Fair-Share ILS: A Simple State-of-the-art Iterated Local Search Hyperheuristic.

Steven Adriaensen
Vrije Universiteit Brussel
steadria@vub.ac.be

Tim Brys
Vrije Universiteit Brussel
timbrys@vub.ac.be

Ann Nowé
Vrije Universiteit Brussel
ann.nowe@vub.ac.be

ABSTRACT

In this work we present a simple state-of-the-art selection hyperheuristic called Fair-Share Iterated Local Search (*FS-ILS*). *FS-ILS* is an iterated local search method using a conservative restart condition. Each iteration, a perturbation heuristic is selected proportionally to the acceptance rate of its previously proposed candidate solutions (after iterative improvement) by a domain-independent variant of the Metropolis condition. *FS-ILS* was developed in prior work using a semi-automated design approach. That work focused on how the method was found, rather than the method itself. As a result, it lacked a detailed explanation and analysis of the method, which will be the main contribution of this work. In our experiments we analyze *FS-ILS*'s parameter sensitivity, accidental complexity and compare it to the contestants of the CHeSC (2011) competition.

Categories and Subject Descriptors

G.1.6 [**Mathematics of Computing**]: Numerical Analysis—*Optimization*

General Terms

Algorithms, Experimentation, Performance

Keywords

Hyperheuristics, Combinatorial Optimization

1. INTRODUCTION

Many interesting combinatorial optimization problems cannot be solved in polynomial time, i.e. are NP-hard. Classical examples of such problems are MAX-SAT, the Vehicle Routing and Traveling Salesman problems. 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 required. One approach that recently received a lot of attention is metaheuristic optimization [10, 12]. 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.

Most research in metaheuristics is focused on problem specific techniques. This approach is motivated by the *No Free Lunch Theorem* [19], stating that on average, performance over all instances is the same for every method, therefore advocating a *made-to-measure* approach. As a consequence, metaheuristic methods are not readily applied to newly encountered problems, or even new instances of similar problems. Furthermore, the cost of developing and applying *made-to-measure* metaheuristic solutions is high, resulting in few practical applications.

Recently, there is a renewed interest in general metaheuristic search methods that attempt to solve a wide range of problems. Rather than attempting to outperform *made-to-measure* methods, these methods provide a cheap *off-the-peg* alternative. A popular approach in this renaissance are hyperheuristics [6]. A hyperheuristic combines a set of low level heuristics to solve a given problem instance. Here, low level heuristics are initialization, perturbative and recombination heuristics used in the problem instance domain. In known domains, the hyperheuristic approach can therefore leverage efficient domain-specific components used in *made-to-measure* methods. They can be applied to any domain, provided a set of low level heuristics is defined for it first. Note that the *No Free Lunch Theorem* is no argument against hyperheuristic methods as on every problem domain, the domain-specific aspects of the method (low level heuristics) differ, i.e. a different method is used.

Hyperheuristic methods can be divided in two classes. *Generation Hyperheuristics* generate (meta-)heuristics to solve a specific problem (domain); finding the combination of low-level heuristics precedes the solving of the problem (static). *Selection Hyperheuristics* iteratively select low-level heuristics to explore the search space while solving a problem (dynamic). As such, selection hyperheuristics can be viewed as domain-independent Variable Neighbourhood Search methods.

In this paper, we present Fair-Share Iterated Local Search (*FS-ILS*), a selection hyperheuristic. *FS-ILS* is an Iterated Local Search (ILS) method using a conservative restart condition. Each iteration, a perturbation heuristic is selected proportionally to the acceptance rate of the previously proposed candidate solutions (after iterative improvement) by a domain-independent variant of the Metropolis condition. One of the core design principles was simplicity and modularity, to improve reproducibility, and to encourage re-use in research and by practitioners. *FS-ILS* was obtained in [2] using a semi-automated design approach. Because [2] focused on "how" the method was found, rather than the method itself, it lacked a detailed explanation and analysis of the method, which will be the main contribution of this paper.

GECCO'14, July 12–16, 2014, Vancouver, BC, Canada.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2662-9/14/07 ...$15.00.
http://dx.doi.org/10.1145/2576768.2598285 .

