Designing Reusable Metaheuristic Methods: A Semi-automated Approach

Steven Adriaensen, Tim Brys and Ann Nowé

Abstract— **Many interesting optimization problems cannot be solved efficiently. Recently, a lot of work has been done on metaheuristic optimization methods that quickly find approximate solutions to otherwise intractable problems. While successful, the field suffers from a notable lack of reuse of methods, both in practical applications as in research.**

In this paper, we describe a semi-automated approach to design more re-usable methods, based on key principles of re-usability such as simplicity, modularity and generality. We illustrate this methodology by designing general metaheuristics (using hyperheuristics) and show that the methods obtained are competitive with the contestants of the Cross-Domain Heuristic Search Competition (2011). In particular, we find a method performing better than the competition's winner, which can be considered the state-of-the-art in domain-independent metaheuristic search.

I. Introduction

Many interesting combinatorial optimization problems cannot be solved in polynomial time, i.e. are NP-hard. Classical examples of such problems are the Maximum Satisfiability (MAX-SAT), Vehicle Routing (VRP) and Traveling Salesman Problem (TSP). Already for relatively small instances, solving these problems can become intractable. As the problems we are interested in solving in practice are often even magnitudes larger, non-exact methods are often considered. One approach that recently received a lot of attention are so called metaheuristic optimization methods ([1], [2]). Metaheuristic optimization methods attempt to find good solutions to a problem by iteratively trying to improve a (set of) candidate solution(s). In practice, these methods often manage to quickly find approximate solutions to otherwise intractable problems. In recent years, a wide variety of metaheuristic methods and techniques were developed, both by researchers and practitioners. However, the field suffers from a notable lack of reuse of these methods.

First we offer some insight as to why this is the case. We do so by evaluating some properties affecting the re-usability of methods, well known in the (software) engineering community.

Simplicity: A lot of methods are used for no other reason than that they are simple. Complex methods are difficult to understand, implement, reproduce,... i.e. reuse.
State-of-the-art metaheuristic methods are often very complex, a complexity we argue not to be intrinsic, but rather accidental, i.e. simpler alternatives exist, but are just not considered. Heuristics are by nature vague and inexact notions, implementations on the other hand must be exact. This prompts us to make ad-hoc design decisions, adding accidental complexity. When first testing a method, typically various issues in performance are observed, which we subsequently patch. These patches often solve problems only partially, or cause new problems, incrementally adding further complexity.

Modularity: A modular method is a composition of components, each of which can be replaced and reused independently. This opens up the possibility of partial reuse and allows us to contain complexity.
In research, methods are often presented as an integral solution to some problem, i.e. a monolithic whole.

Generality: The generality of an algorithm is its ability to solve a wide range of problem instances. Generality is essential for practical applicability and thus re-usability of a method. In research, methods are typically designed to solve specific problems, i.e. are "made-to-measure" [3]. This approach is partly motivated by the "No Free Lunch Theorem" [4], stating that the average performance over all instances is the same for every method. Additionally, analysis usually focuses on peak performance on a small set of well known benchmark instances. Practical applications fail when this performance does not generalize to the problem instances we actually want to solve.

In this paper we describe a systematic approach to design more simple, modular, general, state-of-the-art methods. At its root lies the realization that, as a designer of metaheuristics, we are actually faced with a (meta-)optimization problem. Here, the search space consists of all possible ways to (approximately) solve some optimization problem (e.g. TSP), and the objective is to maximize some measure of performance (e.g. median length of the tour found after 10 minutes of optimization). The inexact notion, i.e. heuristic, the designer comes up with is no single method, but a potentially interesting subset of this search space. In making additional ad-hoc design decisions and patches he is in fact exploring this space manually through "trial & error". A slow and tedious process that, as described above, tends to lead to overly complex methods. In the approach described, we solve this meta-optimization problem semi-automatically. We still use expert knowledge and creativity to prune the search space, but the resulting reduced optimization problem is solved automatically. In formulating this meta-optimization problem, we ensure that the solution (method) obtained is simple, modular and general.

Steven Adriaensen, Tim Brys and Ann Nowé are with the Department of Computer Science, Vrije Universiteit Brussel, Elsene, Brussels, Belgium (email: steven.adriaensen@vub.ac.be).

Steven Adriaensen and Tim Brys are funded by a Ph.D grant of the Research Foundation Flanders (FWO).

