisymmetric and transitive.

Figure 7.1 shows the relationships between the simplifications in our study. The simplifications that in isolation were not found to have a significant (negative) impact on performance in the previous section are underlined. Two simplifications cannot be combined if and only if there exists a path of 1-headed arrows or a 2-headed arrow directly

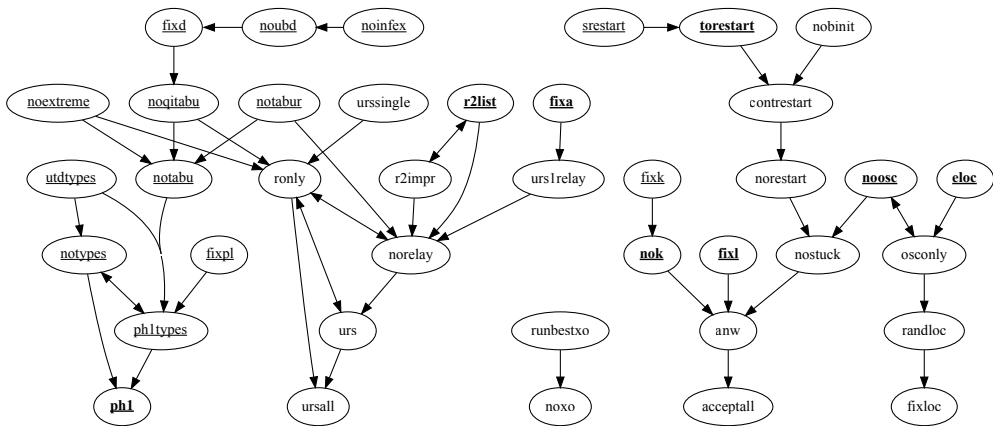


Figure 7.1: Relationships between the 39 simplifications considered

Which raises the question “when is one set of simplifications A simpler than another B ?” We use the following criterion: One set of simplifications A is at least as simple as another B if and only if $\forall s \in B, \exists s' \in A : s' \leftarrow s$, i.e. for each simplification in B , A contains one that subsumes it. Note that, in general, this criterion imposes a partial rather than a total order over A_{simpler} . Finally, remark that simplifications when combined, can perform significantly worse, even though they individually do not and vice versa. In this part of our study, we make the simplifying assumption that this is never the case. Under this assumption, we have a unique set of simplifications L , for which the resulting method is at least as simple as that of any other competitive with AdapHH (subsuming all underlined nodes), i.e. $L = \{\text{ph1, fixa, r2list, nok, fixl, torestart, eloc, noosc}\}$.

We have reimplemented this simplified variant, which we will refer to as Lean-AdapHH, from scratch. We did so for various reasons. First, the publicly available version of AdapHH is not its most elegant possible implementation, includes overly complex or even dead code, and was in need of refactoring. Second, while our simplifications eliminate the effect of mechanisms on algorithm behavior, their code would still present and statistics which are no longer required would still be computed, introducing unnecessary overhead. Finally, in reimplementing Lean-AdapHH, we validate our understanding of AdapHH, the correctness of our simplifications, and the reproducibility of our results.

Table 7.4: Comparison of AdapHH to Lean-AdapHH

method	% >	$\frac{w}{s}$	p	>	>>
Lean-AdapHH	53.765	0.2134	0	60	30
AdapHH	39.3737	0	1	30	8

To verify whether this effort was successful, we have compared both AdapHH and Lean-AdapHH (same process, random order) 31 times on each instance of the extended HyFlex benchmark set. Table 7.4 shows the outcome of this experiment. As before, it shows the rank correlation and the p -value of the Wilcoxon signed rank test. Instead of μ_{rank} , it shows the percentage of tests each method performed better (% >), which is more intuitive and informative in this case. In addition, we show the number of instances (out of 98) each of the methods obtains the better median performance (>) on and how many of these were found to be significant (>>). In summary, we find that Lean-AdapHH performs clearly better than AdapHH. However, this was not our main objective. Rather, we set out to simplify AdapHH (without loss of performance). This can be considered a success, as the resulting implementation counts a mere 288 lines of code (vs. 2324 originally). Our code was in no way compressed to cheat this metric, nor does it rely on third-party libraries (besides HyFlex), and readability was always prioritized over brevity/efficiency. We do not describe our implementation in detail here. However, it is isomorphic to AdapHH, after applying the eight simplifications in L . Also, its code is made publicly available⁷ and is well-documented.

7.4 Critical Reflection

In this Section, we discuss some of the limitations of our case study and ACA in general.

Potential bias: Remark that we “designed Lean-AdapHH” based on the observations we made regarding the relative performance of 40 variants of AdapHH on the extended HyFlex benchmark set (X_{all}). While we did not optimize Lean-AdapHH’s performance on X_{all} directly,⁸ e.g. as we did with FS-ILS on X_{chesc} in Section 5.2, we nonetheless cannot exclude the possibility that this biased our view on the relative performance of Lean-GIHH and GIHH. Furthermore, since we used all available instances, we were unable to perform an unbiased comparison post hoc. In retrospect, we should have split X_{all} into two, used one for ACA, and the other for validation.

⁷Called Lean-GIHH: <https://github.com/Steven-Adriaensen/Lean-GIHH>

⁸Lean-AdapHH was not one of these variants, and our objective was reducing accidental complexity.

Empirical: As discussed in Section 3.4.2, we cannot, based on empirical information alone, show that one algorithm performs worse/better than another. As the ACA technique considered is highly empirical, it inherits this limitation. In our study, we eliminated various “adaptive mechanisms”, originally introduced by the author for the purpose of achieving a higher level of generality, based on the observation that they did not significantly improve performance on 98 specific problem instances. We have no guarantees that these mechanisms are not beneficial on other problems, not considered in our study. That being said, we also have no evidence that they are. In arguing that, given this uncertainty, one should exclude them, we essentially apply the parsimony principle (a.k.a. Occam’s Razor) to algorithmics. Clearly, if ACA is performed by the author, as it should be, he/she may elect to gather additional evidence for their merits, as opposed to excluding them.

Scalability: In our study, we considered 39 candidate simplifications of AdapHH. In Section 7.3.2.1 we compared AdapHH, to 39 simpler variants, each implementing one of these. In Section 7.3.2.2, we argued that some simplifications can be combined, giving rise to a large space of simpler variants. Due to combinatorial explosion, a full comparison, was no longer tractable, leading us to select a combination using a rather crude heuristic. While we will need to resort to heuristics in some cases, we can reduce the resource requirements of ACA in various ways. First, one may allocate resources non-uniformly. For instance, use a statistical racing scheme (\sim those discussed in Section 4.1.2, p. 111) and test only simpler variants which are not yet known to perform significantly worse/better. Also, in some cases, data-efficiency could be improved by using the performance estimation technique we introduced in Chapter 6. Finally, if we merely wish to find the simplest variant, we could at any time eliminate variants more complicated than the simplest one found thus far. More generally, tool support for ACA would be interesting future research.

7.5 Summary

Why bother sharing (and publishing) your algorithms, if they will not be reused by the rest of the community? By describing overly complex solutions (or solutions overly complex) we make reuse unnecessarily difficult, waste space in research journals and valuable time of our peer researchers. In this chapter, we argued for the importance of representing algorithmic contributions as simple as possible and proposed an analysis technique to help doing so. The research described in this chapter was published in [Adriaensen and Nowé 2016a]. In what follows, we briefly summarize the content covered in each section.

In Section 7.1, we argued for simplicity as an important factor contributing to the re-usability of an algorithm. Yet, when looking at state-of-the-art algorithmic solutions to various problems, we often find them to be rather complex. Furthermore, we conjectured

this complexity often not to be essential, but rather an artifact of the “graduate student search” approach by which these methods are designed, which has a tendency to produce methods which (given a lot of manual effort) may perform well, but are overly complex.

In this dissertation, we have examined two different ways of addressing this problem. In previous chapters, we looked into alternative, semi-automated design approaches, in particular programming by configuration, which not only reduce the manual effort involved in the process, but also offer more control and transparency about the quality attributes (e.g. simplicity) of the resulting design. This can be viewed as the “proactive” approach where we avoid introducing accidental complexity in the first place. However, (1) we cannot expect the entire community to suddenly change the way they have been designing algorithms for decades. (2) despite its many shortcomings, we cannot ignore the numerous successes of the manual design approach. (3) design automation does not address the existing accidental complexity in prior art. (4) contemporary automated techniques may still produce overly complex designs. Therefore, in Section 7.2, we explored a “reactive” strategy. Here, rather than trying to eliminate the source, we took the design (approach) as a given and identify/eliminate accidental complexity through a post hoc analysis. A practice we call Accidental Complexity Analysis (ACA). Here, we also characterized the specific ACA technique, a.k.a. “ablation analysis”, we used in the remainder of this chapter.

In Section 7.3, we conducted a case study, applying ACA to AdapHH, a state-of-the-art selection hyper-heuristic for the HyFlex framework. The objective of our study was twofold: (1) illustrate ACA experimentally, and (2) provide evidence of accidental complexity in prior art. In Section 7.3.1, we first provide some background and motivate our choice of AdapHH as the subject of our study. In Section 7.3.2, we describe the empirical study itself, where we considered 39 possible simplifications of AdapHH, each surgically eliminating a specific sub-mechanism. In Section 7.3.2.1, we considered these in isolation, and identified various sub-mechanisms that do not contribute significantly to AdapHH’s overall performance. On the other hand, we have validated the contribution of various other mechanisms. Subsequently, in Section 7.3.2.2, we discussed the possibility of combining these simplifications and presented Lean-AdapHH, a simplified reimplementation of AdapHH, reducing its code complexity with a factor of eight, without loss of performance.

Finally, in Section 7.4, we presented a critical reflection on our own research. Here, we discussed the possible influence of bias and variance on our assessment of Lean-AdapHH’s performance, as well as the limited scalability of the ACA technique used in our case study.

8 | Conclusion

In this chapter, we conclude this dissertation. Here, we highlight certain aspects of this work and offer some broader perspectives. It is structured as follows: First, in Section 8.1, we give an overview of the research performed in this dissertation and summarize our main findings. Subsequently, in Section 8.2, we discuss what we regard to be some of the main limitations of this work and characterize possibly interesting lines of future research.

8.1 Research Questions (revisited)

In this section, we revisit the research questions we have formulated in Section 1.2.2. For each of these questions, we summarize the research we have performed in the context of answering it and formulate a concrete answer. A more complete survey of the content covered in this dissertation, can be found in the summaries at the end of each chapter.

Disclaimer: Two parts, two kinds of research. Remark that we answered some of the questions below, partially (and R.A.1 even entirely) in the first part of this dissertation (i.e. Part I). As discussed in Section 1.3.1, the “research” performed in part I is of a more “descriptive” nature. Our claims here are frequently only supported by informal arguments, as supposed to, e.g., experimental studies, as in Part II. Therefore, albeit carefully justified, we wish to stress that our answers below are to some extent inherently subjective, reflect our personal opinion, and we do not claim that they hold the absolute truth. That being said, we also do not wish to make such claims about our research in part II either.

Bottom line: These are *our answers* which are based on what we have learned through the different kinds of research we have performed in the scope of this dissertation.

Q.A. How can computers help us in designing reusable heuristics?

This was the central question we investigated in this dissertation (see Chapters 3-6). We will postpone our answer to this question until the end. First, we discuss how we approached this question, and answered the three more specific questions we examined in this context. Noteworthy is that we studied this question within the standard framework of algorithmics, i.e. we viewed algorithm design itself as “a problem”, which we refer to as the Algorithm Design Problem (ADP). In the ADP, we are to find the best algorithm to do something (see Section 3.1). Here, we viewed “the problem of designing reusable heuristics” as a special case thereof, where we are to find “the most reusable heuristic”.

Q.A.1. What makes one heuristic more reusable than another?

Various factors affect the reusability of a heuristic (see Section 3.2). An important factor is its performance, i.e. people tend to use heuristics that perform well, or better than others (see Section 3.2.1). The performance of a heuristic can be further subdivided in:

- **efficiency:** Its ability to solve a problem using little resources.
- **efficacy:** Its ability to obtain good solutions to a problem.
- **generality:** Its ability to solve a wide range of different problems.

However, performance is not all that matters (see Section 3.2.2). In particular, we argued for **simplicity** as another important factor affecting the reusability of a heuristic (see Section 7.1). Simpler heuristics are easier to understand, implement, analyze and improve.

In summary, our objective was to design simple heuristics which require little resources to obtain high-quality solutions for a wide range of problems. However, these objectives (simplicity, efficiency, efficacy and generality) are commonly believed to be strongly conflicting. This has prompted us to more closely examine the tension between our objectives (see Section 3.2.3), i.e. does such thing as “the most reusable heuristic” exist?

Here, we found that *theoretically* ADPs where the tension between some of these objectives is indeed inherent and strong can be shown to exist. On the other hand, we did not find any evidence that this is, in fact, the case for ADPs actually encountered *in practice*. In particular, for the ADPs we considered in our case studies in Part II.

Q.A.2. Which of the existing approaches is best suited to design reusable heuristics?

To answer this question, we looked at a wide variety of techniques which are currently being used to design algorithms and discussed what makes them in our opinion more/less suitable in the context of the design of “reusable heuristics”:

Manual vs. Automated: To date, heuristics are commonly designed following a manual, ad hoc, trial & error process (see Section 3.3.1). Despite many successful applications, we argued that this modus operandi is tedious, time-consuming, costly and

frequently results in heuristics with poor reusability. This led us to explore alternatives. In particular, we looked at automated design techniques which we believe to show great potential to address the aforementioned limitations → **automated**.

Fully vs. Semi-automated: In automated design, we further distinguished between fully and semi-automated approaches, giving rise to a rough classification of design approaches based on the role the human designer is envisioned to play in the process (see Section 3.3). In fully-automated approaches, all decisions w.r.t. “how to solve the problem” are left to the computer (see Section 3.3.2: program synthesis, genetic programming, neural Turing machines, etc.). Semi-automated approaches differ in that they try to maximally exploit any knowledge the human designer might possess w.r.t. solving the problem, to constrain the design space a priori (see Section 3.3.3: e.g. algorithm selection, algorithm scheduling, algorithm configuration, optimizing compilers, etc.). We focus on the latter because (A) fully-automated approaches are often computationally intractable, and (B) even in expert-knowledge poor areas like heuristic optimization, we can exclude certain designs a priori → **semi-automated**.

Analytical vs. Empirical: In Section 3.4, we further distinguished between pure analytical and empirical approaches, based on whether the designer uses only a priori information (\sim what is known beforehand) or also a posteriori information (\sim experience obtained through experimentation). By nature, what can be said, a priori, about the relative performance of two heuristics is limited, prompting the designer to collect additional information, a posteriori, i.e. actually test candidate heuristics. → **empirical (a.k.a. simulation-based)**.

Offline vs. Online: In Section 3.5, we further categorized empirical approaches based on whether they solve the ADP offline (“design before use”, see Section 3.5.1) or online (“design during use”, see Section 3.5.2). In Section 3.5.3, we argued that while the offline setting is often impractical, the true online setting is overly restrictive, and in many practical settings what we call a “semi-online” (“design during spare time”) approach suffices. Furthermore, we argued that in this setting, “anytime” (see Section 2.6.4) and not “online” is the property of interest → **anytime (offline)**.

Per-set vs. Per-input vs. Dynamic Algorithm Selection: In Chapter 4, we discussed three generic approaches in more detail, which we regard as being prototypical for semi-automated design. These can be viewed as solving the ADP by reduction to one of three variants of the algorithm selection problem:

- Per-set Algorithm Selection Problem (set-ASP, see Section 4.1)
- Per-input Algorithm Selection Problem (input-ASP, see Section 4.2)
- Dynamic Algorithm Selection Problem (dynamic ASP, see Section 4.3)

For each of these problems, we presented a formal definition,¹ discussed the reduction and described concrete algorithmic techniques for solving them. We also illustrated some of these experimentally, by using them to design a sorting algorithm. We found that the set-ASP solvers offer a level of generality, ease-of-use, control, transparency, anytime and asymptotic performance, etc. that cannot be found in the techniques which are currently used for solving the input-ASP and the dynamic ASP (see Section 4.4.2). → **Design by Per-set Algorithm Selection.**

In summary, in Part I, we argued the “design by per-set algorithm selection” (set-ASP) approach to be, to date, one of the most suitable, general approaches for designing reusable heuristics. Following this approach, the human designer first specifies a set of candidate algorithms and a set of representative training instances. The computer subsequently tests the algorithms on these instances and determines which one, on average, performs best.

In Chapter 5 (Part II), we conducted a case study investigating the suitability of this approach experimentally. Here, we considered one specific instance of the set-ASP approach which we refer to as Programming by Configuration (PbC, see Section 5.1). In PbC, one solves the ADP by reducing it to an Algorithm Configuration Problem (ACP) which is subsequently solved using tools known as algorithm configurators. In Section 5.2, we used PbC to design a metaheuristic [Adriaensen et al. 2014a]. Subsequently, we analyzed the reusability of the resulting design: FS-ILS [Adriaensen et al. 2014b] (see Section 5.3.2). In particular, we investigated its generality [Adriaensen et al. 2015] (see Section 5.3.2.4), showing it to be competitive with the state-of-the-art across 98 instances, 68 of these were “new”, i.e. not used during FS-ILS’s design. These instances were taken from nine different combinatorial optimization problems, 30 were taken from three “new” domains.

Q.A.3. How can we further improve this approach?