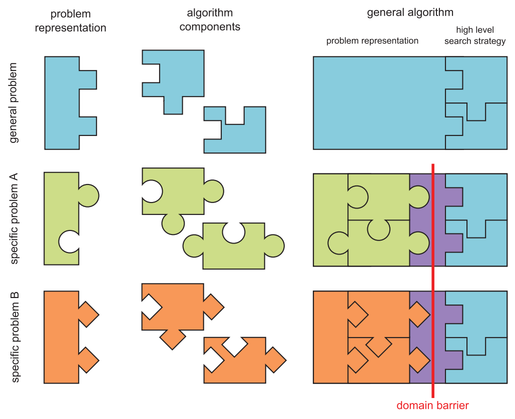

Figure 1: A graphical illustration of how a high-level search strategy can be applied to multiple problems through separation of problem specific and problem independent aspects

The remainder of the paper is organized as follows. Section 2.1 introduces the HyFlex framework. In Section 2.2 we describe *FS-ILS*, and motivate the design choices made. In Section 3 we perform experiments analysing *FS-ILS*'s parameter sensitivity and accidental complexity, and we compare it to the state-of-the-art. Section 4 discusses related research. Finally, in Section 5 we conclude.

2. METHODS

2.1 HyFlex

In the implementation and analysis of *FS-ILS*, we decided to use the HyFlex framework [15]. In this section, we motivate this decision and explain what the framework has to offer.

To evaluate general metaheuristic search strategies, we must test them on multiple instances of different problem domains. In the case of hyperheuristics, this means not only that we need benchmark problems, we also need low-level heuristics for all these domains. To avoid implementing all problem dependent aspects ourselves, we used the HyFlex framework. This framework offers a modular and flexible Java class library for developing and testing iterative general-purpose heuristic search algorithms. HyFlex currently provides 6 different problem domains: MAX-SAT, Bin Packing, Permutation Flow Shop, Personnel Scheduling, Traveling Salesman and Vehicle Routing problems. For each of these, it provides:

- 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.

One of HyFlex's core design principles is that all accesses to these domain-specific components must occur through a problem independent interface. Thanks to this explicit separation (i.e. the domain barrier, see Figure 1) any method using HyFlex can readily be applied to any instance of a problem domain implemented in HyFlex, without alterations.

[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)

Algorithm 1 Fair-Share Iterated Local Search

```
function SOLVE($problem$, $t_{allowed}$)
    SETUP()
    INIT()
    while time()$-t_{start} < t_{allowed}$ do
        $t_{before} \leftarrow$ time()
        $selected \leftarrow$ PERTURBATIONSELECTION($evaluations$)
        if $selected < |llhs_{pert}|$ then
            $c_{proposed} \leftarrow llhs_{pert}[selected]$.apply($c_{current}$)
        else
            $c_{proposed} \leftarrow problem$.generateSolution()
        end if
        $e_{proposed} \leftarrow problem$.e($c_{proposed}$)
        LOCALSEARCH($c_{proposed}$)
        $durations[selected] \overset{+}{\leftarrow}$ time() $-t_{before} + 1$
        if $c_{proposed} \neq c_{current} \wedge$ ACCEPT() then
            $news[selected]$++
            $c_{current} \leftarrow c_{proposed}$
            $e_{current} \leftarrow e_{proposed}$
        end if
        $evaluations[selected] = \frac{news[selected]}{durations[selected]}$
        if RESTART() then
            INIT()
        end if
    end while
    return $c_{best}$
end function
```

HyFlex has been used to support the first Cross-domain Heuristic Search Challenge (CHeSC 2011). This challenge is analogous to the athletics Decathlon event, where the goal is not to excel in one event at the expense of others, but to have a good general performance on each [15]. All 20 contestants were tested on 5 instances from each of the 6 domains. To test generalization to new instances, 2 of the 5 instances used in the competition weren't available before submission, i.e. were "hidden". To test generalization to "new" domains, the Traveling Salesman and Vehicle Routing domains were hidden as well. The winner [14] was the algorithm obtaining the highest accumulated score across these 30 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 8 algorithms score respectively 10, 8, 6, 5, 4, 3, 2 and 1 point.