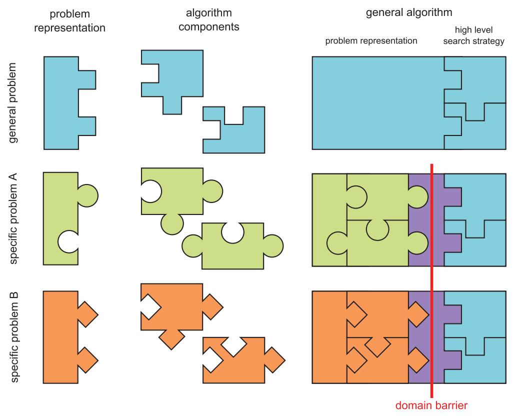

Fig. 1. A graphical illustration of how a high-level search strategy can be reused through separation of problem specific and problem independent components

```
function SOLVE($problem$, $t_{allowed}$)
    initialize()
    $c_{current}, c_{best} \leftarrow problem$.construction_heuristic()
    while $t_{allowed} - t_{elapsed} > 0$ do
        $option \leftarrow hyper\_heuristic$.select($options$)
        $c_{proposed} \leftarrow option$.apply($c_{current}$)
        $c_{best} \leftarrow problem$.best_of($c_{proposed}, c_{best}$)
        if $acceptance\_condition$.accepts($c_{proposed}$) then
            $c_{current} \leftarrow c_{proposed}$
        end if
        if $restart\_condition$.isMet() then
            initialize()
            $c_{current} \leftarrow problem$.construction_heuristic()
            $c_{best} \leftarrow problem$.best_of($c_{current}, c_{best}$)
        end if
    end while
    return $c_{best}$
end function
```

Fig. 2. The high-level search strategy considered in our illustration

We illustrate this approach by using it to design a general metaheuristic method. General metaheuristics strive to solve a wide range of problems. Rather than trying to outperform "made-to-measure" methods, these methods attempt to provide a cheap "off-the-peg" alternative. General metaheuristics can be obtained by explicitely separating problem specific from problem independent components, i.e. the high level search strategy. This high level search strategy accesses problem specific components through a problem independent interface (a.k.a. the "domain barrier" [5]) and can therefore be applied to any domain, given the domain-specific components are provided (see Figure 1). In order to evaluate how good high level search strategies are, we need problem dependent components and benchmark instances for multiple domains. For this reason, we used the HyFlex framework [6] in our implementation.

The remainder of the paper is organized as follows. Section II introduces the HyFlex framework. In Section III we formulate the meta-optimization problem and Section IV describes the algorithm we used to solve it. In Section V we solve our illustrative problem, and compare the resulting method to the state-of-the-art. Section VI discusses related research. Finally, in Section VII we conclude.

II. HyFlex Framework

HyFlex is a modular and flexible Java class library for developing and testing iterative general-purpose heuristic search algorithms. HyFlex currently provides 6 different problem domains: Maximum Satisfiability, Bin Packing, Permutation Flow Shop, Personnel Scheduling [7], Traveling Salesman and Vehicle Routing problem [8]. Each of which consists of:

- A set of 10-12 benchmark instances which can be solved.
- A set of low-level heuristics: One construction heuristic and multiple perturbative and recombination heuristics.[1]
- An evaluation function, measuring the cost of a candidate solution, to be minimized.

HyFlex has been used to support the first Cross-domain Heuristic Search Challenge (CHeSC 2011), during which 20 contestants were tested (31 10 min. runs) on 30 instances, 5 from each of the 6 domains implemented in HyFlex. The winner [9] was the algorithm obtaining the highest accumulated score across these instances. The competition inspired the performance measure (see Section III-B) used in our illustration, and in our analysis we show that the method we obtain is competitive with its contestants.

III. Meta-optimization Problem

In order to automatically solve the meta-optimization problem we are faced with, we must first specify it formally. In general an optimization problem can be formulated as follows:

"Find $x \in S$ such that $\forall y \in S : f(x) \leq f(y)$"

Where S is the set of candidate solutions, i.e. the search space, and f the objective function to be minimized.

A. Search Space

The search space of our meta-optimization problem is the set of possible designs for our metaheuristic. We specify this set as follows: When designing a metaheuristic, we are faced with various design decisions p_i ($1 \leq i \leq n$), each of which has a set of alternative choices D_i. A configuration is a combination of such choices (one for each design decision), and each configuration corresponds to a specific implementation. The set of such configurations we call a design space. The search space is then the subset of the design space we deem interesting (i.e. after pruning). Remark that design decisions can be conditional e.g. the choice for p_j is only relevant when $p_i = a$.