The ease with which PbC can be applied to a wide variety of different ADPs arguably makes it one of the most “off-the-shelf” solutions to automated algorithm design thus far. However, it also has shortcomings. In Chapter 6, we presented a critical reflection on PbC (and the set-ASP approach in general), discussing its limitations and presenting our own work towards addressing these. Arguably, the main limitation of PbC is that it requires a large amount of computational resources. For instance, we allocated roughly one CPU year to the design of FS-ILS (which took the 30 CPUs used less than two weeks). The vast majority of these resources are spent evaluating the average performance of the candidate designs which typically involves executing each of them numerous times (e.g. on different training instances). Reducing the number of executions needed (\sim data-efficiency) is as

¹One contribution, we wish to highlight here, is that we presented the first formal definition of the general dynamic ASP [Adriaensen and Nowé 2016b] as a natural extension of the static ASP [Rice 1976].

such critical. In contemporary configurators, this is mainly achieved by cleverly allocating executions as to avoid wasting resources in evaluating suboptimal designs. In the remainder of Chapter 6, we explored a different (complementary) approach. It is based on the observation that configurators treat the performance of a design, on a given problem instance, as a “black box” function mapping problem instance and design to some real cost value. In doing so, they abstract the fact that this mapping is a consequence of algorithm execution, i.e. design decisions affect the execution on a problem instance in a particular way, which in turn relates to the observed performance. Information which we conjectured could otherwise be used to further improve the data efficiency [Adriaensen and Nowé 2016b]. To this end, we proposed a novel performance estimation technique exploiting this information to generalize performance observations beyond the designs used to obtain them [Adriaensen et al. 2017]. To validate this approach experimentally, we described a “white box” configurator implementing our technique and showed that it indeed improves the data efficiency (up to several orders of magnitude) in three selected scenarios.

Answer Q.A: So, how can computers help us in designing reusable heuristics? Today and in the future? Recall that heuristics are non-exact problem-solving procedures that, despite the lack of relevant theoretical guarantees they provide, often perform very well in practice (see Definition 2.30). This lack of knowledge about the performance of a heuristic makes it intrinsically difficult to predict, *a priori*, whether a given heuristic will perform well, better/worse than another, etc. This, in turn, has a profound impact on how these heuristics are designed: Lacking analytical knowledge, the designer must resort to empirical knowledge, i.e. try out various alternatives to find one that works well. To decide which alternatives to try (first), the designer relies heavily on intuition and experience.

All things considered, it is our belief that, to date, the main role computers can/should play in the design of heuristics is automating the evaluation of alternative heuristics. Put differently, we advocate a semi-automated approach where the human designer proposes candidate heuristics and the computer tries them out to determine which one of these performs best (\sim set-ASP approach). While this automated evaluation is to date still time-consuming, it is time the human researcher can spend on other activities.

In the future, we predict that the design of reusable heuristics will

become increasingly automated: We believe that future advances in computing architecture and AI techniques will continue to reduce the reliance on the human designer to *a priori* constrain the design space. Furthermore, we envision the emergence of increasingly sophisticated computer-aided programming tools which seamlessly integrate these high levels of automation in the design process.

happen “in the large”: Currently, individual (or small groups of) researchers, each define their own design spaces and use their own limited computational resources to identify

designs suited for a particular use case. In the future, we believe that we will evolve to a system (\sim the one envisioned in [Swan et al. 2015] for metaheuristics) where

1. Researchers add algorithmic contributions to a shared executable repository.
2. A shared computation infrastructure continuously evaluates candidate designs and identifies the state-of-the-art.

That is, we envision that both the design space definitions and computational resources will be shared on a community level towards achieving the common goal of finding ever more reusable algorithms.

Q.B. How can we improve the reusability of heuristics post hoc?

While it is our belief that this collaborative, increasingly automated approach is the future, it is somewhat naive to expect that this change will happen overnight. Therefore, we also explored an alternative approach. Here, rather than trying to replace/improve the design process, we investigated the possibility of improving the reusability of the resulting design *post hoc*. This approach has the benefits of (1) being less disruptive, (2) being applicable to improve the reusability of existing designs independently of how they were obtained.

In particular, we focused on 'simplicity', a factor we argued to be often overlooked when designing algorithms, which consequently tend to be overly complicated (see Section 7.1). To counteract this effect, we advocated Accidental Complexity Analysis (ACA), a research practice targeted at identifying excessive complexity in algorithms [Adriaensen and Nowé 2016a]. Here, we described a simple post hoc analysis technique targeted at identifying ways in which a method can be simplified, without loss of performance (see Section 7.2); and demonstrated its use(fullness) by applying it to a state-of-the-art metaheuristic: AdapHH (see Section 7.3). This analysis gave us a deeper insight into the relative contributions of AdapHH's various sub-mechanisms to its overall performance, enabling us to reduce its code-complexity by a factor of eight, without loss of performance.

8.2 Limitations and Future Research

In the remainder of this chapter, we briefly discuss a few important limitations of our own research and suggest how these could be addressed in future research.

Our focus on performance: In this work, we have argued that while performance is important, "it is not the only thing that matters" (see Section 3.2.2). In particular, we argued for the importance of simplicity. In contrast, in our discussion of design techniques and in their applications in our case studies, we always considered optimizing average-case performance as our sole objective. For example, the only action taken to "encourage"

simplicity in the design of FS-ILS, was including simpler variants of candidate designs in our design space. While necessary, this alone is insufficient to avoid introducing accidental complexity. Future research could look at how to integrate Occam's razor in set-ASP approaches (e.g. PbC). One option would be to solve the ADP as a Multi-Objective (MO) problem (i.e. performance and simplicity). For instance, PbC using MO configurators (e.g. S-Race [Zhang et al. 2016], MO-ParamILS [Blot et al. 2016]).

Our focus on sequential computation: Historically, computing systems have tended to get faster at an exponential rate (\sim Moore's law). Software performance automatically scaled along, without requiring any additional efforts from the developer. Currently, the situation is no longer as simple [Sutter 2005]: Contemporary systems, rather than getting faster and faster, are able to do more and more work in parallel. To take advantage of these (increasing) parallel processing capabilities, computations must be decomposed/able into a set of interdependent tasks that can be executed efficiently in parallel across multiple cores and/or across multiple networked machines. In this context, the "parallelism" of an algorithm, i.e. its ability to be executed more efficiently in parallel, is an important quality attribute. While this parallel trend is not exactly new, to date, research in algorithmics still often overlooks parallelism. This is a topic which also did not receive the attention it arguably deserves in the scope this dissertation. Nonetheless, most of the techniques we described are readily applicable to the design of parallel algorithms; e.g. in the ASP, one could consider a set of parallel algorithms as algorithm space. That being said, prior art exists specifically targeted at **the design of parallel algorithms**, e.g. parallel algorithm portfolios [Gomes and Selman 2001, Yun and Epstein 2012, Lindauer et al. 2015], which we did not cover extensively in this work. Finally, parallelism is also important at the meta-level, i.e. we wish to **exploit parallel processing capabilities in solving the ADP**. Most contemporary configurators (e.g. GGA, iRace, ParamILS and SMAC) natively support fine-grained parallelization strategies. One could apply similar strategies in our white box configurator (see Section 6.4). Note that the computation of IS estimates can be efficiently parallelized as well (see Section 6.3.2.4).

Our manual approach to the meta-ADP: In this dissertation, we looked at how to design algorithms for solving a given computational problem (target problem), automatically. To this end, we treated "the problem of designing algorithms" itself as a computational problem (i.e. the ADP) and continued to describe various "algorithms for designing algorithms" (i.e. ADP solvers). The attentive reader may wonder: If the ADP is just another computational problem, why then, did we not **apply** some existing **design automation** technique (e.g. PbC) **to design ADP solvers**? Why not consider the ADP itself as target problem? For instance, in Section 6.4, we described a white box configurator which we designed following a manual "graduate student search" approach (see Section 3.3.1). In-

deed, “how to best solve the ADP” (meta-ADP) is an ADP itself, and in a sense “we failed to practice what we preach”. Our excuse? Short answer: The limitations of contemporary automated design approaches. A particular issue with current simulation-based approaches (e.g. PbC) is that they require us to actually solve numerous instances of the target problem. This becomes intractable when candidate solvers take a long time to do so. For example, let us consider a target problem whose instances can be solved in 3.6 seconds, and an ADP solver which requires 1000 evaluations to provide a good design. Solving this ADP instance as such requires 3600 seconds (one hour). Applying the same ADP solver to the associated meta-ADP would take 3600000 seconds (41-42 days). Addressing this shortcoming is crucial to solving the meta-ADP (and many other ADPs) automatically. One promising approach uses **surrogate ADPs**: Instead of solving instances of the actual ADPs of interest to test a candidate ADP solver, we could solve target problems which have similar properties, but are easier. Recently, [Dang et al. 2017] proposed this strategy in the context of the ACP, and demonstrated its use in the automatic configuration of iRace. While constructing surrogates is expensive, it is a one-time expense since they can be reused for any future benchmarking purposes. Recently, [Eggensperger et al. 2018] derived surrogates for all instances in the AClb.

We did not provide tool support: In this dissertation, we have investigated automated algorithm design from an algorithmic perspective, i.e. we examined (1) how one can formulate the ADP as a computational problem and how to solve the resulting problem automatically. However, to make this (semi-)automated design approach practical, we also need (2) tools (libraries, programming languages, etc.) that facilitate this reduction and seamlessly integrate it into the human designer’s workflow. Examples of prior art in (2) are the programming language extensions which were proposed in [Ansel et al. 2009] and [Hoos 2012] allowing the user to specify open design choices and alternative decisions in code, i.e. to effectively program a design space, rather than a single algorithm instance. While we performed some research in this area ourselves, it was not our focus and we have not detailed these efforts in this work.

In the grand scheme of things...

As discussed in Section 1.1, our research was geared towards addressing a wide variety of issues that, in our opinion, negatively impact algorithmic research. However, in the scope of this dissertation, we have merely touched the proverbial tip of the iceberg on numerous topics. In particular, our work towards addressing specific issues was often restricted to high-level discussions, proofs of concept and validation thereof in the context of small-scale case studies. Despite these limitations, it is our belief that this work has helped raising awareness for these issues and that the presented perspectives and suggested approaches will facilitate addressing these shortcomings in the future.

A | Appendix

A.1 Benchmark Environment Description

All experimental results we have reported in this dissertation were obtained using the same machine. While we have actively tried to avoid, where possible, that the specifics of the environment (machine, operating system, etc.) in which our experiments were executed, influenced the outcome thereof, some of the metrics reported in this dissertation were nonetheless platform-dependent. Below we characterize the environment in which we have executed our experiments:

```
name: 'Pluisje'
location: AI-lab
administrator: Frederik Himpe
architecture: cluster
processor: Intel Xeon E5320 (clock speed: 1.86GHz)
memory: 8GB RAM
operating system: Scientific Linux 5.5 (64-bit)
```

We have coded all our experiments in the Java (openjdk version 1.8.0_151) programming language. Also, the code used in our experiments is publicly available¹

¹<https://github.com/Steven-Adriaensen>. More specific references and references to third-party software can be found in the main text.

A.2 The HyFlex Benchmark Set

In various experiments, we considered evaluating the performance of selection hyper-heuristics across multiple domains. To facilitate this evaluation, we used the HyFlex framework [Ochoa et al. 2012], which provides problem instances and problem-specific components for six different problem domains (see Section 5.2.1.2). HyFlex was used to support the CHeSC 2011 competition, and results obtained by its contestants were used as a baseline in some of our experiments. Also, in [Adriaensen et al. 2015], we extended this framework with three additional problem domains (see Section 5.3.2.4). As such, in different experiments, we considered different subsets of problem instances. They are:

X_{prior} : The 40 instances which were made available prior to the CHeSC 2011 competition.

X_{chesc} : The 30 instances which were used in evaluating the performance of contestants during the CHeSC 2011 competition.

X_{old} : The 68 instances which are currently provided by HyFlex (version 1.0, which was released after the CHeSC 2011 competition).

X_{new} : The 30 instances of the three domains which we have added ourselves.

X_{all} : All 98 instances in the extended HyFlex benchmark set.

Below we list all the instances in X_{all} , per problem domain, and indicate in which of the aforementioned subsets they are included. In HyFlex, within a domain, instances are identified by an integer (index), which is specified in the first column of each table.

Maximum Satisfiability (SAT)					
index	$\in X_{\text{prior}}$	$\in X_{\text{chesc}}$	$\in X_{\text{old}}$	$\in X_{\text{new}}$	$\in X_{\text{all}}$
0	1	0	1	0	1
1	1	0	1	0	1
2	1	0	1	0	1
3	1	1	1	0	1
4	1	1	1	0	1
5	1	1	1	0	1
6	1	0	1	0	1
7	1	0	1	0	1
8	1	0	1	0	1
9	1	0	1	0	1
10	0	1	1	0	1
11	0	1	1	0	1

Bin Packing (BP)					
index	$\in X_{\text{prior}}$	$\in X_{\text{chesc}}$	$\in X_{\text{old}}$	$\in X_{\text{new}}$	$\in X_{\text{all}}$
0	1	0	1	0	1
1	1	1	1	0	1
2	1	0	1	0	1
3	1	0	1	0	1
4	1	0	1	0	1
5	1	0	1	0	1
6	1	0	1	0	1
7	1	1	1	0	1
8	1	0	1	0	1
9	1	1	1	0	1
10	0	1	1	0	1
11	0	1	1	0	1

Personnel Scheduling (PSP)					
index	$\in X_{\text{prior}}$	$\in X_{\text{chesc}}$	$\in X_{\text{old}}$	$\in X_{\text{new}}$	$\in X_{\text{all}}$
0	1	0	1	0	1
1	1	0	1	0	1
2	1	0	1	0	1
3	1	0	1	0	1
4	1	0	1	0	1
5	1	1	1	0	1
6	1	0	1	0	1
7	1	0	1	0	1
8	1	1	1	0	1
9	1	1	1	0	1
10	0	1	1	0	1
11	0	1	1	0	1

Permutation Flow Shop (PFS)					
index	$\in X_{\text{prior}}$	$\in X_{\text{chesc}}$	$\in X_{\text{old}}$	$\in X_{\text{new}}$	$\in X_{\text{all}}$
0	1	0	1	0	1
1	1	1	1	0	1
2	1	0	1	0	1
3	1	1	1	0	1
4	1	0	1	0	1
5	1	0	1	0	1
6	1	0	1	0	1
7	1	0	1	0	1
8	1	1	1	0	1
9	1	0	1	0	1
10	0	1	1	0	1
11	0	1	1	0	1

Traveling Salesman Problem (TSP)					
index	$\in X_{\text{prior}}$	$\in X_{\text{chesc}}$	$\in X_{\text{old}}$	$\in X_{\text{new}}$	$\in X_{\text{all}}$
0	0	1	1	0	1
1	0	0	1	0	1
2	0	1	1	0	1
3	0	0	1	0	1
4	0	0	1	0	1
5	0	0	1	0	1
6	0	1	1	0	1
7	0	1	1	0	1
8	0	1	1	0	1
9	0	0	1	0	1

Vehicle Routing Problem (VRP)					
index	$\in X_{\text{prior}}$	$\in X_{\text{chesc}}$	$\in X_{\text{old}}$	$\in X_{\text{new}}$	$\in X_{\text{all}}$
0	0	0	1	0	1
1	0	1	1	0	1
2	0	1	1	0	1
3	0	0	1	0	1
4	0	0	1	0	1
5	0	1	1	0	1
6	0	1	1	0	1
7	0	0	1	0	1
8	0	0	1	0	1
9	0	1	1	0	1

0-1 Knapsack Problem (KP)					
index	$\in X_{\text{prior}}$	$\in X_{\text{chesc}}$	$\in X_{\text{old}}$	$\in X_{\text{new}}$	$\in X_{\text{all}}$
0	0	0	0	1	1
1	0	0	0	1	1
2	0	0	0	1	1
3	0	0	0	1	1
4	0	0	0	1	1
5	0	0	0	1	1
6	0	0	0	1	1
7	0	0	0	1	1
8	0	0	0	1	1
9	0	0	0	1	1

Quadratic Assignment Problem (QAP)					
index	$\in X_{\text{prior}}$	$\in X_{\text{chesc}}$	$\in X_{\text{old}}$	$\in X_{\text{new}}$	$\in X_{\text{all}}$
0	0	0	0	1	1
1	0	0	0	1	1
2	0	0	0	1	1
3	0	0	0	1	1
4	0	0	0	1	1
5	0	0	0	1	1
6	0	0	0	1	1
7	0	0	0	1	1
8	0	0	0	1	1
9	0	0	0	1	1

Max Cut Problem (MAC)					
index	$\in X_{\text{prior}}$	$\in X_{\text{chesc}}$	$\in X_{\text{old}}$	$\in X_{\text{new}}$	$\in X_{\text{all}}$
0	0	0	0	1	1
1	0	0	0	1	1
2	0	0	0	1	1
3	0	0	0	1	1
4	0	0	0	1	1
5	0	0	0	1	1
6	0	0	0	1	1
7	0	0	0	1	1
8	0	0	0	1	1
9	0	0	0	1	1

A.3 Properties of \hat{o} and Their Proofs

Here, we will list and proof some properties of the Importance Sampling (IS) estimate of algorithm performance described in Section 6.3.2.3. More specifically, for the variant defined in Equation 6.6:

$$\hat{o}(\theta) = \frac{1}{|E'|} \sum_{e \in E'} w_\theta(e) p(e)$$

with

$$w_\theta(e) = \frac{\text{pr}(e|\theta)}{Q'(e)} \quad \text{and} \quad Q'(e) = \sum_{\theta' \in \Theta'} \frac{|E'_{\theta'}|}{|E'|} \text{pr}(e|\theta')$$

where, $E' = \bigcup_{\theta' \in \Theta'} E'_{\theta'}$ is a sample of executions, generated using $\theta' \in \Theta'$ on $x \sim \mathcal{D}$.