In this work we not only compare *FS-ILS* to the CHeSC contestants, we also quantify the method's performance on a problem instance relative to the median performance of the contestants on the same instance.

2.2 Fair-Share Iterated Local Search

In this section we give a detailed explanation and motivation of the design decisions made in the *FS-ILS* method. The main purpose of this section is to simplify reproduction and share the train of thought that lead to *FS-ILS*. To this purpose, we also provide the complete and unambigious pseudo-code for *FS-ILS* (Algorithms 1-7). This pseudo-code was written to be language- and HyFlex-independent, but doesn't omit any implementation details. The code's entry point is the *Solve* function, which performs optimization (minimization) on a given *problem*, for a time $t_{allowed}$, after which the best candidate solution found c_{best} is returned. First, it **initializes** the candidate solution using the construction heuristic. Next it iteratively **selects** and applies a low-level, domain-specific **perturbation heuristic**, performs **local search** on the resulting candidate solution, decides whether or not to accept the obtained candidate as new incumbent solution using an **acceptance condition**, and **restarts** if necessary.

Before discussing the specifics of these different aspects of the algorithm, we briefly discuss some **naming conventions** for the variables used in the pseudo-code. Variables named c_{foo}, e_{foo} represent candidate solution foo and its respective evaluation function value. E.g. $c_{current}$, $c_{proposed}$ are the variables representing the incumbent and proposed candidate solution respectively, while $e_{current}$ and $e_{proposed}$ refer to their respective evaluation function values. Variables named t indicate some time, e.g. t_{start} is the time at which the optimization process started. Some variables have two variants, one for a specific run (re-initialized every restart) and one for the entire optimization process. To differentiate the former from the latter we add a super-script run to the former. E.g. e^{run}_{best} is the best evaluation function value observed this run (since last restart), while e_{best} is the best evaluation function value observed over the entire optimization process (including prior runs). MAX_FLOAT is the largest floating point number (without overflow). time() returns the current wall clock time. $random$.integer and $random$.real(a,b) draw an integer or real number uniformly at random from $[a, b)$. An array a is indexed $[0, |a| - 1]$, where $|a|$ denotes the length of the array. The expression $a[0 : n] \leftarrow b$ is used to represent the initialization of an array a of length n with all elements initialized to b. Finally the assignment operator $\leftarrow$ has copy semantics.

2.2.1 Initialization

Algorithm 2 Fair-Share Iterated Local Search: Setup

```
procedure SETUP
    $llhs_{ls} \leftarrow problem$.get_llhs($GREEDY$)
    $llhs_{pert} \leftarrow problem$.get_llhs($NON\text{-}GREEDY$)
    $wait_{max} \leftarrow 1$
    $t_{start} \leftarrow$ time()
    $t_{best} \leftarrow +\infty$
    $e_{best} \leftarrow MAX\_FLOAT$
end procedure
```

Algorithm 3 Fair-Share Iterated Local Search: Initialization

```
procedure INIT
    $news[0 : |llhs_{pert}| + 1] \leftarrow 1$
    $durations[0 : |llhs_{pert}| + 1] \leftarrow 0$
    $evaluations[0 : |llhs_{pert}| + 1] \leftarrow \frac{MAX\_FLOAT}{|llhs_{pert}|+1}$
    $n_{impr} \leftarrow 0$
    $wait \leftarrow 0$
    $t^{run}_{start} \leftarrow$ time()
    $c_{current} \leftarrow problem$.generateSolution()
    $e_{current} \leftarrow problem$.e($c_{current}$)
    $e^{run}_{best} \leftarrow e_{current}$
end procedure
```

Setup (Algorithm 2) and *Init* (Algorithm 3) are auxiliary procedures that together initialize all variables. *Setup* handles all variables that are initialized once at the beginning of the optimization process, while *Init* deals with the variables that are re-initialized after each restart.

2.2.2 Perturbation Heuristic Selection

In this section we explain and motivate the hyperheuristic selection procedure used to select a perturbation heuristic (option). The relevant logic can be found in Algorithm 1 and Algorithm 4.