In specifying the search space in such manner, we explicitely modularize the design decisions to be made when implementing our heuristic. Accidental complexity can be avoided by including simple(r) alternatives for each design decision. Furthermore, prior art can be reused by including alternatives commonly used in literature.

[1]Each perturbative and recombination heuristic in addition takes two parameters: *intensity of mutation* (α) and *depth of search* β, ($0 \leq \alpha, \beta \leq 1$). In all our experiments we used $\alpha, \beta = 0.2$ (default value)

```
<root> Algorithm
Algorithm($hh,LocalSearch,AcceptAll,$restart)
Algorithm($hh,$os_not_ls,$accept,$restart)

<hh> HyperHeuristic
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})

<oe> EvaluationRule
NewBest
Improvement
Change

<os_not_ls> Options
PerturbativeHeuristic
IteratedLocalSearch

<accept> AcceptanceCondition
AcceptAll
AcceptNoWorse
AcceptTopList(20)
AcceptBestList(10)
AcceptLate(10000.0)
AcceptRandomWorse(0.1)
AcceptProbalisticWorse(0.5)
AcceptSA(2.0)

<restart> RestartCondition
RestartNever
RestartStuck
```

Fig. 3. Description of the search space considered in our illustration.

In our illustration, we consider simple, single point, high level search strategies as candidate solutions (see Figure 2). First, the construction heuristic is used to generate an initial candidate solution, then iteratively a **hyperheuristic** is used to select an **option**.[2] Next, the selected **option** is applied to generate a proposal candidate solution, and an **acceptance condition** is used to decide whether to accept it or not. Finally, we either perform a new iteration or **restart** the search process.

Here the main design decisions are the hyperheuristic, options, acceptance and restart condition used. In the Appendix, we briefly describe the alternatives considered for each. Figure 3 depicts the resulting design space, listing for each design decision its alternatives. Alternatives for a design decision p_i are either specified in a named section $<p_i>$ 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 a introduces further (conditional) design decisions, these are specified in round brackets after its name.

B. Objective Function

The objective function should quantify how good a method is. To do so, one typically evaluates its performance on a set of benchmark instances. While optimizing performance over a set of benchmark instances can be considered a multi-objective optimization problem, most often the objective function is scalarized, taking the average performance over all benchmark instances as the single objective. One of the main challenges is choosing a measure of performance (per instance) that summarizes in a meaningful way, i.e. without creating a bias towards a particular instance.

In our illustration we use the following objective function:

$$f(x) = \frac{1}{|P|} \sum_{\pi \in P} \mathbb{E}[s(R_{x,\pi}, \pi)]$$

Here P is the set of benchmark instances used in the CHeSC (2011) competition. $R_{x,\pi}$ is the result obtained by method x on an instance π, which is a random variable when x is stochastic. The scoring function s assigns a score to the result of method x on an instance π as follows: A method is assigned 10, 8, 6, 5, 4, 3, 2 or $10^{(8-r)}$ points, based on the rank of its result r among the median results of the contestants on π.[3] When multiple methods tie, they're assigned the average of the scores for the tied ranks.

In general the distribution of $R_{x,\pi}$, i.e. the solution quality distribution [2], is unknown and we can't compute f directly. In practice we will therefore use an evaluation function e that is a consistent unbiased estimator of f:

$$e(x) = \frac{1}{|\tilde{P}|} \sum_{\pi \in \tilde{P}} \frac{1}{n_\pi} \sum_{r \in \tilde{R}_{x,\pi}} s(r, \pi)$$

where $\tilde{R}_{x,\pi}$ is the set of results obtained by method x over n_π independent runs on the benchmark instance π and $\tilde{P} = \{\pi \in P : n_\pi > 0\}$.

IV. Meta-optimization Algorithm

To automatically solve the problem formulated in Section III, we need an algorithm. If we consider design decisions to be parameters, we obtain a parametrization problem. Various automatic parametrization methods have been described [10]–[15], and can thus be readily reused. Note that some adaptations to algorithm or problem might be required to handle conditional parameters.

Instead of advocating a particular algorithm, we'll list some properties we believe the algorithm should have. Typically, the algorithm will most of the time be testing (i.e. running) some candidate solution. Therefore, the key question is how to distribute these runs over the methods in the search space.

- More promising methods should receive more runs. This way, we avoid wasting too much time on poor methods and are able to reliably discriminate between good methods.
- Asymptotically $\forall x \in S : e(x) = f(x)$. Having this property guarantees that we, in the limit, will find the best method. Not having this property, we risk over-confidence [16] as $max_{x \in S}(e(x))$ is a biased estimator for $max_{x \in S}(f(x))$.