Theorem A.1: \hat{o} combines all/only relevant observations

The weight function w_θ in Equation 6.6 satisfies

$$\text{pr}(e|\theta) > 0 \iff w_\theta(e) > 0, \forall e \in E', \forall \theta \in \Theta.$$

Proof. We show that $Q'(e) \in \mathbb{R}_0^+, \forall e \in E'$ which implies the statement in the theorem. $Q'(e)$ is non-negative and finite since it is a finite sum of non-negative, finite terms. $Q'(e)$ is also strictly positive, since at least one of these terms is strictly positive as $e \in E' \implies \exists \theta' \in \Theta' : \text{pr}(e|\theta') > 0 \wedge |E'_{\theta'}| > 0$. \square

Lemma A.2: \hat{o} is an unbiased estimator for o

Assuming E' generated as described above and $\text{pr}(e|\theta) > 0 \implies Q'(e) > 0$, $\hat{o}(\theta)$ as in Equation 6.6 is a random variable with

$$\mathbf{E}[\hat{o}(\theta)] = o(\theta)$$

i.e. \hat{o} is an unbiased estimator for o .

Proof.

$$\begin{aligned} \mathbf{E}[\hat{o}(\theta)] &= \mathbf{E} \left[\frac{1}{|E'|} \sum_{e \in E'} w_\theta(e) p(e) \right] \\ &= \mathbf{E} \left[\frac{1}{|E'|} \sum_{\theta' \in \Theta'} \sum_{e \in E'_{\theta'}} w_\theta(e) p(e) \right] \end{aligned}$$

$$\begin{aligned}
&= \frac{1}{|E'|} \sum_{\theta' \in \Theta'} \mathbf{E}_{E'_{\theta'} \sim \theta'} \left[\sum_{e \in E'_{\theta'}} w_{\theta}(e) p(e) \right] \\
&= \sum_{\theta' \in \Theta'} \frac{1}{|E'|} \mathbf{E}_{E'_{\theta'} \sim \theta'} \left[\sum_{e \in E'_{\theta'}} w_{\theta}(e) p(e) \right] \\
&= \sum_{\theta' \in \Theta'} \frac{|E'_{\theta'}|}{|E'|} \mathbf{E}_{e \sim \theta'} [w_{\theta}(e) p(e)] \\
&= \sum_{\theta' \in \Theta'} \frac{|E'_{\theta'}|}{|E'|} \sum_{e \in E} w_{\theta}(e) p(e) \text{pr}(e|\theta') \\
&= \sum_{e \in E} w_{\theta}(e) p(e) \sum_{\theta' \in \Theta'} \frac{|E'_{\theta'}|}{|E'|} \text{pr}(e|\theta') \\
&= \sum_{e \in E} \frac{\text{pr}(e|\theta)}{Q'(e)} p(e) Q'(e) \\
&= \sum_{e \in E} \text{pr}(e|\theta) p(e) \\
&= o(\theta)
\end{aligned}$$

□

Theorem A.3: \hat{o} is a consistent estimator for o

$\hat{o}(\theta)$ as shown in Equation 6.6 is a consistent estimator for $o(\theta)$, i.e.

$$\lim_{|E'| \rightarrow \infty} \Pr(|\hat{o}(\theta) - o(\theta)| < \epsilon) = 0, \forall \epsilon > 0. \quad (\text{convergence in probability})$$

Assuming there exist finite constants $b \geq 0$ and $q_{\min} > 0$ such that $\forall e \in E$

1. $|\text{pr}(e|\theta)p(e)| \leq b$.
2. $\text{pr}(e|\theta) > 0 \implies \exists \theta' \in \Theta' : \lim_{|E'| \rightarrow +\infty} \frac{|E'_{\theta'}|}{|E'|} \text{pr}(e|\theta') \geq q_{\min}$.

Proof. Using Markov's inequality we have

$$\lim_{|E'| \rightarrow +\infty} \Pr(|\hat{o}(\theta) - o(\theta)| < \epsilon) \leq \lim_{|E'| \rightarrow +\infty} \frac{\mathbf{E}[(\hat{o}(\theta) - o(\theta))^2]}{\epsilon^2}$$

As probabilities are by definition non-negative, it suffices to show that

$$\lim_{|E'| \rightarrow +\infty} \mathbf{E}[(\hat{o}(\theta) - o(\theta))^2] = 0 \quad (\text{convergence in MSE}).$$

Furthermore, since the MSE can be decomposed in a variance and (squared) bias term [Wackerly et al. 2007], we have

$$\begin{aligned}
\lim_{|E'| \rightarrow +\infty} \mathbf{E}[(\hat{o}(\theta) - o(\theta))^2] &= \lim_{|E'| \rightarrow +\infty} \mathbf{Var}[\hat{o}(\theta)] + (\mathbf{E}[(\hat{o}(\theta) - o(\theta))])^2 \\
&= \lim_{|E'| \rightarrow +\infty} \mathbf{Var}[\hat{o}(\theta)] \quad (\text{using Lemma A.2}) \\
&= \lim_{|E'| \rightarrow +\infty} \mathbf{Var} \left[\frac{1}{|E'|} \sum_{e \in E'} w_\theta(e) p(e) \right] \\
&= \lim_{|E'| \rightarrow +\infty} \mathbf{Var} \left[\frac{1}{|E'|} \sum_{\theta' \in \Theta'} \sum_{e \in E'_{\theta'}} w_\theta(e) p(e) \right] \\
&= \lim_{|E'| \rightarrow +\infty} \frac{1}{|E'|^2} \sum_{\theta' \in \Theta'} \mathbf{Var}_{E'_{\theta'} \sim \theta'} \left[\sum_{e \in E'_{\theta'}} w_\theta(e) p(e) \right] \\
&= \lim_{|E'| \rightarrow +\infty} \frac{1}{|E'|^2} \sum_{\theta' \in \Theta'} |E'_{\theta'}| \mathbf{Var}_{e \sim \theta'} [w_\theta(e) p(e)] \\
&= \lim_{|E'| \rightarrow +\infty} \frac{1}{|E'|^2} \sum_{\theta' \in \Theta'} |E'_{\theta'}| \mathbf{E}_{e \sim \theta'} [w_\theta(e)^2 p(e)^2] \\
&\quad - \lim_{|E'| \rightarrow +\infty} \frac{1}{|E'|^2} \sum_{\theta' \in \Theta'} |E'_{\theta'}| \mathbf{E}_{e \sim \theta'} [w_\theta(e) p(e)]^2 \\
&\leq \lim_{|E'| \rightarrow +\infty} \frac{1}{|E'|^2} \sum_{\theta' \in \Theta'} |E'_{\theta'}| \mathbf{E}_{e \sim \theta'} [w_\theta(e)^2 p(e)^2] \\
&\leq \lim_{|E'| \rightarrow +\infty} \frac{1}{|E'|^2} \sum_{\theta' \in \Theta'} |E'_{\theta'}| \sum_{e \in E} \text{pr}(e|\theta') w_\theta(e)^2 p(e)^2 \\
&\leq \lim_{|E'| \rightarrow +\infty} \frac{1}{|E'|^2} \sum_{\theta' \in \Theta'} |E'_{\theta'}| \sum_{e \in E} \text{pr}(e|\theta') \frac{\text{pr}(e|\theta')^2}{Q'(e)^2} p(e)^2 \\
&\leq \lim_{|E'| \rightarrow +\infty} \frac{b^2}{q_{\min}^2 |E'|^2} \sum_{\theta' \in \Theta'} |E'_{\theta'}| \sum_{e \in E} \text{pr}(e|\theta') \\
&\leq \lim_{|E'| \rightarrow +\infty} \frac{b^2}{q_{\min}^2 |E'|^2} \sum_{\theta' \in \Theta'} |E'_{\theta'}| \\
&\leq \lim_{|E'| \rightarrow +\infty} \frac{b^2}{q_{\min}^2 |E'|} \\
&\leq 0
\end{aligned}$$

□

Theorem A.4: \hat{o} degenerates to \bar{o} for non-overlapping samples

When all observations, thus far, are relevant only for the configuration that was used to generate them, the IS estimate reduces to an ordinary sample average, i.e.

$$\begin{aligned} \text{pr}(e|\theta) * \text{pr}(e|\theta') &= 0, \forall e \in E', \forall \theta \in \Theta, \forall \theta' \in \Theta' \setminus \{\theta\} \quad (\text{A}) \\ \implies \\ \hat{o}(\theta) &= \bar{o}(\theta), \forall \theta \in \Theta \end{aligned}$$

Proof. We first show that the left hand side of the implication (A) implies

$$w_\theta(e) = \begin{cases} \frac{|E'|}{|E'_\theta|} & e \in E'_\theta \\ 0 & e \in E' \setminus E'_\theta \end{cases}$$

Assuming $e \in E'_\theta$: This implies $\text{pr}(e|\theta) > 0$.

From (A) it follows that $\text{pr}(e|\theta') = 0, \forall \theta' \in \Theta' \setminus \{\theta\}$ and therefore

$$w_\theta(e) = \frac{\text{pr}(e|\theta)}{\sum_{\theta' \in \Theta'} \frac{|E'_\theta|}{|E'|} \text{pr}(e|\theta')} = \frac{\text{pr}(e|\theta)}{\frac{|E'_\theta|}{|E'|} \text{pr}(e|\theta)} = \frac{|E'|}{|E'_\theta|}$$

Assuming $e \in E' \setminus E'_\theta$: This implies $\exists \theta' \in \Theta' \setminus \{\theta\} : \text{pr}(e|\theta') > 0$.

From (A) it follows that $\text{pr}(e|\theta) = 0$ and therefore $w_\theta(e) = 0$.

We can now rewrite Equation 6.6 as follows:

$$\hat{o}(\theta) = \frac{1}{|E'|} \sum_{e \in E'} w_\theta(e) p(e) = \frac{1}{|E'|} \sum_{e \in E'_\theta} \frac{|E'|}{|E'_\theta|} p(e) = \frac{1}{|E'_\theta|} \sum_{e \in E'_\theta} p(e) = \bar{o}(\theta) \quad \square$$

Theorem A.5: pr' and pr are interchangeable in the computation of \hat{o}

Assuming pr' is any function satisfying criteria (a) and (b) in Equation 6.1, we have

$$w_\theta(e) = \frac{\text{pr}'(e|\theta)}{\sum_{\theta' \in \Theta'} \frac{|E'_\theta|}{|E'|} \text{pr}'(e|\theta')}, \forall e \in E', \forall \theta \in \Theta$$

Proof. We will prove the cases $w_\theta(e) = 0$ and $w_\theta(e) > 0$ separately.

$w_\theta(e) = 0$: Note that this implies $\text{pr}(e|\theta) = 0$ (Theorem A.1).

By criterion 6.1.a we therefore have $\text{pr}'(e|\theta) = 0$.

Also, as $e \in E'$ we have $\sum_{\theta' \in \Theta'} \frac{|E'_{\theta'}|}{|E'|} \text{pr}(e|\theta') > 0, \forall e \in E'$.

Combining this with the fact that terms in $Q'(e)$ are non-negative, criterion 6.1.a

implies that also $\sum_{\theta' \in \Theta'} \frac{|E'_{\theta'}|}{|E'|} \text{pr}'(e|\theta') > 0$ and $\frac{\text{pr}'(e|\theta)}{\sum_{\theta' \in \Theta'} \frac{|E'_{\theta'}|}{|E'|} \text{pr}'(e|\theta')} = 0$ follows.

$w_\theta(e) > 0$: Note that this implies $\text{pr}(e|\theta) > 0$ (Theorem A.1).

$$\begin{aligned}
 w_\theta(e) &= \frac{\text{pr}(e|\theta)}{\sum_{\theta' \in \Theta'} \frac{|E'_{\theta'}|}{|E'|} \text{pr}(e|\theta')} \\
 &= \frac{1}{\sum_{\theta' \in \Theta'} \frac{|E'_{\theta'}|}{|E'|} \frac{\text{pr}(e|\theta')}{\text{pr}(e|\theta)}} && \text{since } \text{pr}(e|\theta) > 0 \\
 &= \frac{1}{\sum_{\theta' \in \Theta'} \frac{|E'_{\theta'}|}{|E'|} \frac{\text{pr}'(e|\theta')}{\text{pr}'(e|\theta)}} && \text{criterion 6.1.b} \\
 &= \frac{\text{pr}'(e|\theta)}{\sum_{\theta' \in \Theta'} \frac{|E'_{\theta'}|}{|E'|} \text{pr}'(e|\theta')} && \text{since } \text{pr}'(e|\theta) > 0 \quad \square
 \end{aligned}$$

A.4 The Dynamic Metaheuristic Scheduling wb-ACP

In Section 6.5.3, we considered the problem of designing a dynamic scheduler for two anytime (meta-heuristic) optimization methods, which we solved by reducing it to a “white box configuration problem” (wb-ACP, see Definition 6.1). Here, we describe the specifics of the target algorithm a , and the procedure for computing pr' , used in this reduction.

Target Algorithm (a)

The pseudo-code for a is shown in Algorithm 24. As input it takes x , the minimization problem instance to be solved, hh_1 and hh_2 , two preemptable anytime optimizers that can be used for doing so, and a configuration θ consisting of 22 real-valued parameter values determining how resources are allocated to hh_1 and hh_2 . Remark that this allocation depends on “the anytime performance” of hh_1 and hh_2 during “previously allocated slots”. The “the anytime performance” of a method hh_j during a run is fully determined by its “solution quality trace” T_j , which is the set of pairs (t_i, e_i) , indicating that after running hh_j for t_i seconds, it returns a solution with evaluation function value e_i .

Algorithm 24 Target algorithm used in the wb-ACP reduction of the dynamic metaheuristic scheduling problem.

```

1: procedure SCHEDULE( $\theta, \langle x, hh_1, hh_2 \rangle$ )
2:    $T_1, T_2 \leftarrow \{(0, +\infty)\}$ 
3:    $n_1, n_2 \leftarrow 0$ 
4:    $id_1 \leftarrow (hh_1.q_0, x)$ 
5:    $id_2 \leftarrow (hh_2.q_0, x)$ 
6:   for  $i = 1 : 100$  do
7:      $pr_1 \leftarrow \text{PRNEXT1}(\theta, T_1, T_2, n_1, n_2)$ 
8:      $\text{pivot} \sim \mathcal{U}(0, 1)$ 
9:     if  $\text{pivot} \leq pr_1$  then
10:       $j \leftarrow 1$ 
11:    else
12:       $j \leftarrow 2$ 
13:    end if
14:     $\langle id_j, T' \rangle \leftarrow \text{continue}(hh_j, id_j, 6)$ 
15:     $T_j \leftarrow T_j \cup T'$ 
16:     $n_j \leftarrow n_j + 1$ 
17:  end for
18:   $p_e \leftarrow \text{EVALUATE}(hh_1, hh_2, id_1, id_2, T_1, T_2, n_1, n_2)$ 
19:  return  $\langle T_1, T_2, n_1, p_e \rangle$ 
20: end procedure

21: procedure PRNEXT1( $\theta, T_1, T_2, n_1, n_2$ )
22:   $e_1^{\min} \leftarrow \min\{e|(t, e) \in T_1\}$ 
23:   $e_2^{\min} \leftarrow \min\{e|(t, e) \in T_2\}$ 
24:   $\omega_{\text{best1}} \leftarrow 0.5[e_1^{\min} \leq e_2^{\min}] + 0.5[e_1^{\min} < e_2^{\min}]$ 
25:   $f_1^{\min} \leftarrow \min\{e|(t, e) \in T_1 \wedge t \leq \min(6n_1, 6n_2)\}$ 
26:   $f_2^{\min} \leftarrow \min\{e|(t, e) \in T_2 \wedge t \leq \min(6n_1, 6n_2)\}$ 
27:   $\omega_{\text{fair1}} \leftarrow 0.5[f_1^{\min} \leq f_2^{\min}] + 0.5[f_1^{\min} < f_2^{\min}]$ 
28:   $\omega_{\text{diff1}} \leftarrow \frac{n_1 - n_2}{100}$ 
29:   $\omega_{\text{elapsed}} \leftarrow \frac{n_1 + n_2}{100}$ 
30:  if  $e_1^{\min} < e_2^{\min}$  then
31:     $\omega_{\text{ahead}} \leftarrow \frac{6n_2 - \min\{t|(t, e) \in T_1 \wedge e < e_2^{\min}\}}{600}$ 
32:  else if  $e_1^{\min} > e_2^{\min}$  then
33:     $\omega_{\text{ahead}} \leftarrow \frac{6n_1 - \min\{t|(t, e) \in T_2 \wedge e < e_1^{\min}\}}{600}$ 
34:  else
35:     $\omega_{\text{ahead}} \leftarrow 0$ 
36:  end if
37:   $v_1 \leftarrow \text{MLP.predict}(\theta, \langle \omega_{\text{best1}}, \omega_{\text{fair1}}, \omega_{\text{diff1}}, \omega_{\text{elapsed}}, \omega_{\text{ahead}} \rangle)$ 
38:   $v_2 \leftarrow \text{MLP.predict}(\theta, \langle 1 - \omega_{\text{best1}}, 1 - \omega_{\text{fair1}}, -\omega_{\text{diff1}}, \omega_{\text{elapsed}}, \omega_{\text{ahead}} \rangle)$ 
39:  return  $\frac{v_1}{v_1 + v_2}$ 
40: end procedure

```

```

41: procedure EVALUATE( $hh_1, hh_2, id_1, id_2, T_1, T_2, n_1, n_2$ )
42:    $\langle -, T' \rangle \leftarrow \text{continue}(hh_1, id_1, 600 - 6n_1)$ 
43:    $T_1 \leftarrow T_1 \cup T'$ 
44:    $\langle -, T' \rangle \leftarrow \text{continue}(hh_2, id_2, 600 - 6n_2)$ 
45:    $T_2 \leftarrow T_2 \cup T'$ 
46:    $C = \{c_i | c_i = \min(\min\{e|(t, e) \in T_1 \wedge t \leq 6i\}, \min\{e|(t, e) \in T_2 \wedge t \leq 600 - 6i\}) \wedge 0 \leq i \leq 100\}$ 
47:   return  $-\frac{|\{c_i | c_i < c_{n_1}\}| + 0.5(|\{c_i | c_i = c_{n_1} \wedge i \neq n_1\}|)}{|C|}$ 
48: end procedure

```

First, a performs some initialization for each method hh_j (lines 2-5). It initializes its trace (T_j) to $\{(0, +\infty)\}$, its “previously allocated slots” (n_j) to 0, and its execution state (id_j). An execution state fully describes the current state of a running program, it consists of its current control (initially $hh_j.q_0$) and memory state (initially x). Subsequently, a allocates 100 (6s) slots iteratively to hh_1 and hh_2 . Each iteration, at line 7, a call to the stochastic scheduling policy π_θ returns pr_1 the likelihood of assigning the next slot to hh_1 instead of hh_2 (lines 21-40). First, at lines 22-36, π_θ derives the following five runtime features:

ω_{best1} is equal to 1 if and only if the best solution (thus far) obtained by hh_1 is better than that obtained by hh_2 . Equal to 0, if and only if hh_2 obtained the better solution and equal to 0.5 otherwise (initially, or in case of ties).