In our illustrative example, we used the FocusedILS metaheuristic described in [15]. FocusedILS is an Iterated Local Search (ILS) procedure, with random initialization and restart, operating on the 1-exchange neighborhood. The 1-exchange neighborhood of a configuration consists of all configurations that can be obtained by changing a single design choice. Initially, R configurations are drawn uniformly at random. From the best of these configurations, a local search procedure is started, and the local optimum obtained is used as the initial incumbent solution of the ILS procedure.

[2]Options here correspond to one or more applications of the domain-specific heuristics.

[3]Ordered according to increasing cost, as determined by the domain-specific evaluation function.

In ILS, iteratively a proposal is generated by performing a random walk of s steps, starting from the incumbent solution, followed by local search. If the generated proposal is better than the incumbent solution, it is used as incumbent solution in the subsequent iteration. At the end of each iteration, the algorithm is restarted with a probability of p.

FOCUSEDILS assigns runs to configurations in a particular fashion. Each time two configurations are compared (to see which one is best), an additional test will be performed using the least tested configuration (both in case of ties). Subsequently, further tests are performed until the highest evaluated configuration has at least as much runs as the configuration to which it is compared. Each time a better configuration is found, a number of bonus tests are performed, equal to the number of tests performed since we last found a better configuration. Benchmark instances are considered in a fixed order, i.e. the $(i+1)^{th}$ test of a configuration is performed on the $(i\%|P|+1)^{th}$ instance, where % is the modulo operator.

V. EXPERIMENTS

In this section we describe the experiments performed in this paper, and discuss their results. It is organized as follows. In Section V-A we solve the meta-optimization problem from our illustration. Section V-B compares the best method obtained to the contestants of the CHeSC (2011) competition. Finally, in Section V-C we verify its generality.

A. Meta-optimization Experiment

1) Setup: In this experiment we used FOCUSEDILS to solve the meta-optimization problem described in Section III.

To reduce the duration of our experiments, and to add diversification, we ran 30 processes of FOCUSEDILS in parallel. Each of these processes used the default parameter settings ($R = 10$, $p = 0.01$, $s = 3$), but considered the benchmark instances in a different, random order. We do so to compensate for FOCUSEDILS's initial bias towards methods performing well on the benchmarks that are re-evaluated first.

For fair comparison, we applied the benchmark program provided on the CHeSC (2011) website[4] before each run to determine $t_{allowed}$ such that $t_{allowed}$ time on our machine corresponds to 10 minutes on the machine used during the competition.

2) Results: The results in this section are those accumulated over all the meta-optimization processes. In total 52870 results where obtained, distributed over 2414 candidate solutions and 30 instances.

We first discuss how FOCUSEDILS distributed these runs. Ideally, we'd like runs 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 is expected under a uniform distribution.

[4]http://www.asap.cs.nott.ac.uk/external/chesc2011/benchmarking.html

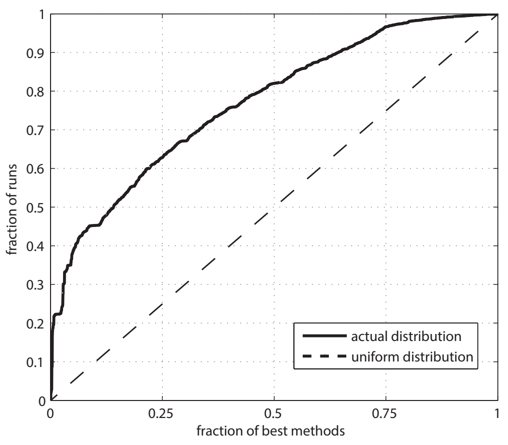

Fig. 4. Distribution of runs over the fraction best methods

This is due to the fact that FOCUSEDILS considers benchmarks in a fixed order, and most methods were tested less than $|P|$ times. As discussed in Section IV, we want more promising candidate solutions to receive more runs. Figure 4 shows the fraction of runs performed by the fraction of highest evaluated methods. Here we observe that the better methods 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 solution was tested at least once.

Next, we have a look at the best methods 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 methods. Table I shows the design choices made,[5] the number of runs performed, and the evaluation function value of the 15 top ranked methods.

Looking at the design choices, we find that 12 out of 15 of these methods use a QSELECT hyperheuristic with ROULETTEWHEEL selection. All 15 methods use the ITERATEDLOCALSEARCH options, in fact, the best methods

[5]See Appendix for the abbreviations used.