ω_{fair1} : Clearly, only looking at “which method obtained the best solution thus far” is unfair as we may have allocated resources non-uniformly. ω_{fair1} is similar to ω_{best1} , but compares the solution quality obtained at the time of the shortest run.

ω_{diff1} : measures how much longer we have ran hh_1 than hh_2 .

ω_{elapsed} : measures the fraction of the total budget we have allocated thus far.

ω_{ahead} : measures how much less time “the method which has obtained the best solution thus far” needed to obtain a solution quality better than the best obtained thus far by the other method. Remark that ω_{ahead} may be negative if $\omega_{\text{best1}} \neq \omega_{\text{fair1}}$.

To determine the likelihood of allocating the next slot to hh_1 given ω (lines 37-40), π_θ uses the Multi-layer Perceptron (MLP) depicted in Figure A.1. This MLP has five inputs (+ bias), one hidden layer of three nodes (+ bias) and a single output node. Layers are fully connected, giving rise to a total of 22 edges whose weights are given by θ . All nodes use a sigmoid activation function, i.e. the output of a node in function of its inputs x and weights w is given by $\left(1 - e^{-(w_0 + \sum_i w_i x_i)}\right)^{-1}$. Concretely, pr_1 is given by $\frac{v_1}{v_1 + v_2}$, where v_1 and v_2 are the predictions of this MLP for inputs $\langle \omega_{\text{best1}}, \omega_{\text{fair1}}, \omega_{\text{diff1}}, \omega_{\text{elapsed}}, \omega_{\text{ahead}} \rangle$ and $\langle 1 - \omega_{\text{best1}}, 1 - \omega_{\text{fair1}}, -\omega_{\text{diff1}}, \omega_{\text{elapsed}}, \omega_{\text{ahead}} \rangle$, respectively. Remark that all candidate schedulers allocate the first slot to hh_1/hh_2 with equal likelihood (since $v_1 = v_2$).

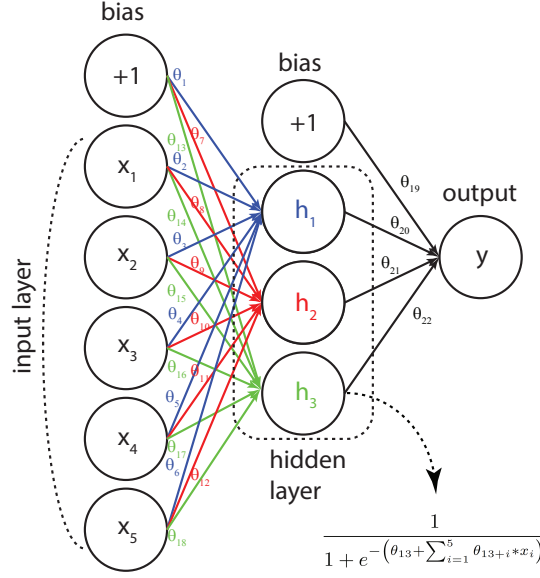


Figure A.1: Architecture of the MLP used in Algorithm 24.

Next, at lines 8-13, a selects hh_1 with likelihood pr_1 and continues the selected method (hh_j) for six seconds from its current execution state (id_j) at line 14. At line 15, it updates its trace (T_j), adding any new best solutions found (T') and increments its “previously allocated slots” counter (n_j) at line 16.

Finally, at line 19, a returns $e = (T_1, T_2, n_1, p_e)$, where p_e is the desirability of the allocating $(n_1, 100 - n_1)$ slots to (hh_1, hh_2) , which is computed at line 18. Remark that this code would obviously not be part of the interpreted scheduler, which at this point would simply return the best solution obtained by either hh_1 or hh_2 . a computes p_e , at lines 41-48, by comparing the quality of the solution obtained by this allocation to that of those that would have been obtained by the 100 other possible allocations. To this end, we first continue to run each method for the full budget (10 min) and update their traces accordingly. Next, from the now complete traces T_1 and T_2 , we derive the solution quality obtained by any of the 101 possible schedules at line 46. At line 47, we quantify the desirability of our schedule as “minus the fraction of schedules which obtained a better solution”. To account for ties, we also subtract half of the fraction of tying schedules. Remark that the resulting evaluation value falls in the range $[-1, 0]$ and is 0/-1 if and only if the allocation made by our scheduler is the unique best/worst.

Computing pr'

$\text{pr}'(\langle T_1, T_2, n_1, p_e \rangle | \theta)$ is the likelihood that the scheduling policy π_θ allocates $(n_1, 100 - n_1)$ slots to (hh_1, hh_2) , given their traces (T_1, T_2) . This can be computed efficiently/elegantly using dynamic programming (see Algorithm 25). Here, each iteration k , we compute the likelihood of all allocations of $k + 1$ slots (allocating at most n_1 slots to hh_1 and $100 - n_1$ slots to hh_2), based on the likelihood of the allocations of k slots.

Algorithm 25 pr' used in the dynamic metaheuristic scheduling wb-ACP reduction.

```

1: procedure  $\text{pr}'(\theta, \langle T_1, T_2, n_1, p_e \rangle)$ 
2:    $\text{pr}_{0,0} = 1$ 
3:    $\text{pr}_{i,j} = 0, \forall 0 < i \leq n_1, \forall 0 < j \leq 100 - n_1$ 
4:   for  $k = 0 : 99$  do
5:     for  $l = 0 : k$  do
6:       if  $\text{pr}_{l,k-l} > 0$  then
7:          $\text{pr}_1 \leftarrow \text{PrNEXT1}(\theta, \{(t, e) \in T_1 | t < 6l\}, \{(t, e) \in T_2 | t < 6(k-l)\}, l, k-l)$ 
8:         if  $l < n_1$  then
9:            $\text{pr}_{l+1,k-l} \leftarrow \text{pr}_{l+1,k-l} + \text{pr}_1 * \text{pr}_{l,k-l}$ 
10:        end if
11:        if  $k-l < 100 - n_1$  then
12:           $\text{pr}_{l,k-l+1} \leftarrow \text{pr}_{l,k-l+1} + (1 - \text{pr}_1) * \text{pr}_{l,k-l}$ 
13:        end if
14:      end if
15:    end for
16:  end for
17:  return  $\text{pr}_{n_1, 100-n_1}$ 
18: end procedure

```

A.5 AdapHH's Anatomy

AdapHH is the selection hyper-heuristic that won the CHeSC 2011 competition. After the competition its code was made publicly available (under the name GIHH). We used this implementation as a baseline in our experiments in Section 5.3.2.4. Also, we considered it as the subject of our case study in Section 7.3, targeted at identifying needless complexity. To fully understand the simplifications that were considered in context of the latter, one must have a thorough understanding of the method and its many sub-mechanisms. Thus far, in Section 5.3.2.4, p. 178, we have presented a description of its high-level search strategy (see Algorithm 26). Here, we complement this description, providing more detailed descriptions of all sub-mechanisms in AdapHH. These descriptions are our own, based on a thorough analysis of the source code, and are, for ease of reference, ordered according to their discussion in Section 7.3.2.1.

Algorithm 26 High-level search strategy for AdapHH

```

1: function SOLVE(problem,  $t_{\text{allowed}}$ )
2:    $v_{\text{inc}} \leftarrow \text{init}()$ 
3:    $c_{\text{phase}} \leftarrow 0$ 
4:   while  $t_{\text{elapsed}} < t_{\text{allowed}}$  do
5:      $c_{\text{phase}} \leftarrow c_{\text{phase}} + 1$ 
6:     if  $\neg \text{restart disabled} \wedge \text{restart condition met}$  then
7:        $\text{reinit}()$ 
8:     else if  $\text{restart disabled}$  then
9:        $\text{best-init}()$ 
10:    end if
11:    update  $p_{\text{relay}}$ 
12:    if  $p_{\text{relay}} < \text{pivot} \sim \mathcal{U}(0, 1)$  then
13:       $h_{\text{single}} \leftarrow \text{select single heuristic}$ 
14:       $v_{\text{prop}} \leftarrow h_{\text{single}}(v_{\text{inc}})$ 
15:    else
16:       $h_{\text{rh1}} \leftarrow \text{select first heuristic in relay hybridization}$ 
17:       $v_{\text{prop}} \leftarrow h_{\text{rh1}}.\text{apply}(v_{\text{inc}})$ 
18:      if  $v_{\text{prop}}$  not best since last reinit then
19:         $h_{\text{rh2}} \leftarrow \text{select second heuristic in relay hybridization}$ 
20:         $v_{\text{prop}} \leftarrow h_{\text{rh2}}.\text{apply}(v_{\text{prop}})$ 
21:        update rh-selection for  $h_{\text{rh1}}, h_{\text{rh2}}$ 
22:      end if
23:    end if
24:    update parameters/statistics for  $h_{\text{single}}, h_{\text{rh1}}, h_{\text{rh2}}$ 
25:    if  $\text{accept}(v_{\text{prop}})$  then
26:       $v_{\text{inc}} \leftarrow v_{\text{prop}}$ 
27:    end if
28:    if  $c_{\text{phase}} = \text{pl}$  then
29:      update elitist set, heuristic types and pl.
30:       $c_{\text{phase}} \leftarrow 0$ 
31:    end if
32:  end while
33: end function

```

Phase-based Heuristic Set Selection (ADHS)

Crossover Heuristics: AdapHH, unlike many other single point hyper-heuristics, uses crossover heuristics (xo). These differ from ordinary (mut, rr, ls) low-level heuristics in that they take two, rather one candidate solution as input. The second solution used in the application of crossover heuristics, in AdapHH, is selected uniformly at random from a list of five candidate solutions. Initially, this list contains copies of the initial solution. Whenever a new best solution is obtained, a random of these is replaced (line 24).

Adaptive Phase Length: AdapHH subdivides a run into multiple phases, each lasting a varying # iterations (pl). Initially, $pl = \lfloor \sqrt{2N} \rfloor * PH_{factor}$, where N is the # heuristics in the domain and PH_{factor} a parameter (default 500). After each phase (line 29) $pl = \lfloor \frac{t_{allowed}}{PH_{requested} * t_{subset}} \rfloor$, where $t_{allowed}$ is the time we are allowed to optimize, t_{subset} the average duration per application of a heuristic within the elitist set and $PH_{requested}$ a parameter (default 100). Furthermore, the range of pl is bounded:

$$pl = \max(N * PH_{factor}, \min(pl, \lfloor \sqrt{2N} \rfloor * PH_{factor}^{\min}))$$

where PH_{factor}^{\min} is a parameter (default 50).

Heuristic Types: In HyFlex, low-level heuristics are statically assigned to one of four categories (xo, mut, rr, ls).² AdapHH further classifies each low-level heuristic i dynamically as being of one of the following types:

$$t(i) = \begin{cases} \text{ImprovingOrEqual} & f_{imp}(i) > 0 \wedge f_{wrs}(i) = 0 \\ \text{ImprovingMore} & f_{imp}(i) \geq f_{wrs}(i) > 0 \\ \text{EqualOnly} & f_{wrs}(i) = f_{imp}(i) = 0 \\ \text{WorseningMore} & f_{wrs}(i) > f_{imp}(i) > 0 \\ \text{WorseningOrEqual} & f_{imp}(i) = 0 \wedge f_{wrs}(i) > 0 \end{cases}$$

where $f_{imp}(i)$ and $f_{wrs}(i)$ are the total amount of improvement and worsening induced by applications of i over the entire run. Initially, each heuristic is classified as ImprovingMore and types are updated after each phase (line 29).

²Technically five, including the single initialization heuristic. However, as also noted in [Adriaenssen 2013, Van Onsem et al. 2014], this classification is not particularly informative.

Phase-based Tabu:

QI Tabu Criterion: After each phase (line 29) the performance of a low-level heuristic i is measured as

$$p_i = 10^8 (c_{p,best}(i) + 1)^2 \left(\frac{t_{remain}}{t_{p,spent}(i)} \right) + 10^5 \left(\frac{f_{p,imp}(i)}{t_{p,spent}(i)} \right) - 10^{-4} \left(\frac{f_{p,wrs}(i)}{t_{p,spent}(i)} \right) + 10^{-6} \left(\frac{f_{imp}(i)}{t_{spent}(i)} \right) - 10^{-9} \left(\frac{f_{wrs}(i)}{t_{spent}(i)} \right)$$

where $c_{p,best}(i)$ is the number of new best solutions found by i during the last phase, t_{remain} is the time left to optimize and $t_{spent}(i)$ is the total time spent by applications of heuristic i over the entire run. Notations with subscript p (e.g. $f_{p,imp}$) represent the value of a metric (e.g. f_{imp}) during the last phase only. Quality indices qi_i , ranging from 1 to N are determined from p_i by means of rank. The highest p_i gets N , and so on. Each heuristic j which was not used last phase has $qi_j = 1$. A heuristic i in the elitist subset is excluded, if $qi_i < \frac{\sum_j qi_j}{N}$.

Extreme Exclusion: Heuristic i is also excluded (line 29) if *all* the following hold:

- i found no new best solutions during last phase.
- i was not tabu during one of the last two phases.
- $exc(i) > 2 * avg(exc)$
- $std(exc) = \sqrt{\sum_i (exc(i) - avg(exc))^2} > 2$
- After exclusion at least one heuristic in the elitist set found a new best solution over the entire run.

where $exc(j)$ is the number of times heuristic j is slower than the fastest heuristic in the full heuristic set.

Tabu Duration: Each heuristic has a tabu duration d_i determining the # phases it remains tabu after being excluded and which is initially set to $D = \lfloor \sqrt{2N} \rfloor$. If a heuristic i which was re-included last phase, is again excluded its tabu duration d_i is increased by one (i.e. it will now remain tabu one phase longer). Otherwise, d_i is reset to D . This incrementation stops when it reaches an upper-bound of $2D$, if this occurs i is excluded permanently if $exc(i) > 5$.

Heuristic Selection

AdapHH combines two mechanisms to select heuristics during a phase: Single heuristic selection and Relay Hybridization (RH). Each iteration (line 11), RH is applied with probability $p_{\text{relay}} = (\frac{c_{\text{phase}}}{\text{pl}})^{\gamma}$, where c_{phase} is the number of iterations performed in current phase and γ the ratio of new best solutions found by a single heuristic (+1) over those found by the combination of two heuristics (+1). Furthermore, RH is made tabu (line 29) for d_r phases if it did not find any new best solutions during last phase. Initially, $d_r = D_r^{\text{lb}}$ (default 1), and d_r is incremented for each consecutive exclusion with upper bound D_r^{ub} (default 10). Otherwise d_r is reset.

Single Heuristic Selection (SS): Heuristic i is selected proportionally to

$$\left(\frac{c_{\text{best}}(i) + 1}{\max(t_{\text{spent}}(i), 1\text{ms})} \right)^{1+3\text{tf}}$$

from the elitist set (line 13). With tf the fraction of time remaining.

Relay Hybridization (RH): Using Relay Hybridization, two (possibly tabu) heuristics are applied consecutively (within one iteration), where the selection probability of the second heuristic depends on the first heuristic selected (lines 16-22).