TABLE I
TOP 15 METHODS FOUND BY THE META-OPTIMIZATION PROCESSES

r	hh	os	ac	rc	runs	e
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(10k)	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

TABLE II

THE RESULTS OF THE CHESC 2011 COMPETITION WITH FS-ILS AS COMPETITOR

r	algorithm	s_{total}	s_{sat}	s_{bp}	s_{ps}	s_{fs}	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

using the PERTURBATIVEHEURISTIC or LOCALSEARCH alternatives are only evaluated at 3.83, 0.65 and ranked 65^{th}, 133^{th} respectively. 4 out of the 8 acceptance conditions considered appear in one of the top 15 methods, of which ACCEPTPROBABILISTICWORSE (9) and ACCEPTTOPLIST (4) are the most prevalent. Notably, the best method using ACCEPTALL is evaluated at 1.87 and ranks 124^{th}, motivating the use of acceptance conditions in general. While restart conditions are not required to obtain performant methods, we do note that the methods ranked 1^{st} and 2^{nd}, using RESTARTSTUCK, do outperform their variations, using RESTARTNEVER, ranked 3^{rd} and 4^{th} respectively.

The differences in evaluations among the 4 best methods are rather large and based on a large number of runs. Therefore, these differences are likely to be significant and unlikely to suffer from over-confidence. In our following experiments, we therefore focus on the best method found, which we'll from now on refer to as "Fair Share Iterated Local Search" (FS-ILS).[6]

B. FS-ILS as a CHeSC Contestant

1) Setup: In this experiment we compared the performance of FS-ILS to that of the contestants of the CHeSC (2011) competition. Here we computed, using the program available at the competition website, the outcome of the competition as if FS-ILS would have been a competitor. This program takes as input the median solution quality FS-ILS obtains on each instance. To avoid any artifacts caused by over-confidence, results were taken from 930 new runs, 31 on each instance.

2) Results: Table II shows the outcome of the CHeSC (2011) competition with FS-ILS as competitor. It shows the total score and scores on each domain for all contestants.

[6]Code can be found at https://github.com/Steven-Adriaensen/FS-ILS

TABLE III

COMPARISON OF FS-ILS TO ADAPHH

algorithm	$\hat{e}_{training}$	$\hat{e}_{test}$	$\hat{e}_{total}$
FS-ILS	9.08	8.16	8.43
AdapHH	9.17	8.02	8.36

We see that FS-ILS would have won the competition and performs best on 3 of the 6 domains. Furthermore, the worst score it obtains on any domain is higher than the worst score of any other contestant.

Note 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. Nonetheless, FS-ILS could have been a competitor and as such the comparison above does illustrate its competitiveness. Furthmore, it's worth mentioning that FS-ILS is far less complex than most other contestants, in particular ADAPHH [9], its main competitor.

C. Validation of FS-ILS

1) Setup: In previous experiments, we showed that FS-ILS performs extremely well on the benchmark instances used in the competition. However, in practice, we are interested in solving new instances, and the results from prior experiments give no guarantees that the observed performance will generalize. In our meta-optimization process, in anology with machine learning, we learn a metaheuristic that performs well on a given set of "training" instances. To avoid over-fitting, the metaheuristic obtained must be validated, i.e. we must evaluate its performance on a set of new "test" instances.

In our final experiment, we test FS-ILS on 28 new benchmark instances (30 runs each), 7 from all 4 domains available prior to the CHeSC (2011) competition,[7] but not used in the competition itself. These instances were chosen because results for 8 benchmark algorithms were made available for them prior to the competition.[8] The availability of these benchmark algorithms allows us to reuse the evaluation function described in Section III-B using these 8 benchmark algorithms ($\hat{e}$), instead of the 20 CHeSC contestants.

As these 8 benchmark algorithms can hardly be considered state-of-the-art, we compare the performance of our method to that of ADAPHH. For a fair comparison, we evaluate the performance of a publicly available version of ADAPHH (using the default parameter settings) on these 28 benchmark instances (30 runs), under the same conditions.

2) Results: Table III shows the evaluation ($\hat{e}$) of FS-ILS and ADAPHH on all 40 instances for which results of the 8 benchmark algorithms were made available ($\hat{e}_{total}$), 12 of which were CHeSC (2011) benchmark instances ($\hat{e}_{training}$) and 28 that weren't ($\hat{e}_{test}$).

[7]Maximum Satisfiability, Bin Packing, Permutation Flow Shop, Personnel Scheduling

[8]http://www.asap.cs.nott.ac.uk/external/chesc2011/defaulthh.html

In comparing $\hat{e}_{training}$ and $\hat{e}_{test}$, we find that FS-ILS performs better on the training instances, than the test instances. This is unlikely due to over-fitting, as we observe the same for ADAPHH. More likely, the 8 benchmark algorithms just happen to be slightly more competitive on the test instances. Overall FS-ILS performs slightly better than ADAPHH, and ironically ADAPHH performs better on the training instances (for which FS-ILS was optimized), while FS-ILS performs better on the test instances.

This experiment shows that the performance of FS-ILS generalizes well to new instances of the 4 original domains.

VI. Related Research

Various authors have noted the lack of reuse and made suggestions to improve the modularity (GLSM [2, Chapter 3]) and generality (algorithm selection [17], portfolios [18], hyperheuristics [3]) of metaheuristic methods.

While to date, most methods are designed using the traditional "trial & error" approach, attempts have been made to automate (part of) the design process. E.g. Genetic programming [19] was used to generate heuristics from scratch [3, Section 5], and Automatic parametrization ([12]–[15]) was used to determine the best values for the parameters of a given method.

The approach followed in this paper was largely inspired by, and built upon this research. In combining various of these ideas, we strive to obtain more reusable, metaheuristics.

VII. Conclusion

In this section we conclude and give a short summary of the ground covered in this paper, discussing limitations and further research.

In summary this paper makes two main contributions:

- We describe a semi-automated approach to design (re)usable metaheuristic methods. In this approach, we perform an explicit modularization and abstraction of the design decisions in a method, considering various alternatives for each design decision. Next, an automatic search is performed to find the combination of design choices that results in the most performant method. Unlike the traditional "trial & error" approach, this systematic approach allows us to control the accidental complexity, modularity and generality of the result, encouraging reuse of both methods and components.
- We provide a concrete illustration of the approach described. Here we consider the design of general metaheuristic methods, using hyperheuristics. We find FS-ILS. a simple Iterated Local Search (ILS) method, and show its performance to be competitive with the state-of-the-art. The experiments also identified various interesting design choices. Our results support the conjecture that ILS is a promising high-level search strategy, and the use of acceptance conditions in general. We also identify a promising new class of hyperheuristics, selecting options in a non-greedy, acceptance and time proportional fashion, and provide a cross-domain adaptation of the popular Metropolis acceptance condition.

In a sense this paper ends where most begin. Rather than describing a specific metaheuristic method, we describe and illustrate a way to obtain such methods. As a consequence, we didn't motivate or perform any in depth analysis of FS-ILS and its design choices, but plan to do so in the near future. Furthermore, designing methods is an iterative process, where the designer uses the insights obtained in each iteration, to refine the search space, leading to further improvements. E.g. Given the outcome of our experiments, a logical next step would be to consider the design of ILS methods. Finally, note that the approach described is applicable beyond the domain of metaheuristic optimization, and can be used to design software systems in general.

The experiments performed in this paper serve as a proof of concept. The problem formulation and meta-optimization method used therefore should not be considered best practice. The described approach has some limitations. It offers no guarantees that the performance of the method obtained will generalize to new problem instances. Arguably, optimizing performance on a small subset of problem instances, is not the best indication of the performance in general [4]. Remark that the same holds for methods obtained through the traditional "trial & error" approach. Interesting future research would be to consider different objectives that are better indicators of general performance.

The approach described is a time-consuming process. Both the implementation of alternatives and the meta-optimization process itself take a lot of time. Nonetheless, we'll argue that this cost is acceptable. First, we note that the traditional "trial & error" approach is time consuming as well. Furthermore, by reusing the results, alternatives, and methods obtained, the time required can be reduced and the time spent armortized. Finally, during the meta-optimization process, the researcher is free to perform other activities, for it is the computer that is engaged.

One might argue that our quest for simplicity results in methods that will not generalize very well. First, because most of the complexity in state-of-the-art methods stems from some form of adaptiveness to the problem. Second, because it is not difficult to construct a problem on which FS-ILS would fail. We'd argue that in both cases, problems exist on which they'll perform poorly. It is precisely due to a method's simplicity that it is obvious in which cases this occurs. Furthermore, this obviousness is desirable when reusing a method. For practitioners know what to be aware off when formulating their problem, and researchers know what to attempt to improve.

References

[1] G. A. Kochenberger *et al.*, *Handbook in Metaheuristics*. Springer, 2003.

[2] H. H. Hoos and T. Stützle, *Stochastic local search: Foundations & applications*. Morgan Kaufmann, 2004.