Selection first heuristic in RH (line 16): A low-level heuristic i from the full heuristic set is selected proportionally to pr_i . Initially, we have $\text{pr}_i = \frac{1}{N}$ and pr_i 's are updated dynamically. Specifically, updates $\text{pr}_k = \text{pr}_k + \alpha_r(j) * (x_k - \text{pr}_k)$ are performed (line 21) for each heuristic k when a new best solution is found, where $x_k = 1$ for the selected heuristic i and $x_k = 0$ otherwise. Here, j is the second heuristic in the RH. The learning rate α_r is determined for each individual heuristic as $\alpha_r(i) = \frac{A_r}{\text{exc}(i)}$, where A_r is a parameter (default 0.5) and $\alpha_r(i) \geq \frac{A_r}{25}$ is guaranteed. If the first heuristic in RH finds a new best solution, we do not apply the second heuristic (cancel RH).

Selection second heuristic in RH (line 19): For each heuristic i , a list of length L_{12} (default 10) is kept, containing the indices of low-level heuristics that found a new best solution when being applied after i . When a new best solution is found and the list is full, the oldest entry will be replaced. Given it is non-empty, the second heuristic is selected uniformly from this list (which can contain duplicates) with a probability of P_{12} (default 0.25). Otherwise, the second heuristic is selected based on dynamic type. Specifically, with a probability P_{ioe} (default 0.5), a heuristic with heuristic class type

ImprovingOrEqual selected uniformly at random. Otherwise (or if none exist), a heuristic of type ImprovingMore is selected uniformly at random. Finally, if no such heuristic exists, the ordinary single selection scheme is used to select the second heuristic.

Dynamic Parameter Adaptation

The value of $\alpha = \beta = v_i$ used when applying a low-level heuristic i is initially set to V_{init} (default 0.2) and is updated using $v_i = v_i + \theta_x * u$ within bounds $[V_{\text{lb}}, V_{\text{ub}}]$ (default $[0.2, 1.0]$) after each application (line 24). Here u is a random variable, whose distribution depends on the heuristic type and x (see Table A.1) indicating whether v_{prop} is a new best ($x = 0$, $\theta_0 = 10^{-2}$), better ($x = 1$, $\theta_1 = 10^{-3}$), worse ($x = 2$, $\theta_2 = 5 * 10^{-4}$) or as good ($x = 3$, $\theta_3 = 10^{-4}$) as v_{inc} .

Parameter Oscillation: A mechanism oscillating v_i is used (line 22) when the AdapHH is stuck (acceptance threshold reached maximum) and reinitialization is disabled. Once stuck, v_i is periodically linearly varied between V_{lo} and V_{hi} (following a triangle wave). For ImprovingOrEqual heuristics $V_{\text{lo}} = 0.5$ and $V_{\text{hi}} = 1.0$, for others $V_{\text{lo}} = 0.2$ and $V_{\text{hi}} = 0.5$. The # applications from V_{lo} to V_{hi} is V_p (default 5000).

Table A.1: Distribution of u for heuristic types t and feedback x

Heuristic Type	x	$\Pr(u = -1)$	$\Pr(u = 0)$	$\Pr(u = 1)$	\tilde{u}
ImprovingOrEqual	0/1	0	0.5	0.5	0.5
	2	n.a	n.a	n.a	n.a
	3	0.5	0.25	0.25	-0.25
ImprovingMore	0/1	0.25	0.25	0.5	0.25
	2	0.5	0.5	0	-0.5
	3	0.5	0.5	0	-0.5
EqualOnly	0/1	n.a	n.a	n.a	n.a
	2	n.a	n.a	n.a	n.a
	3	0	0	1	1
WorseningMore	0/1	0.5	0	0.5	0
	2	1	0	0	-1
	3	0	0	1	1
WorseningOrEqual	0/1	n.a	n.a	n.a	n.a
	2	1	0	0	-1
	3	0	0	1	1

Acceptance Heuristic and Reinitialization

Acceptance Criterion (line 25): Always accept improvement. Do not accept any worsening for an initial $\#$ worsening iterations k after improving the incumbent solution or accepting a worsening solution. Afterwards, accept solutions no worse than a threshold value taken from the `best_list` at `index`. Here, `best_list` is the list of fitness values of the l most recent new best solutions found since last restart. Initially, it holds l times the fitness of the initial solution.

Index Increasing: Initially, the `index` points to the second smallest element in the list and it is increased every time we have $K = K_{\text{mult}} * k > \text{it}_{\text{wrs}}$, where K_{mult} is a parameter (default 5) and it_{wrs} is the number of worsening iterations since last new best or increment. This `index` is reset when a new best solution is found.

List Length Adaptation: Initially l is set to L_{initial} (default 10). Each iteration l is updated as follows $l = \frac{L_{\text{initial}}}{2} + \text{tf}^3(\frac{L_{\text{initial}}}{2} + 1)$, i.e. it decreases in length up and till half its initial size.

Patience Adaptation: Initially $k = k_{\text{lb}}$ (default 5). k is updated whenever a new best solution is found as $k = \lfloor \frac{L_{\text{initial}} - 1}{L_{\text{initial}}} k + \frac{1}{L_{\text{initial}}} y \rfloor$. Let $\text{cw} = \lfloor \frac{\text{it}_{\text{wrs}}}{k} \rfloor$. If $\text{cw} = 0$, $y = \text{it}_{\text{wrs}}$, i.e. we lower k . Otherwise, $\text{cw} > 0$ and $y = k + k * \text{tf} * (1 - 0.5^{\text{cw} - 1})$, i.e. we increase k . K is updated as well (i.e. $K_{\text{mult}} * k$ invariant).

Reinitialization (lines 6-10): Whenever the threshold `index` exceeds l the acceptance mechanism signals that the search is stuck. This potentially activates the reinitialization mechanism. We found deducing the exact restart condition from the code difficult, in this section we present a set of simplified rules (`srestart`, see also Section 7.3.2.1):

$0.8 \leq \text{tf}$: Reinitialize on stuck.

$0.5 \leq \text{tf} < 0.8$: Given we are stuck and restart was not yet disabled, reinitialize if we have performed a reinitialization before and the runs following the last five reinitializations have not failed to find a new best solution. Otherwise, disable restart.

$\text{tf} < 0.5$: Disable restart.

Reinitialization (line 7) entails reinitializing the v_{inc} , the five solutions used for crossover and the `best_list`, `index` and other acceptance related counters. When restart is disabled and the current run's best is not the overall best, we perform `best-init` (line 9), where v_{inc} is replaced with the overall best solution, `best_list` by that of the best run and `index` and other acceptance related counters are reset.

List of Publications

Full Papers in Proceedings of International Conferences

- 2017 Steven Adriaensen**, Filip Moons, Ann Nowé. "An Importance Sampling Approach to the Estimation of Algorithm Performance in Automated Algorithm Design", Learning and Intelligent Optimization (LION) 11, 2017.
- 2016 Steven Adriaensen**, Ann Nowé. "Case study: An analysis of accidental complexity in a state-of-the-art hyper-heuristic for HyFlex", 2016 IEEE Congress on Evolutionary Computation (CEC). IEEE, pp. 1485-1492, 2016.
- 2016 Steven Adriaensen**, Ann Nowé. "Towards a white box approach to automated algorithm design", International Joint Conferences on Artificial Intelligence (IJCAI), pp. 554-560, 2016.
- 2015 Steven Adriaensen**, Gabriela Ochoa, Ann Nowé, "A Benchmark Set Extension and Comparative Study for the HyFlex Framework", 2015 IEEE Congress on Evolutionary Computation (CEC), IEEE, pp. 784-791, 2015.
- 2014 Steven Adriaensen**, Tim Brys, Ann Nowé, "Fair-share ILS: a simple state-of-the-art iterated local search hyperheuristic", Proceedings of the 2014 conference on Genetic and evolutionary computation (GECCO), pp. 1303-1310, 2014.
- 2014 Steven Adriaensen**, Tim Brys, Ann Nowé, "Designing reusable metaheuristic methods: A semi-automated approach", 2014 IEEE Congress on Evolutionary Computation (CEC), pp. 2969-2976, 2014.

Extended abstracts

- 2015** Jerry Swan, **Steven Adriaensen**, Mohamed Bishr, Edmund K Burke, John A Clark, Patrick De Causmaecker, Juanjo Durillo, Kevin Hammond, Emma Hart, Colin G Johnson, 18 others, “A Research Agenda for Metaheuristic Standardization”, Proceedings of the XI Metaheuristics International Conference (MIC), 2015.
- 2015** **Steven Adriaensen**, Yasmin fathy, Ann Nowé, “On task scheduling policies for work-stealing schedulers”, Proceedings of the 7th Multidisciplinary International Conference on Scheduling: Theory and Applications (MISTA), 2015.
- 2014** **Steven Adriaensen**, Tim Brys, Ann Nowé, “Fair-share ILS: a simple state-of-the-art iterated local search hyperheuristic”, Benelux Conference on Artificial Intelligence (BNAIC), 2014.

List of Acronyms

~	similar to (if used in text) or distributed according to (if used in math)
a.k.a.	also known as
ACA	Accidental Complexity Analysis (see Section 7.2)
ACO	Ant Colony Optimization (see Section 2.7.2.3)
ACP	Algorithm Configuration Problem (see Definition 5.1)
AC	Algorithm Configuration
ADHS	adaptive heuristic set
ADP	Algorithm Design Problem (see Section 3.1)
AILLA	adaptive iteration limited list-based threshold accepting
AI	Artificial Intelligence
ANOVA	Analysis of Variance
AR	Aggressive Racing (see Section 4.1.2, p. 113)
ASP	Algorithm Selection Problem
AS	Algorithm Selection

B & B	Branch & Bound (see Section 2.7.1.2)
BFS	Breath First Search (see Section 2.7.1.2)
BF	Brute-Force
BP	Bin Packing problem (see Section 2.2.1)
BSS	Best Single Solver
CHeSC	Cross-domain Heuristic Search Challenge (see Section 5.2.1.2)
CNF	Conjunctive Normal Form
CPU	Central Processing Unit
CT	Church-Turing thesis
CVRP	Capacitated Vehicle Routing Problem (see Section 2.2.1)
DAG	Directed Acyclic Graph
DFS	Depth First Search (see Section 2.7.1.2)
DLS	Dynamic Local Search (see Section 2.7.2.2)
DP	Dynamic Programming
Dr.	Doctor
e.g.	exempli gratia (latin) - for example
EA	Evolutionary Algorithm (see Section 2.7.2.3)
EI	Expected Improvement (criterion)
EMC	Exponential Monte Carlo (acceptance criterion)
ES	Evolution Strategies
et al.	et alia (latin) - and others
etc.	et cetera (latin) - and other similar things
ET	Eligibility Traces
FIFO	First In, First Out

FS-ILS	Fair Share Iterated Local Search (see Section 5.3)
GA	Genetic Algorithm (see Algorithm 7)
GGA	Gender-based Genetic Algorithm (see Section 4.1.2, p. 115)
GOP	Global function Optimization Problem (see Definition 2.4)
GP	Genetic Programming (see Section 3.3.2, p. 83)
GRASP	Greedy Randomized Adaptive Search Procedure (see Section 2.7.2.2)
HH	Hyper-Heuristic (see Section 2.7.2.5)
i.e.	id est (latin) - that is
i.i.d.	independent and identically distributed
ID	Instantaneous Description (see Section 2.3.1.2)
iff	if and only if (\iff)
II	Iterative Improvement (see Section 2.7.2.2)
ILS	Iterated Local Search (see Section 2.7.2.2)
input-ASP	Per-input Algorithm Selection Problem (see Definition 4.2)
ISAC	Instance-Specific Algorithm Configuration (see Section 4.2.3, p. 128)
IS	Importance Sampling, see Section 6.3.2.2
JoH	Journal of Heuristics
KP	Knapsack Problem (see Section 2.2.1)
LBS	Local Beam Search (see Section 2.7.2.3)
LCS	Learning Classifier Systems
LIFO	Last In, First Out
LVA	Las Vegas Algorithm
<i>m</i>-reducible	Many-one Reducible (see Definition 2.12)
MAC	Max-Cut problem (see Section 2.2.1)

MA	Memetic Algorithm (see Section 2.7.2.3)
MCMC	Markov Chain Monte Carlo
MC	Monte Carlo (RL technique)
MDP	Markov Decision Problem (see Definition 4.9)
MitL	Metaheuristics in the Large
ML	Machine Learning
MLP	Multi-layer Perceptron
MO	Multi-Objective
MSE	Mean Squared Error
NFL	No Free Lunch theorem (see Theorem 3.1)
NN	Neural Network (ML model)
NTM	Non-deterministic Turing Machine (see Definition 2.19)
OASC	Open Algorithm Selection Challenge
PbC	Programming by Configuration (see Section 5.1)
PbO	Programming by Optimization (see Section 5.1)
PEP	Performance Evaluation Problem (see Section 6.3.1)
PFS	Permutation Flow Shop problem (see Section 2.2.1)
PoC	Proof of Concept
Prof.	Professor
PSP	Personnel Scheduling Problem (see Section 2.2.1)
PS	Program Synthesis (see Section 3.3.2)
PTM	Probabilistic Turing Machine (see Definition 2.24)
QAP	Quadratic Assignment Problem (see Section 2.2.1)
RF	Random Forest (ML model)

RH	Relay Hybridization
RL	Reinforcement Learning (see Section 4.3.2.2)
ROAR	Random Order Aggressive Racing (see Section 4.1.2, p. 115)
RSA	Rivest-Shamir-Adleman (cryptographic system)
RTD	Run-Time Distribution (see Section 3.2.1)
SAT	boolean satisfiability problem (see Section 2.2.1)
SA	Simulated Annealing (see Section 2.7.2.2)
SBSE	Search-based Software Engineering
set-ASP	Per-set Algorithm Selection Problem (see Definition 4.1)
SE	Standard Error
SMAC	Sequential Model-based Algorithm Configuration (Section 4.1.2, p. 115)
SMBO	Sequential Model-Based Optimization
SR	Statistical Racing (see Section 4.1.2, p. 111)
SS	Single Selection
subset-ASP	Per-subset Algorithm Selection Problem (see Definition 4.3)
™	Trademark symbol
TM	(deterministic) Turing Machine (see Definition 2.8)
TreeOpt	tree optimization framework (see Algorithm 4)
TSP	Traveling Salesman Problem (see Section 2.2.1)
TS	Tabu Search (see Section 2.7.2.2)
URS	Uniform Random Selection
UTM	Universal Turing Machine
VBS	Virtual Best Solver
VLSNS	Very Large-Scale Neighborhood Search (see Section 2.7.2.2)

VND	Variable Neighborhood Descent (see Section 2.7.2.2)
VNS	Variable Neighborhood Search (see Section 2.7.2.2)
VRP	Vehicle Routing Problem (see Section 2.2.1)
vs.	versus
VUB	Vrije Universiteit Brussel
w.r.t.	with respect to
wb	white box
XOR	Either one or the other, not both (exclusive or).

Bibliography

- [Aaronson 2005] Scott Aaronson (2005). *Guest column: NP-complete problems and physical reality*. ACM Sigact News, 36(1), 30–52.
- [Abelson et al. 1996] Harold Abelson, Gerald Jay Sussman, and Julie Sussman (1996). *Structure and interpretation of computer programs*: Justin Kelly.
- [Addelman 1969] Sidney Addelman (1969). *The generalized randomized block design*. The American Statistician, 23(4), 35–36.
- [Adriaensen 2013] Steven Adriaensen (2013). *Robust Domain-Independent Heuristic Optimization*. Master’s thesis, Vrije Universiteit Brussel, Belgium.
- [Adriaensen et al. 2014a] Steven Adriaensen, Tim Brys, and Ann Nowé (2014a). *Designing reusable metaheuristic methods: A semi-automated approach*. In *Evolutionary Computation (CEC), 2014 IEEE Congress on*, pp. 2969–2976. IEEE.
- [Adriaensen et al. 2014b] Steven Adriaensen, Tim Brys, and Ann Nowé (2014b). *Fair-share ILS: a simple state-of-the-art iterated local search hyperheuristic*. In *Proceedings of the 2014 conference on Genetic and evolutionary computation*, pp. 1303–1310. ACM.
- [Adriaensen et al. 2017] Steven Adriaensen, Filip Moons, and Ann Nowe (2017). *An Importance Sampling Approach to the Estimation of Algorithm Performance in Automated Algorithm Design*. In *Proc. of LION-11*.

- [Adriaensen and Nowé 2016a] Steven Adriaensen and Ann Nowé (2016a). *Case study: An analysis of accidental complexity in a state-of-the-art hyper-heuristic for HyFlex*. In *Evolutionary Computation (CEC), 2016 IEEE Congress on*, pp. 1485–1492. IEEE.
- [Adriaensen and Nowé 2016b] Steven Adriaensen and Ann Nowé (2016b). *Towards a white box approach to automated algorithm design*. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, pp. 554–560.
- [Adriaensen et al. 2015] Steven Adriaensen, Gabriela Ochoa, and Ann Nowe (2015). *A Benchmark Set Extension and Comparative Study for the HyFlex Framework..* In *Evolutionary Computation (CEC), 2015 IEEE Congress on*, pp. 784–791. IEEE.
- [Agrawal et al. 2004] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena (2004). *PRIMES is in P*. *Annals of mathematics*, pp. 781–793.
- [Alpern et al. 1994] Bowen Alpern, Larry Carter, Ephraim Feig, and Ted Selker (1994). *The uniform memory hierarchy model of computation*. *Algorithmica*, 12(2), 72–109.
- [Ansel et al. 2009] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe (2009). *PetaBricks: a language and compiler for algorithmic choice*, vol. 44: ACM.
- [Ansótegui et al. 2015] Carlos Ansótegui, Yuri Malitsky, Horst Samulowitz, Meinolf Sellmann, and Kevin Tierney (2015). *Model-Based Genetic Algorithms for Algorithm Configuration..* In *IJCAI*, pp. 733–739.
- [Ansótegui et al. 2017] Carlos Ansótegui, Josep Pon, Meinolf Sellmann, and Kevin Tierney (2017). *Reactive Dialectic Search Portfolios for MaxSAT..* In *AAAI*, pp. 765–772.
- [Ansótegui et al. 2009] Carlos Ansótegui, Meinolf Sellmann, and Kevin Tierney (2009). *A gender-based genetic algorithm for the automatic configuration of algorithms*. In *International Conference on Principles and Practice of Constraint Programming*, pp. 142–157. Springer.
- [Audi 2015] Robert Audi (2015). *The Cambridge dictionary of philosophy*.
- [Audibert and Bubeck 2010] Jean-Yves Audibert and Sébastien Bubeck (2010). *Best arm identification in multi-armed bandits*. In *COLT-23th Conference on Learning Theory-2010*, pp. 13–p.
- [Ayob and Kendall 2003] Masri Ayob and Graham Kendall (2003). *A monte carlo hyper-heuristic to optimise component placement sequencing for multi head placement machine*. In *Proceedings of the international conference on intelligent technologies, InTech*, vol. 3, pp. 132–141. Citeseer.

- [Back 1996] Thomas Back (1996). *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*: Oxford university press.
- [Bader-El-Den and Poli 2007] Mohamed Bader-El-Den and Riccardo Poli (2007). *Generating SAT local-search heuristics using a GP hyper-heuristic framework*. In *International Conference on Artificial Evolution (Evolution Artificielle)*, pp. 37–49. Springer.
- [Bahnsen et al. 2015] Alejandro Correa Bahnsen, Djamila Aouada, and Björn Ottersten (2015). *Example-dependent cost-sensitive decision trees*. *Expert Systems with Applications*, 42(19), 6609–6619.
- [Balaprakash et al. 2007] Prasanna Balaprakash, Mauro Birattari, and Thomas Stützle (2007). *Improvement strategies for the F-Race algorithm: Sampling design and iterative refinement*. In *International workshop on hybrid metaheuristics*, pp. 108–122. Springer.
- [Basin et al. 2004] David Basin, Yves Deville, Pierre Flener, Andreas Hamfelt, and Jørgen Fischer Nilsson (2004). *Synthesis of programs in computational logic*. *Program Development in Computational Logic*, 3049, 30–65.
- [Battiti et al. 2008] Roberto Battiti, Mauro Brunato, and Franco Mascia (2008). *Reactive search and intelligent optimization*, vol. 45: Springer Science & Business Media.
- [Battiti and Tecchiolli 1994] Roberto Battiti and Giampietro Tecchiolli (1994). *The reactive tabu search*. *ORSA journal on computing*, 6(2), 126–140.
- [Bellman 1957] Richard Bellman (1957). *A Markovian decision process*. *Journal of Mathematics and Mechanics*, pp. 679–684.
- [Ben-David et al. 1989] Shai Ben-David, Benny Chor, and Oded Goldreich (1989). *On the theory of average case complexity*. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pp. 204–216. ACM.
- [Bengio et al. 2013] Yoshua Bengio, Aaron Courville, and Pascal Vincent (2013). *Representation learning: A review and new perspectives*. *IEEE transactions on pattern analysis and machine intelligence*, 35(8), 1798–1828.
- [Beranek 1961] William Beranek (1961). RONALD A. HOWARD “Dynamic Programming and Markov Processes”.
- [Bezerra et al. 2016] Leonardo CT Bezerra, Manuel López-Ibáñez, and Thomas Stützle (2016). *Automatic component-wise design of multiobjective evolutionary algorithms*. *IEEE Transactions on Evolutionary Computation*, 20(3), 403–417.

- [Biedenkapp et al. 2017] André Biedenkapp, Marius Thomas Lindauer, Katharina Eggen-sperger, Frank Hutter, Chris Fawcett, and Holger H Hoos (2017). *Efficient Parameter Importance Analysis via Ablation with Surrogates..* In *AAAI*, pp. 773–779.
- [Birattari and Dorigo 2004] Mauro Birattari and Marco Dorigo (2004). *The problem of tuning metaheuristics as seen from a machine learning perspective*. PhD thesis, Université libre de Bruxelles.
- [Birattari and Kacprzyk 2009] Mauro Birattari and Janusz Kacprzyk (2009). *Tuning metaheuristics: a machine learning perspective*, vol. 197: Springer.
- [Birattari et al. 2002] Mauro Birattari, Thomas Stützle, Luis Paquete, and Klaus Varren-trapp (2002). *A racing algorithm for configuring metaheuristics*. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, pp. 11–18. Morgan Kaufmann Publishers Inc.
- [Birattari et al. 2010] Mauro Birattari, Zhi Yuan, Prasanna Balaprakash, and Thomas Stützle (2010). *F-Race and iterated F-Race: An overview*. In *Experimental methods for the analysis of optimization algorithms*, pp. 311–336: Springer.
- [Birbil and Fang 2003] Ş İlker Birbil and Shu-Chering Fang (2003). *An electromagnetism-like mechanism for global optimization*. *Journal of global optimization*, 25(3), 263–282.
- [Bischi et al. 2016] Bernd Bischi, Pascal Kerschke, Lars Kotthoff, Marius Lindauer, Yuri Malitsky, Alexandre Fréchette, Holger Hoos, Frank Hutter, Kevin Leyton-Brown, Kevin Tierney, et al. (2016). *Aslib: A benchmark library for algorithm selection*. *Artificial Intelligence*, 237, 41–58.
- [Blass and Gurevich 2003] Andreas Blass and Yuri Gurevich (2003). *Algorithms: A quest for absolute definitions*. *Bulletin of the EATCS*, 81, 195–225.
- [Blot et al. 2016] Aymeric Blot, Holger H Hoos, Laetitia Jourdan, Marie-Éléonore Kessaci-Marmion, and Heike Trautmann (2016). *MO-ParamILS: a multi-objective automatic algorithm configuration framework*. In *International Conference on Learning and Intelligent Optimization*, pp. 32–47. Springer.
- [Boryczka 2002] Mariusz Boryczka (2002). *Ant colony programming for approximation problems*. In *Intelligent Information Systems 2002*, pp. 147–156: Springer.
- [Brefeld et al. 2003] Ulf Brefeld, Peter Geibel, and Fritz Wysotzki (2003). *Support vector machines with example dependent costs*. In *European Conference on Machine Learning*, pp. 23–34. Springer.

- [Breiman 2001] Leo Breiman (2001). *Random forests*. Machine learning, 45(1), 5–32.
- [Brunner 2010] Jens O Brunner (2010). *Literature Review on Personnel Scheduling*. Flexible Shift Planning in the Service Industry, pp. 5–12.
- [Brys 2016] Tim Brys (2016). *Reinforcement Learning with Heuristic Information*. PhD thesis, Vrije Universiteit Brussel.
- [Burkard et al. 1997] Rainer E Burkard, Stefan E Karisch, and Franz Rendl (1997). *QAPLIB—a quadratic assignment problem library*. Journal of Global optimization, 10(4), 391–403.
- [Burke et al. 2010a] Edmund Burke, Timothy Curtois, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Sanja Petrovic, José A Valázquez-Rodríguez, and Michel Gendreau (2010a). *Iterated local search vs. hyper-heuristics: Towards general-purpose search algorithms*. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pp. 1–8. IEEE.
- [Burke et al. 2003] Edmund Burke, Graham Kendall, Jim Newall, Emma Hart, Peter Ross, and Sonia Schulenburg (2003). *Hyper-heuristics: An emerging direction in modern search technology*. Handbook of metaheuristics, pp. 457–474.
- [Burke and Bykov 2008] Edmund K Burke and Yuri Bykov (2008). *A late acceptance strategy in hill-climbing for exam timetabling problems*. In *PATAT 2008 Conference, Montreal, Canada*.
- [Burke et al. 2013] Edmund K Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and Rong Qu (2013). *Hyper-heuristics: A survey of the state of the art*. Journal of the Operational Research Society, 64(12), 1695–1724.
- [Burke et al. 2010b] Edmund K Burke, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and John R Woodward (2010b). *A classification of hyper-heuristic approaches*. In *Handbook of metaheuristics*, pp. 449–468: Springer.
- [Carchrae and Beck 2005] Tom Carchrae and J Christopher Beck (2005). *Applying Machine Learning to Low-knowledge Control Of Optimization Algorithms*. Computational Intelligence, 21(4), 372–387.
- [Christofides 1976] Nicos Christofides (1976). *Worst-case analysis of a new heuristic for the travelling salesman problem*. Technical Report, Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group.

- [Church 1932] Alonzo Church (1932). *A set of postulates for the foundation of logic*. *Annals of mathematics*, 33(2), 346–366.
- [Church 1963] Alonzo Church (1963). *Application of recursive arithmetic to the problem of circuit synthesis*.
- [Collberg et al. 2015] Christian Collberg, Todd Proebsting, and Alex M Warren (2015). *Repeatability and benefaction in computer systems research*. University of Arizona TR 14, 4.
- [Colquhoun 2003] David Colquhoun (2003). *Challenging the tyranny of impact factors*. *Nature*, 423(6939), 479.
- [Colton and Wiggins 2012] Simon Colton and Geraint A Wiggins (2012). *Computational creativity: The final frontier?*. In *Proceedings of the 20th European conference on artificial intelligence*, pp. 21–26. IOS Press.
- [Cook 1971] Stephen A Cook (1971). *The complexity of theorem-proving procedures*. In *Proceedings of the third annual ACM symposium on Theory of computing*, pp. 151–158. ACM.
- [Cook 1979] Stephen A Cook (1979). *Deterministic CFL's are accepted simultaneously in polynomial time and log squared space*. In *Proceedings of the eleventh annual ACM symposium on Theory of computing*, pp. 338–345. ACM.
- [Cooke 2011] Roger L Cooke (2011). *The history of mathematics: A brief course*. John Wiley & Sons.
- [Cowling et al. 2000] Peter Cowling, Graham Kendall, and Eric Soubeiga (2000). *A hyperheuristic approach to scheduling a sales summit*. In *International Conference on the Practice and Theory of Automated Timetabling*, pp. 176–190. Springer.
- [Cramer 1985] Michael Lynn Cramer (1985). *A representation for the adaptive generation of simple sequential programs*. In *Proceedings of the First International Conference on Genetic Algorithms*, pp. 183–187.
- [Curtois et al. 2009] Tim Curtois, Gabriela Ochoa, Matthew Hyde, and José Antonio Vázquez-Rodríguez (2009). *A hyflex module for the personnel scheduling problem*. School of Computer Science, University of Nottingham, Tech. Rep.
- [DaCosta et al. 2008] Luis DaCosta, Alvaro Fialho, Marc Schoenauer, and Michèle Sebag (2008). *Adaptive operator selection with dynamic multi-armed bandits*. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pp. 913–920. ACM.

- [Dang et al. 2017] Nguyen Dang, Leslie Pérez Cáceres, Patrick De Causmaecker, and Thomas Stützle (2017). *Configuring irace using surrogate configuration benchmarks*. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 243–250. ACM.
- [De Boer et al. 2005] Pieter-Tjerk De Boer, Dirk P Kroese, Shie Mannor, and Reuven Y Rubinstein (2005). *A tutorial on the cross-entropy method*. *Annals of operations research*, 134(1), 19–67.
- [Dean and Boddy 1988] Thomas L Dean and Mark S Boddy (1988). *An Analysis of Time-Dependent Planning*. In *AAAI*, vol. 88, pp. 49–54.
- [Di Gaspero and Urli 2012] Luca Di Gaspero and Tommaso Urli (2012). *Evaluation of a family of reinforcement learning cross-domain optimization heuristics*. In *Learning and Intelligent Optimization*, pp. 384–389: Springer.
- [Ding et al. 2015] Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May O'Reilly, and Saman Amarasinghe (2015). *Autotuning algorithmic choice for input sensitivity*. In *ACM SIGPLAN Notices*, vol. 50, pp. 379–390. ACM.
- [Dorigo 1992] Marco Dorigo (1992). *Optimization, learning and natural algorithms*. Ph. D. Thesis, Politecnico di Milano, Italy.
- [Dorigo et al. 2006] Marco Dorigo, Mauro Birattari, and Thomas Stutzle (2006). *Ant colony optimization*. *IEEE computational intelligence magazine*, 1(4), 28–39.
- [Eggensperger et al. 2018] Katharina Eggensperger, Marius Lindauer, Holger H Hoos, Frank Hutter, and Kevin Leyton-Brown (2018). *Efficient benchmarking of algorithm configurators via model-based surrogates*. *Machine Learning*, pp. 1–27.
- [Eiben et al. 1999] Ágoston E Eiben, Robert Hinterding, and Zbigniew Michalewicz (1999). *Parameter control in evolutionary algorithms*. *IEEE Transactions on evolutionary computation*, 3(2), 124–141.
- [Eiben et al. 2007] Agoston E Eiben, Zbigniew Michalewicz, Marc Schoenauer, and James E Smith (2007). *Parameter control in evolutionary algorithms*. In *Parameter setting in evolutionary algorithms*, pp. 19–46: Springer.
- [Eskandar et al. 2012] Hadi Eskandar, Ali Sadollah, Ardeshir Bahreininejad, and Mohd Hamdi (2012). *Water cycle algorithm—A novel metaheuristic optimization method for solving constrained engineering optimization problems*. *Computers & Structures*, 110, 151–166.

- [Eusuff and Lansey 2003] Muzaffar M Eusuff and Kevin E Lansey (2003). *Optimization of water distribution network design using the shuffled frog leaping algorithm*. Journal of Water Resources planning and management, 129(3), 210–225.
- [Fanelli 2011] Daniele Fanelli (2011). *Negative results are disappearing from most disciplines and countries*. Scientometrics, 90(3), 891–904.
- [Fawcett and Hoos 2016] Chris Fawcett and Holger H Hoos (2016). *Analysing differences between algorithm configurations through ablation*. Journal of Heuristics, 22(4), 431–458.
- [Fawcett et al. 2009] Chris Fawcett, Holger H Hoos, and Marco Chiarandini (2009). *An automatically configured modular algorithm for post enrollment course timetabling*. University of British Columbia, Department of Computer Science, Tech. Rep.
- [Feo and Resende 1995] Thomas A Feo and Mauricio GC Resende (1995). *Greedy randomized adaptive search procedures*. Journal of global optimization, 6(2), 109–133.
- [Fink 1998] Eugene Fink (1998). *How to Solve It Automatically: Selection Among Problem Solving Methods..* In *AIPS*, pp. 128–136.
- [Flach 1994] Peter A Flach (1994). *Simply Logical intelligent reasoning by example*.
- [Flourens 1842] Pierre Flourens (1842). *Recherches expérimentales sur les propriétés et les fonctions du système nerveux dans les animaux vertébrés*: Ballière.
- [Fortnow 2009] Lance Fortnow (2009). *The status of the P versus NP problem*. Communications of the ACM, 52(9), 78–86.
- [Fowler and Beck 1999] Martin Fowler and Kent Beck (1999). *Refactoring: improving the design of existing code*: Addison-Wesley Professional.
- [Fukunaga 2004] Alex S Fukunaga (2004). *Evolving local search heuristics for SAT using genetic programming*. In *Genetic and Evolutionary Computation Conference*, pp. 483–494. Springer.
- [Gagliolo and Schmidhuber 2006] Matteo Gagliolo and Jürgen Schmidhuber (2006). *Learning dynamic algorithm portfolios*. Annals of Mathematics and Artificial Intelligence, 47(3-4), 295–328.
- [Gathercole and Ross 1994] Chris Gathercole and Peter Ross (1994). *Dynamic training subset selection for supervised learning in genetic programming*. Parallel Problem Solving from Nature-PPSN III, pp. 312–321.

- [Geem et al. 2001] Zong Woo Geem, Joong Hoon Kim, and Gobichettipalayam Vasudevan Loganathan (2001). *A new heuristic optimization algorithm: harmony search*. simulation, 76(2), 60–68.
- [Glover 1989] Fred Glover (1989). *Tabu search - part I*. ORSA Journal on computing, 1(3), 190–206.
- [Gödel 1931] Kurt Gödel (1931). *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I*. Monatshefte für mathematik und physik, 38(1), 173–198.
- [Goetschalckx and Ramon 2007] Robby Goetschalckx and Jan Ramon (2007). *On policy learning in restricted policy spaces*. In *Proceedings of the 22nd national conference on Artificial intelligence-Volume 2*, pp. 1858–1859. AAAI Press.
- [Goldreich 2008] Oded Goldreich (2008). *Computational Complexity: A Conceptual Perspective*. US: Cambridge University Press.
- [Gomes and Selman 2001] Carla P Gomes and Bart Selman (2001). *Algorithm portfolios*. Artificial Intelligence, 126(1-2), 43–62.
- [Goss et al. 1989] Simon Goss, Serge Aron, Jean-Louis Deneubourg, and Jacques Marie Pasteels (1989). *Self-organized shortcuts in the Argentine ant*. Naturwissenschaften, 76(12), 579–581.
- [Graves et al. 2014] Alex Graves, Greg Wayne, and Ivo Danihelka (2014). *Neural turing machines*. arXiv preprint arXiv:1410.5401.
- [Gulwani et al. 2017] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. (2017). *Program synthesis*. Foundations and Trends® in Programming Languages, 4(1-2), 1–119.
- [Hamerly et al. 2003] Greg Hamerly, Charles Elkan, et al. (2003). *Learning the k in k-means*. In *NIPS*, vol. 3, pp. 281–288.
- [Hamfelt et al. 2001] Andreas Hamfelt, Jørgen Fischer Nilsson, and Nikolaj Oldager (2001). *Logic program synthesis as problem reduction using combining forms*. Automated Software Engineering, 8(2), 167–193.
- [Hansen and Ostermeier 1996] Nikolaus Hansen and Andreas Ostermeier (1996). *Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaptation*. In *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on*, pp. 312–317. IEEE.