[3] E. K. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Ozcan, and R. Qu, "Hyper-heuristics: A survey of the state of the art," *Journal of the Operational Research Society*, vol. 64, no. 12, pp. 1695–1724, 2013.

[4] D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization," *Evolutionary Computation, IEEE Transactions on*, vol. 1, no. 1, pp. 67–82, 1997.

[5] E. Burke, G. Kendall, J. Newall, E. Hart, P. Ross, and S. Schulenburg, "Hyper-heuristics: An emerging direction in modern search technology," *International series in operations research and management science*, pp. 457–474, 2003.

[6] G. Ochoa, M. Hyde, T. Curtois, J. Vazquez-Rodriguez, J. Walker, M. Gendreau, G. Kendall, B. McCollum, A. Parkes, S. Petrovic *et al.*, "Hyflex: a benchmark framework for cross-domain heuristic search," *Evolutionary Computation in Combinatorial Optimization*, pp. 136–147, 2012.

[7] T. Curtois, G. Ochoa, M. Hyde, and J. A. Vázquez-Rodríguez, "A hyflex module for the personnel scheduling problem," *School of Computer Science, University of Nottingham, Tech. Rep*, 2009.

[8] J. D. Walker, G. Ochoa, M. Gendreau, and E. K. Burke, "Vehicle routing and adaptive iterated local search within the hyflex hyper-heuristic framework," in *Learning and Intelligent OptimizatioN*. Springer.

[9] M. Misir, K. Verbeeck, P. De Causmaecker, and G. Vanden Berghe, "An intelligent hyper-heuristic framework for chesc 2011," in *Learning and Intelligent OptimizatioN*, 2012.

[10] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Neural Networks, 1995. Proceedings., IEEE International Conference on*, vol. 4. IEEE, 1995, pp. 1942–1948.

[11] R. Rubinstein, "The cross-entropy method for combinatorial and continuous optimization," *Methodology and computing in applied probability*, vol. 1, no. 2, pp. 127–190, 1999.

[12] M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp, "A racing algorithm for configuring metaheuristics," in *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, New York, USA, 9-13 July 2002*. Morgan Kaufmann, 2002, pp. 11–18.

[13] M. Oltean, "Evolving evolutionary algorithms using linear genetic programming," *Evolutionary Computation*, vol. 13, no. 3, pp. 387–410, 2005.

[14] B. Adenso-Diaz and M. Laguna, "Fine-tuning of algorithms using fractional experimental designs and local search," *Operations Research*, vol. 54, no. 1, pp. 99–114, 2006.

[15] F. Hutter, H. H. Hoos, and T. Stutzle, "Automatic algorithm configuration based on local search," in *Proceedings of the National Conference on Artificial Intelligence*, vol. 22, no. 2. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2007, p. 1152.

[16] M. Birattari and M. Dorigo, "The problem of tuning metaheuristics as seen from a machine learning perspective," 2004.

[17] J. R. Rice, "The algorithm selection problem." *Advances in Computers*, vol. 15, pp. 65–118, 1976.

[18] C. P. Gomes and B. Selman, "Algorithm portfolios," *Artificial Intelligence*, vol. 126, no. 1, pp. 43–62, 2001.

[19] J. R. Koza, *Genetic Programming: On the Programming of Computers by the Means of Natural Selection*. Massachusetts: The MIT Press, 1992.

[20] E. K. Burke and Y. Bykov, "A late acceptance strategy in hill-climbing for exam timetabling problems," in *Proc. of the 7th International Conference on the Practice and Theory of Automated Timetabling, Montreal*, 2008.

APPENDIX

Here, we briefly describe the alternatives considered for each of the design decisions in our illustration. When choosing a certain alternative introduces further (conditional) design decisions, these are specified in round brackets after its name. The abbreviations used for alternatives in Table I, are given in square brackets.

Hyperheuristics:

UNIFORM [U]
: Selects an option uniformly at random.

QSELECT(SEL, EVAL) [QS]
: Uses a selection rule sel 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 $eval$ is used to re-evaluate o_i based on all its n_i previous applications.[9]

Selection rules:

ROULETTEWHEEL [RW]
: Selects an option o_i with probability $\frac{q(o_i)}{\sum_{o_j} q(o_j)}$.

EPSILONGREEDY(ϵ) [EG]
: Selects the best option with probability $1 - \epsilon$, and an option uniformly at random otherwise.

POLYRANK(k) [PR]
: Selects the r_i^{th} worst option with probability $\frac{r_i^k}{\sum_{r_j} r_j^k}$.

Evaluation rules:

CHANGE [C]
: $e(c_{current}) - e(c_{proposed})$

IMPROVEMENT [I]
: $max(0, e(c_{current}) - e(c_{proposed}))$

DURATION [D]
: The time in ms it took to generate $c_{proposed}$.

ACCEPTED [AC]
: 1 if $c_{proposed}$ was accepted, 0 otherwise.

NOOP [N]
: 1 if $c_{proposed} = c_{current}$, 0 otherwise.

NEWBEST [NB]
: 1 if $e(c_{proposed}) < e(c_{best})$, 0 otherwise.

TOTAL(oe) [T]
: The accumulation of evaluation rule oe over all evaluations.

AVERAGE(oe) [A]
: The moving average of evaluation rule oe over all evaluations.

EXPAVERAGE(oe,α) [EA]
: The exponential moving average of evaluation rule oe over all evaluations, using a weighting factor α.

WINDOWAVERAGE(oe,M) [WA]
: The average of evaluation rule oe in the last M evaluations.

SPEED [S]
: $\frac{n_i+1}{Total(Duration)(o_i)}$.

SPEEDACCEPTED [SA]
: $\frac{Total(Accepted)(o_i)+1}{Total(Duration)(o_i)}$.

SPEEDNEW [SN]
: $\frac{Total(Accepted)(o_i)-Total(Noop)(o_i)+1}{Total(Duration)(o_i)}$.

IMPROVEMENTOVERTIME(M) [IOT]
: $\frac{C*WindowAverage(Improvement,M)(o_i)+1}{Average(Duration)(o_i)}$.
Here C is some large constant.[10]

[9] The first 6 evaluation rules described are oblivious, i.e. they only consider the most recent application.

[10] 10^5 in our implementation

Options:

LOCALSEARCH [LS]

Each option corresponds to applying one of the *greedy* domain-specific perturbative heuristics.

PERTURBATIVEHEURISTIC [PH]

Each option corresponds to applying one of the domain-specific perturbative heuristics.

ITERATEDLOCALSEARCH [ILS]

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.

Acceptance Conditions:

ACCEPTALL [AA]

Accepts all proposals.

ACCEPTNOWORSE [ANW]

Accepts all non-worsening proposals.

ACCEPTTOPLIST(n) [ATL]

Accepts all proposals no worse than the n^{th} best solution observed so far.

ACCEPTBESTLIST(n) [ABL]

Accepts all proposals no worse than the n^{th} best, *new best* solution observed so far.

ACCEPTLATE(t) [AL]

Accepts all proposals no worse than the incumbent solution k iterations ago [20], 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.

ACCEPTRANDOMWORSE(ϵ) [ARW]

Accepts all non-worsening proposals and worsening proposals with a probability ϵ.

ACCEPTPROBABILISTICWORSE(T) [APW]

Accepts proposals with a probability $e^{\frac{e(c_{current})-e(c_{proposed})}{T*\mu_{impr}}}$, where e is the evaluation function and μ_{impr} the (moving) average improvement in improving iterations.

ACCEPTSA(T) [ASA]

Accepts proposals with a probability $e^{\frac{e(c_{current})-e(c_{proposed})}{T*\mu_{impr}}*\frac{t_{allowed}}{(t_{allowed}-t_{elapsed})}}$ where $t_{allowed}$ is the time we are allowed to optimize and $t_{elapsed}$ the time we have been optimizing.

Restart Conditions:

RESTARTNEVER [RN]

Never perform a restart.

RESTARTSTUCK [RS]

Perform a restart when $w > \frac{t_{allowed}}{t_{elapsed}} * w_{max}$, 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 re-initialized (initialize()). Exceptions are $t_{allowed}$, $t_{elapsed}$, c_{best} and w_{max}.

The search space is defined as a subset of the design space described above. Below we summarize how we pruned this search space.

- Multiple of the design decisions are continuous. For practical purposes we discretize the search space, considering only a finite set of alternatives for each design decision.
- Some of the evaluation rule alternatives have "the evaluation rule used" as design decision, leading to possible infinite recursion. To avoid this, we combine these alternatives only with the CHANGE, IMPROVEMENT and NEWBEST alternatives. Furthermore, other oblivious alternatives and TOTAL are not considered as alternatives individually.
- ROULETTEWHEEL selection is only combined with evaluation rules that evaluate to strictly positive values.
- LOCALSEARCH options are only combined with the ACCEPTALL acceptance condition, since all proposals are non-worsening.

Figure 3 describes the resulting search space unambiguously.