- [Harik and Lobo 1999] Georges R Harik and Fernando G Lobo (1999). *A parameter-less genetic algorithm*. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation-Volume 1*, pp. 258–265. Morgan Kaufmann Publishers Inc.
- [Harman and Jones 2001] Mark Harman and Bryan F Jones (2001). *Search-based software engineering*. Information and software Technology, 43(14), 833–839.
- [Hart et al. 1968] Peter E Hart, Nils J Nilsson, and Bertram Raphael (1968). *A formal basis for the heuristic determination of minimum cost paths*. Systems Science and Cybernetics, IEEE Transactions on, 4(2), 100–107.
- [Hellman 1980] Martin Hellman (1980). *A cryptanalytic time-memory trade-off*. IEEE transactions on Information Theory, 26(4), 401–406.
- [Hesterberg 1988] Timothy Classen Hesterberg (1988). *Advances in importance sampling*. PhD thesis, Stanford University.
- [Hooker 1995] John N Hooker (1995). *Testing heuristics: We have it all wrong*. Journal of heuristics, 1(1), 33–42.
- [Hoos 2012] Holger H Hoos (2012). *Programming by optimization*. Communications of the ACM, 55(2), 70–80.
- [Hoos and Stützle 1998] Holger H Hoos and Thomas Stützle (1998). *Evaluating las vegas algorithms: pitfalls and remedies*. In *Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence*, pp. 238–245. Morgan Kaufmann Publishers Inc.
- [Hoos and Stützle 2004] Holger H Hoos and Thomas Stützle (2004). *Stochastic local search: Foundations and applications*. Elsevier.
- [Hopcroft et al. 2006] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman (2006). *Automata theory, languages, and computation*. International Edition, 24.
- [Horvitz et al. 2001] Eric Horvitz, Yongshao Ruan, Carla Gomes, Henry Kautz, Bart Selman, and Max Chickering (2001). *A Bayesian approach to tackling hard computational problems*. In *Proceedings of the Seventeenth conference on Uncertainty in artificial intelligence*, pp. 235–244. Morgan Kaufmann Publishers Inc.
- [Hsiao et al. 2012] Ping-Che Hsiao, Tsung-Che Chiang, and Li-Chen Fu (2012). *A vns-based hyper-heuristic with adaptive computational budget of local search*. In *Evolutionary Computation (CEC), 2012 IEEE Congress on*, pp. 1–8. IEEE.
- [Hsu 1996] Jason Hsu (1996). *Multiple comparisons: theory and methods*. CRC Press.

- [Huberman et al. 1997] Bernardo A Huberman, Rajan M Lukose, and Tad Hogg (1997). *An economics approach to hard computational problems*. Science, 275(5296), 51–54.
- [Hunt and Thomas 2001] Andrew Hunt and David Thomas (2001). *The Art in Computer Programming*. The Pragmatic Programmers, LLC.
- [Hutter et al. 2007a] Frank Hutter, Domagoj Babic, Holger H Hoos, and Alan J Hu (2007a). *Boosting verification by automatic tuning of decision procedures*. In *Formal Methods in Computer Aided Design, 2007. FMCAD'07*, pp. 27–34. IEEE.
- [Hutter and Hamadi 2005] Frank Hutter and Youssef Hamadi (2005). *Parameter adjustment based on performance prediction: Towards an instance-aware problem solver*. Microsoft Research, Tech. Rep. MSR-TR-2005-125.
- [Hutter et al. 2006] Frank Hutter, Youssef Hamadi, Holger H Hoos, and Kevin Leyton-Brown (2006). *Performance prediction and automated tuning of randomized and parametric algorithms*. In *International Conference on Principles and Practice of Constraint Programming*, pp. 213–228. Springer.
- [Hutter et al. 2010] Frank Hutter, Holger Hoos, and Kevin Leyton-Brown (2010). *Automated configuration of mixed integer programming solvers*. Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, pp. 186–202.
- [Hutter et al. 2011] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown (2011). *Sequential model-based optimization for general algorithm configuration*. In *International Conference on Learning and Intelligent Optimization*, pp. 507–523. Springer.
- [Hutter et al. 2009] Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Thomas Stützle (2009). *ParamILS: an automatic algorithm configuration framework*. Journal of Artificial Intelligence Research, 36(1), 267–306.
- [Hutter et al. 2007b] Frank Hutter, Holger H Hoos, and Thomas Stützle (2007b). *Automatic algorithm configuration based on local search*. In *AAAI*, vol. 7, pp. 1152–1157.
- [Hutter et al. 2014] Frank Hutter, Manuel López-Ibáñez, Chris Fawcett, Marius Lindauer, Holger H Hoos, Kevin Leyton-Brown, and Thomas Stützle (2014). *Aclib: a benchmark library for algorithm configuration*. In *International Conference on Learning and Intelligent Optimization*, pp. 36–40. Springer.
- [Hyde et al. 2009] Matthew Hyde, Gabriela Ochoa, T Curtois, and JA Vázquez-Rodríguez (2009). *A hyflex module for the one dimensional bin-packing problem*. School of Computer Science, University of Nottingham, Tech. Rep.

- [Hyde et al. 2011] Matthew Hyde, Gabriela Ochoa, José Antonio Vázquez-Rodríguez, and Tim Curtois (2011). *A HyFlex Module for the MAX-SAT Problem*. University of Nottingham, Tech. Rep.
- [Igel and Toussaint 2005] Christian Igel and Marc Toussaint (2005). *A no-free-lunch theorem for non-uniform distributions of target functions*. *Journal of Mathematical Modelling and Algorithms*, 3(4), 313–322.
- [Ingber 1996] Lester Ingber (1996). *Adaptive simulated annealing (ASA): lessons learned*. *Control and Cybernetics*, 25(1).
- [Johnson et al. 1879] Wm Woolsey Johnson, William E Story, et al. (1879). *Notes on the 15 puzzle*. *American Journal of Mathematics*, 2(4), 397–404.
- [Jones et al. 1998] Donald R Jones, Matthias Schonlau, and William J Welch (1998). *Efficient global optimization of expensive black-box functions*. *Journal of Global optimization*, 13(4), 455–492.
- [Jünger et al. 1995] Michael Jünger, Gerhard Reinelt, and Giovanni Rinaldi (1995). *The traveling salesman problem*. *Handbooks in operations research and management science*, 7, 225–330.
- [Kadioglu et al. 2010] Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney (2010). *ISAC-Instance-Specific Algorithm Configuration..* In *ECAI*, vol. 215, pp. 751–756.
- [Kadioglu et al. 2017] Serdar Kadioglu, Meinolf Sellmann, and Markus Wagner (2017). *Learning a Reactive Restart Strategy to Improve Stochastic Search*. In *International Conference on Learning and Intelligent Optimization*, pp. 109–123. Springer.
- [Kahn and Marshall 1953] Herman Kahn and Andy W Marshall (1953). *Methods of reducing sample size in Monte Carlo computations*. *Journal of the Operations Research Society of America*, 1(5), 263–278.
- [Karaboga 2005] Dervis Karaboga (2005). *An idea based on honey bee swarm for numerical optimization*. Technical Report, Technical report-tr06, Erciyes university, engineering faculty, computer engineering department.
- [Karafotias et al. 2014] Giorgos Karafotias, Agoston Endre Eiben, and Mark Hoogendoorn (2014). *Generic parameter control with reinforcement learning*. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pp. 1319–1326. ACM.

- [Karafotias et al. 2013] Giorgos Karafotias, Mark Hoogendoorn, and AE Eiben (2013). *Why parameter control mechanisms should be benchmarked against random variation*. In *Evolutionary Computation (CEC), 2013 IEEE Congress on*, pp. 349–355. IEEE.
- [Karp 1972] Richard M Karp (1972). *Reducibility among combinatorial problems*. In *Complexity of computer computations*, pp. 85–103: Springer.
- [Kautz et al. 2002] Henry Kautz, Eric Horvitz, Yongshao Ruan, Carla Gomes, and Bart Selman (2002). *Dynamic restart policies*. *Aaai/iaai*, 97, 674–681.
- [KhudaBukhsh et al. 2009] Ashiqur R KhudaBukhsh, Lin Xu, Holger H Hoos, and Kevin Leyton-Brown (2009). *SATenstein: Automatically Building Local Search SAT Solvers from Components*. In *IJCAI*, vol. 9, pp. 517–524.
- [Kirkpatrick et al. 1983] Scott Kirkpatrick, C Daniel Gelatt, Mario P Vecchi, et al. (1983). *Optimization by simulated annealing*. *science*, 220(4598), 671–680.
- [Kish 1965] Leslie Kish (1965). *Survey Sampling Wiley*. New York.
- [Kleene 1936] Stephen Cole Kleene (1936). *General recursive functions of natural numbers*. *Mathematische Annalen*, 112(1), 727–742.
- [Knuth 1968] Donald E Knuth (1968). *The art of computer programming (ongoing series)*.
- [Knuth 1976] Donald E Knuth (1976). *Big omicron and big omega and big theta*. *ACM Sigact News*, 8(2), 18–24.
- [Koulamas et al. 1994] C Koulamas, SR Antony, and R Jaen (1994). *A survey of simulated annealing applications to operations research problems*. *Omega*, 22(1), 41–56.
- [Koza 1992] John R Koza (1992). *Genetic programming: on the programming of computers by means of natural selection*, vol. 1: MIT press.
- [Krall 1998] Andreas Krall (1998). *Efficient JavaVM just-in-time compilation*. In *Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on*, pp. 205–212. IEEE.
- [Krishnanand and Ghose 2009] KN Krishnanand and Debasish Ghose (2009). *Glowworm swarm optimization for simultaneous capture of multiple local optima of multimodal functions*. *Swarm intelligence*, 3(2), 87–124.
- [Lagoudakis and Littman 2000] Michail G Lagoudakis and Michael L Littman (2000). *Algorithm Selection using Reinforcement Learning*. In *ICML*, pp. 511–518.

- [Lam and Li 2010] Albert YS Lam and Victor OK Li (2010). *Chemical-reaction-inspired metaheuristic for optimization*. IEEE Transactions on Evolutionary Computation, 14(3), 381–399.
- [Langton 1997] Christopher G Langton (1997). *Artificial life: An overview*: Mit Press.
- [Lanzi 2002] Pier Luca Lanzi (2002). *Learning classifier systems from a reinforcement learning perspective*. Soft Computing, 6(3-4), 162–170.
- [Laporte 1992] Gilbert Laporte (1992). *The vehicle routing problem: An overview of exact and approximate algorithms*. European Journal of Operational Research, 59(3), 345–358.
- [Larsen and Von Ins 2010] Peder Olesen Larsen and Markus Von Ins (2010). *The rate of growth in scientific publication and the decline in coverage provided by Science Citation Index*. Scientometrics, 84(3), 575–603.
- [Lawler and Wood 1966] Eugene L Lawler and David E Wood (1966). *Branch-and-bound methods: A survey*. Operations research, 14(4), 699–719.
- [LeCun et al. 2015] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton (2015). *Deep learning*. Nature, 521(7553), 436–444.
- [Leister 2014] Tom Leister (2014). *Effective Sample Size*. https://golem.ph.utexas.edu/category/2014/12/effective_sample_size.html. Accessed: 2018-04-11.
- [Lenstra and Kan 1975] Jan K Lenstra and AHG Rinnooy Kan (1975). *Some simple applications of the travelling salesman problem*. Journal of the Operational Research Society, 26(4), 717–733.
- [Leyton-Brown et al. 2003] Kevin Leyton-Brown, Eugene Nudelman, Galen Andrew, Jim McFadden, and Yoav Shoham (2003). *A portfolio approach to algorithm selection*. In *IJCAI*, vol. 1543, p. 2003.
- [Leyton-Brown et al. 2002] Kevin Leyton-Brown, Eugene Nudelman, and Yoav Shoham (2002). *Learning the empirical hardness of optimization problems: The case of combinatorial auctions*. In *International Conference on Principles and Practice of Constraint Programming*, pp. 556–572. Springer.
- [Leyton-Brown et al. 2009] Kevin Leyton-Brown, Eugene Nudelman, and Yoav Shoham (2009). *Empirical hardness models: Methodology and a case study on combinatorial auctions*. Journal of the ACM (JACM), 56(4), 22.

- [Li et al. 2006] Lihong Li, Thomas J Walsh, and Michael L Littman (2006). *Towards a Unified Theory of State Abstraction for MDPs..* In *ISAIM*.
- [Lientz and Swanson 1981] Bennet P Lientz and E Burton Swanson (1981). *Problems in application software maintenance*. Communications of the ACM, 24(11), 763–769.
- [Lin and Kernighan 1973] Shen Lin and Brian W Kernighan (1973). *An effective heuristic algorithm for the traveling-salesman problem*. Operations research, 21(2), 498–516.
- [Lindauer et al. 2016] Marius Lindauer, Rolf-David Bergdoll, and Frank Hutter (2016). *An empirical study of per-instance algorithm scheduling*. In *International Conference on Learning and Intelligent Optimization*, pp. 253–259. Springer.
- [Lindauer et al. 2015] Marius Lindauer, Holger Hoos, and Frank Hutter (2015). *From sequential algorithm selection to parallel portfolio selection*. In *International Conference on Learning and Intelligent Optimization*, pp. 1–16. Springer.
- [Liskov and Wing 1994] Barbara H Liskov and Jeannette M Wing (1994). *A behavioral notion of subtyping*. ACM Transactions on Programming Languages and Systems (TOPLAS), 16(6), 1811–1841.
- [Liu and Motoda 1998] Huan Liu and Hiroshi Motoda (1998). *Feature extraction, construction and selection: A data mining perspective*, vol. 453: Springer Science & Business Media.
- [López-Ibáñez et al. 2016] Manuel López-Ibáñez, L Pérez Cáceres, Jérémie Dubois-Lacoste, Thomas Stützle, and Mauro Birattari (2016). *The irace package: User guide*. IRIDIA, Université Libre de Bruxelles, Belgium, Tech. Rep. TR/IRIDIA/2016-004.
- [López-Ibáñez et al. 2011] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Thomas Stützle, and Mauro Birattari (2011). *The irace package, iterated race for automatic algorithm configuration*. Technical Report, Citeseer.
- [López-Ibáñez and Stützle 2010] Manuel López-Ibáñez and Thomas Stützle (2010). *Automatic Configuration of Multi-Objective ACO Algorithms..* In *ANTS Conference*, pp. 95–106. Springer.
- [Lourencço et al. 2003] Helena R Lourencço, Olivier C Martin, and Thomas Stutzle (2003). *Iterated local search*. International series in operations research and management science, pp. 321–354.
- [Lukasik and Zak 2009] Szymon Lukasik and Slawomir Zak (2009). *Firefly Algorithm for Continuous Constrained Optimization Tasks..* In *ICCCI*, pp. 97–106. Springer.

- [Luke 2009] Sean Luke (2009). *Essentials of metaheuristics*, vol. 113: Lulu Raleigh.
- [Mahmood et al. 2014] A Rupam Mahmood, van Hado P Hasselt, and Richard S Sutton (2014). *Weighted importance sampling for off-policy learning with linear function approximation*. In *Advances in Neural Information Processing Systems*, pp. 3014–3022.
- [Malitsky and Sellmann 2010] Yuri Malitsky and Meinolf Sellmann (2010). *Stochastic offline programming*. *International Journal on Artificial Intelligence Tools*, 19(04), 351–371.
- [Manna and Waldinger 1980] Zohar Manna and Richard Waldinger (1980). *A deductive approach to program synthesis*. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1), 90–121.
- [Marmion et al. 2013] Marie-Éléonore Marmion, Franco Mascia, Manuel López-Ibáñez, and Thomas Stützle (2013). *Automatic design of hybrid stochastic local search algorithms*. In *International Workshop on Hybrid Metaheuristics*, pp. 144–158. Springer.
- [Maron and Moore 1994] Oded Maron and Andrew W Moore (1994). *Hoeffding races: Accelerating model selection search for classification and function approximation*. *Advances in neural information processing systems*, pp. 59–59.
- [Maron and Moore 1997] Oded Maron and Andrew W Moore (1997). *The racing algorithm: Model selection for lazy learners*. In *Lazy learning*, pp. 193–225: Springer.
- [Martello et al. 1999] Silvano Martello, David Pisinger, and Paolo Toth (1999). *Dynamic programming and strong bounds for the 0-1 knapsack problem*. *Management Science*, 45(3), 414–424.
- [Mascia et al. 2014a] Franco Mascia, Manuel López-Ibáñez, Jérémie Dubois-Lacoste, and Thomas Stützle (2014a). *Grammar-based generation of stochastic local search heuristics through automatic algorithm configuration tools*. *Computers & operations research*, 51, 190–199.
- [Mascia et al. 2014b] Franco Mascia, Paola Pellegrini, Mauro Birattari, and Thomas Stützle (2014b). *An analysis of parameter adaptation in reactive tabu search*. *International Transactions in Operational Research*, 21(1), 127–152.
- [Mascia and Stützle 2012] Franco Mascia and Thomas Stützle (2012). *A non-adaptive stochastic local search algorithm for the chesc 2011 competition*. In *Learning and Intelligent Optimization*, pp. 101–114: Springer.

- [Maslow 1966] Abraham H. Maslow (1966). *The Psychology of Science*. New York: Harper & Row.
- [Mazure et al. 1997] Bertrand Mazure, Lakhdar Sais, and Éric Grégoire (1997). *Tabu search for SAT*. In *AAAI/IAAI*, pp. 281–285.
- [Meignan 2011] David Meignan (2011). *An Evolutionary Programming Hyper-heuristic with Co-evolution for CHESc 2011*. In *OR53 Annual Conference*.
- [Minsky 1961] M. Minsky (1961). *Steps toward Artificial Intelligence*. Proceedings of the IRE, 49(1), 8–30.
- [Mirjalili and Lewis 2016] Seyedali Mirjalili and Andrew Lewis (2016). *The whale optimization algorithm*. *Advances in Engineering Software*, 95, 51–67.
- [Mirjalili et al. 2014] Seyedali Mirjalili, Seyed Mohammad Mirjalili, and Andrew Lewis (2014). *Grey wolf optimizer*. *Advances in Engineering Software*, 69, 46–61.
- [Misir et al. 2012] M Misir, Katja Verbeeck, Patrick De Causmaecker, and G Vanden Berghe (2012). *The effect of the set of low-level heuristics on the performance of selection hyper-heuristics*. In *International Conference on Parallel Problem Solving from Nature*, pp. 408–417. Springer.
- [Misir et al. 2011] Mustafa Misir, Patrick De Causmaecker, Greet Vanden Berghe, and Katja Verbeeck (2011). *An adaptive hyper-heuristic for CHESc 2011*.
- [Misir et al. 2012a] Mustafa Misir, Patrick De Causmaecker, Katja Verbeeck, and Greet Vanden Berghe (2012a). *Intelligent Hyper-heuristics: A Tool for Solving Generic Optimisation Problems*. PhD thesis
- [Misir et al. 2012b] Mustafa Misir, Katja Verbeeck, Patrick De Causmaecker, and Greet Vanden Berghe (2012b). *An intelligent hyper-heuristic framework for chesc 2011*. In *Learning and Intelligent Optimization*, pp. 461–466: Springer.
- [Mitchell 1980] Tom M Mitchell (1980). *The need for biases in learning generalizations*: Department of Computer Science, Laboratory for Computer Science Research, Rutgers Univ. New Jersey.
- [Mladenović and Hansen 1997] Nenad Mladenović and Pierre Hansen (1997). *Variable neighborhood search*. *Computers & operations research*, 24(11), 1097–1100.
- [Mladenović et al. 2016] Nenad Mladenović, Raca Todosijević, and Dragan Urošević (2016). *Less is more: basic variable neighborhood search for minimum differential dispersion problem*. *Information Sciences*, 326, 160–171.

- [Mohri et al. 2012] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar (2012). *Foundations of machine learning*: MIT press.
- [Moscato and Cotta 2002] Pablo Moscato and C Cotta (2002). *Memetic algorithms*. Handbook of Applied Optimization, pp. 157–167.
- [Mucherino and Seref 2007] Antonio Mucherino and Onur Seref (2007). *Monkey search: a novel metaheuristic search for global optimization*. In *AIP conference proceedings*, vol. 953, pp. 162–173. AIP.
- [Nelson and Gailly 1995] Mark Nelson and Jean-Loup Gailly (1995). *The data compression book 2nd edition*. M & T Books, New York, NY.
- [Nesterov and Nemirovskii 1994] Yurii Nesterov and Arkadii Nemirovskii (1994). *Interior-point polynomial algorithms in convex programming*: SIAM.
- [Ng et al. 1999] Andrew Y Ng, Daishi Harada, and Stuart Russell (1999). *Policy invariance under reward transformations: Theory and application to reward shaping*. In *ICML*, vol. 99, pp. 278–287.
- [Nudelman et al. 2004] Eugene Nudelman, Kevin Leyton-Brown, Holger H Hoos, Alex Devkar, and Yoav Shoham (2004). *Understanding random SAT: Beyond the clauses-to-variables ratio*. In *International Conference on Principles and Practice of Constraint Programming*, pp. 438–452. Springer.
- [Ochoa et al. 2012] Gabriela Ochoa, Matthew Hyde, Tim Curtois, Jose Vazquez-Rodriguez, James Walker, Michel Gendreau, Graham Kendall, Barry McCollum, Andrew Parkes, Sanja Petrovic, et al. (2012). *HyFlex: a benchmark framework for cross-domain heuristic search*. Evolutionary Computation in Combinatorial Optimization, pp. 136–147.
- [Oshiro et al. 2012] Thais Mayumi Oshiro, Pedro Santoro Perez, and José Augusto Baranauskas (2012). *How many trees in a random forest?*. In *MLDM*, pp. 154–168. Springer.
- [Owen 2014] AB Owen (2014). *Monte Carlo theory, methods and examples (book draft)*.
- [Pan 2012] Wen-Tsao Pan (2012). *A new fruit fly optimization algorithm: taking the financial distress model as an example*. Knowledge-Based Systems, 26, 69–74.
- [Papadimitriou 1994] Christos H Papadimitriou (1994). *On the complexity of the parity argument and other inefficient proofs of existence*. Journal of Computer and system Sciences, 48(3), 498–532.

- [Parnas 1985] David Lorge Parnas (1985). *Software aspects of strategic defense systems*. Communications of the ACM, 28(12), 1326–1335.
- [Passino 2002] Kevin M Passino (2002). *Biomimicry of bacterial foraging for distributed optimization and control*. IEEE control systems, 22(3), 52–67.
- [Precup et al. 2000] Doina Precup, Richard S Sutton, and Satinder P Singh (2000). *Eligibility Traces for Off-Policy Policy Evaluation..* In *ICML*, pp. 759–766. Citeseer.
- [Puterman 2014] Martin L Puterman (2014). *Markov decision processes: discrete stochastic dynamic programming*: John Wiley & Sons.
- [Randall 2004] Marcus Randall (2004). *Near parameter free ant colony optimisation*. In *International Workshop on Ant Colony Optimization and Swarm Intelligence*, pp. 374–381. Springer.
- [Rashedi et al. 2009] Esmat Rashedi, Hossein Nezamabadi-Pour, and Saeid Saryazdi (2009). *GSA: a gravitational search algorithm*. Information sciences, 179(13), 2232–2248.
- [Reinelt 1991] Gerhard Reinelt (1991). *TSPLIB – A traveling salesman problem library*. ORSA journal on computing, 3(4), 376–384.
- [Rice 1976] John R Rice (1976). *The algorithm selection problem*. Advances in computers, 15, 65–118.
- [Rodriguez et al. 2007] Jose Antonio Vazquez Rodriguez, Sanja Petrovic, and Abdellah Salhi (2007). *An investigation of hyper-heuristic search spaces*. In *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*, pp. 3776–3783. IEEE.
- [Roy et al. 2011] Gourab Ghosh Roy, Swagatam Das, Prithwish Chakraborty, and Pon-nuthurai N Suganthan (2011). *Design of non-uniform circular antenna arrays using a modified invasive weed optimization algorithm*. IEEE Transactions on antennas and propagation, 59(1), 110–118.
- [Rubinstein and Kroese 2016] Reuven Y Rubinstein and Dirk P Kroese (2016). *Simulation and the Monte Carlo method*, vol. 10: John Wiley & Sons.
- [Rummery and Niranjan 1994] Gavin A Rummery and Mahesan Niranjan (1994). *On-line Q-learning using connectionist systems*, vol. 37: University of Cambridge, Department of Engineering.

- [Russell et al. 1995] Stuart Jonathan Russell, Peter Norvig, John F Canny, Jitendra M Malik, and Douglas D Edwards (1995). *Artificial intelligence: a modern approach*, vol. 2: Prentice hall Englewood Cliffs, NJ.
- [Ryser-Welch and Miller 2014] Patricia Ryser-Welch and Julian F Miller (2014). *A Review of Hyper-Heuristic Frameworks*. In *Proceedings of the 50th Anniversary Convention of the AISB*.
- [Santos 1969] Eugene S Santos (1969). *Probabilistic Turing machines and computability*. *Proceedings of the American Mathematical Society*, 22(3), 704–710.
- [Sarewitz 2016] Daniel Sarewitz (2016). *The pressure to publish pushes down quality*. *Nature*, 533(7602), 147–147.
- [Schumacher et al. 2001] C Schumacher, Michael D Vose, and L Darrell Whitley (2001). *The no free lunch and problem description length*. In *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*, pp. 565–570. Morgan Kaufmann Publishers Inc.
- [Shah-Hosseini 2009] Hamed Shah-Hosseini (2009). *The intelligent water drops algorithm: a nature-inspired swarm-based optimization algorithm*. *International Journal of Bio-Inspired Computation*, 1(1-2), 71–79.
- [Singh et al. 1994] Satinder P Singh, Tommi S Jaakkola, Michael I Jordan, et al. (1994). *Learning Without State-Estimation in Partially Observable Markovian Decision Processes*. In *ICML*, pp. 284–292.
- [Skiena 1998] Steven S Skiena (1998). *The algorithm design manual: Text*, vol. 1: Springer Science & Business Media.
- [Sörensen 2015] Kenneth Sörensen (2015). *Metaheuristics, the metaphor exposed*. *International Transactions in Operational Research*, 22(1), 3–18.
- [Sörensen and Glover 2013] Kenneth Sörensen and Fred W Glover (2013). *Metaheuristics*. In *Encyclopedia of operations research and management science*, pp. 960–970: Springer.
- [Sorensen et al. 2017] Kenneth Sorensen, Marc Sevaux, and Fred Glover (2017). *A history of metaheuristics*. *arXiv preprint arXiv:1704.00853*.
- [Stipčević and Koç 2014] Mario Stipčević and Çetin Kaya Koç (2014). *True random number generators*. In *Open Problems in Mathematics and Computational Science*, pp. 275–315: Springer.

- [Stützle and Dorigo 1999] Thomas Stützle and Marco Dorigo (1999). *ACO algorithms for the traveling salesman problem*. Evolutionary Algorithms in Engineering and Computer Science, pp. 163–183.
- [Sutter 2005] Herb Sutter (2005). *The free lunch is over: A fundamental turn toward concurrency in software*. Dr. Dobbs's journal, 30(3), 202–210.
- [Sutton and Barto 1998] Richard S Sutton and Andrew G Barto (1998). *Reinforcement learning: An introduction*, vol. 1: MIT press Cambridge.
- [Swan et al. 2015] Jerry Swan, Steven Adriaensen, Mohamed Bishr, Edmund K Burke, John A Clark, Patrick De Causmaecker, Juanjo Durillo, Kevin Hammond, Emma Hart, Colin G Johnson, et al. (2015). *A Research Agenda for Metaheuristic Standardization*. In *Proceedings of the XI Metaheuristics International Conference*.
- [Swan et al. 2016] Jerry Swan, Patrick De Causmaecker, Simon Martin, and Ender Özcan (2016). *A re-characterization of hyper-heuristics*.
- [Swan et al. 2011] Jerry Swan, Ender Özcan, and Graham Kendall (2011). *Hyperion—a recursive hyper-heuristic framework*. In *Learning and intelligent optimization*, pp. 616–630: Springer.
- [Teller and Andre 1997] Astro Teller and David Andre (1997). *Automatically choosing the number of fitness cases: The rational allocation of trials*. Genetic Programming, 97, 321–328.
- [Trevisan 2010] Luca Trevisan (2010). *CS254: Computational Complexity - handout 2*. <https://www.cs.stanford.edu/~trevisan/cs254-10/lecture02.pdf>. Accessed: 2018-04-11.
- [Turing 1950] Alan M Turing (1950). *Computing machinery and intelligence*. Mind, 59(236), 433–460.
- [Turing 1937] Alan Mathison Turing (1937). *On computable numbers, with an application to the Entscheidungsproblem*. Proceedings of the London mathematical society, 2(1), 230–265.
- [Urbanowicz and Moore 2009] Ryan J Urbanowicz and Jason H Moore (2009). *Learning classifier systems: a complete introduction, review, and roadmap*. Journal of Artificial Evolution and Applications, 2009, 1.
- [Urra et al. 2013] Enrique Urra, Daniel Cabrera-Paniagua, and Claudio Cubillos (2013). *Towards an object-oriented pattern proposal for heuristic structures of diverse abstraction levels*. XXI Jornadas Chilenas de Computación 2013, PROCEEDINGS.

- [Van Onsem et al. 2014] Willem Van Onsem, Bart Demoen, and Patrick De Causmaecker (2014). *Hyper-criticism: A critical reflection on today's hyper-heuristics*. In *Proceedings of the 28th Annual Conference of the Operational Research Society*, vol. 28, pp. 159–161.
- [Vázquez-Rodríguez et al. 2009a] Vázquez-Rodríguez, Gabriela Ochoa, Tim Curtois, and Matthew Hyde (2009a). *A Travelling Salesman Problem Domain for the HyFlex Framework*. School of Computer Science, University of Nottingham, Tech. Rep.
- [Vázquez-Rodríguez et al. 2009b] José Antonio Vázquez-Rodríguez, Gabriela Ochoa, Tim Curtois, and Matthew Hyde (2009b). *A hyflex module for the permutation flow shop problem*. School of Computer Science, University of Nottingham, Tech. Rep.
- [Veach 1997] Eric Veach (1997). *Robust monte carlo methods for light transport simulation*. No. 1610: Stanford University PhD thesis.
- [Vinkers et al. 2015] Christiaan H Vinkers, Joeri K Tjink, and Willem M Otte (2015). *Use of positive and negative words in scientific PubMed abstracts between 1974 and 2014: retrospective analysis*. *Bmj*, 351, h6467.
- [Voudouris and Tsang 2003] Christos Voudouris and Edward Tsang (2003). *Guided local search*. *Handbook of metaheuristics*, pp. 185–218.
- [Wackerly et al. 2007] Dennis Wackerly, William Mendenhall, and Richard Scheaffer (2007). *Mathematical statistics with applications*: Nelson Education.
- [Walker et al. 2012] James D Walker, Gabriela Ochoa, Michel Gendreau, and Edmund K Burke (2012). *Vehicle routing and adaptive iterated local search within the hyflex hyper-heuristic framework*. In *Learning and Intelligent Optimization*, pp. 265–276: Springer.
- [Watkins and Dayan 1992] Christopher JCH Watkins and Peter Dayan (1992). *Q-learning*. *Machine learning*, 8(3-4), 279–292.
- [West et al. 2001] Douglas Brent West et al. (2001). *Introduction to graph theory*, vol. 2: Prentice hall Upper Saddle River.
- [Weyland 2010] Dennis Weyland (2010). *A Rigorous Analysis of the Harmony Search Algorithm: How the Research Community can be Misled by a Novel Methodology*. *International Journal of Applied Metaheuristic Computing*, 1(2), 50–60.
- [Williams 1992] Ronald J Williams (1992). *Simple statistical gradient-following algorithms for connectionist reinforcement learning*. In *Reinforcement Learning*, pp. 5–32: Springer.
- [Woeginger 2003] Gerhard Woeginger (2003). *Exact algorithms for NP-hard problems: A survey*. *Combinatorial Optimization - Eureka, You Shrink!*, pp. 185–207.

- [Wolpert and Macready 1997] David H Wolpert and William G Macready (1997). *No free lunch theorems for optimization*. IEEE transactions on evolutionary computation, 1(1), 67–82.
- [Xu et al. 2010] Lin Xu, Holger Hoos, and Kevin Leyton-Brown (2010). *Hydra: Automatically Configuring Algorithms for Portfolio-Based Selection..* In AAAI, vol. 10, pp. 210–216.
- [Xu et al. 2008] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown (2008). *SATzilla: portfolio-based algorithm selection for SAT*. Journal of artificial intelligence research, 32, 565–606.
- [Yang 2010] Xin-She Yang (2010). *A new metaheuristic bat-inspired algorithm*. Nature inspired cooperative strategies for optimization (NICSO 2010), pp. 65–74.
- [Yang 2012] Xin-She Yang (2012). *Flower pollination algorithm for global optimization..* In UCNC, pp. 240–249. Springer.
- [Yang 2014] Xin-She Yang (2014). *Nature-inspired optimization algorithms*: Elsevier.
- [Yang and Deb 2009] Xin-She Yang and Suash Deb (2009). *Cuckoo search via Lévy flights*. In *Nature & Biologically Inspired Computing, 2009. NaBIC 2009. World Congress on*, pp. 210–214. IEEE.
- [Yun and Epstein 2012] Xi Yun and Susan L Epstein (2012). *Learning algorithm portfolios for parallel execution*. In *Learning and intelligent optimization*, pp. 323–338: Springer.
- [Žák 1983] Stanislav Žák (1983). *A Turing machine time hierarchy*. Theoretical Computer Science, 26(3), 327–333.
- [Zhang et al. 2016] Tiantian Zhang, Michael Georgiopoulos, and Georgios C Anagnostopoulos (2016). *Multi-objective model selection via racing*. IEEE transactions on cybernetics, 46(8), 1863–1876.