Algorithm 4 Fair-Share Iterated Local Search: Perturbation Heuristic Selection

```
function PERTURBATIONSELECTION($evaluations$)
    $norm \leftarrow 0$
    for $i \leftarrow 0 : |evaluations|$ do
        $norm \overset{+}{\leftarrow} evaluations[i]$
    end for
    $pivot \leftarrow random$.real(0,norm)
    $selected \leftarrow 0$
    $ac \leftarrow evaluations[0]$
    while ac < pivot do
        $selected$++
    end while
    return selected
end function
```

As options, we consider all non-greedy perturbative heuristics in the domain ($llhs_{pert}$). In HyFlex, this corresponds to the heuristics in the *mutation* and *ruin-recreate* categories. In addition, we include the construction heuristic ($problem$.generateSolution()) as a last option. We do this because some HyFlex domains, in particular the Traveling Salesman domain, lack proper explorative heuristics. Either the perturbation induced is too small to escape the local optimum, or perturbation results in a solution much worse than the initial candidate solution.

In order to decide which option to use, we assign to each a value ($evaluations$) that reflects how well it performed in the past. Each iteration, we select an option proportional to this value (a.k.a. roulette wheel selection). Before its first application, an option is assumed to be of very high quality (orders of magnitudes larger than any real quality value) to make sure every option is tried at least once.

One of the key difficulties in hyperheuristics is how to evaluate explorative heuristics, as they might only lead to improvement many steps later. Therefore, we evaluate our options based on the candidate solution obtained after local search, i.e. we evaluate the perturbation heuristic combined with the iterative improvement procedure described in Section 2.2.3.

Perturbation heuristics are the only source of diversification in the algorithm. Without sufficiently diverse proposals ($c_{proposed}$) a method risks getting stuck in local optima. Therefore, we decide to evaluate and thus select options in a non-greedy fashion. It is up to the acceptance condition (see Section 2.2.4) to decide whether or not a proposal is good, i.e. to control diversification.

Following this mindset, we started developing our hyperheuristic selection procedure starting from plain uniform selection. We found that plain uniform selection discriminates fast options. E.g. consider two options a and b, which take 1 and 10 ms respectively to generate a new candidate solution. Under uniform selection, the algorithm will spend 10 times more time on option b than it does on a. This motivated us to select options proportionally to their speed, i.e. the rate at which they generate solutions $\frac{n[i]}{durations[i]}$ where $n[i]$ is the number of applications and $durations[i]$ is the total time spent using the i^{th} option. Using this procedure, in the limit, an equal amount of time will be spent on every option.

When an option's proposal does not lead to a new incumbent solution ($c_{current}$), either because $c_{current} = c_{proposal}$, or because the proposal is not accepted, the time spent generating it is effectively wasted. To encourage the selection of options that lead to advances in the search process, the selection procedure finally included in *FS-ILS* selects options proportional to the rate at which they generate new incumbent solutions $\frac{news[i]}{durations[i]}$ where $news[i]$ is the number of applications of the i^{th} option that resulted in a new incumbent solution.

2.2.3 Iterative Improvement Procedure

Algorithm 5 Fair-Share Iterated Local Search: Local Search

```
function LocalSearch(c)
    $llhs_{active} \leftarrow llhs_{ls}$
    $e_c \leftarrow problem.\text{e}(c)$
    while $\neg llhs_{active}$.isEmpty() do
        $index \leftarrow random.\text{integer}(0, |llhs_{active}|)$
        $c \leftarrow llhs_{active}.\text{apply}(c)$
        $e_{temp} \leftarrow problem.\text{e}(c)$
        if $e_{temp} < e_c$ then
            $e_c \leftarrow e_{temp}$
            $llhs_{active} \leftarrow llhs_{ls}$
        else
            $active.\text{remove}(index)$
        end if
    end while
    return c
end function
```

In our iterative improvement procedure (Algorithm 5), we consider all greedy perturbative heuristics in the domain ($llhs_{ls}$) as options. In HyFlex, this corresponds to the heuristics in the *local search* category. Applying these heuristics can never lead to a worse candidate solution.

Each iteration of the iterative improvement procedure we select an option uniform at random. When an application of an option does not lead to improvement, it is excluded (tabu) from the selection ($active$), until some other heuristic finds improvement.

If we assume that each application of an option leads to improvement, unless we are in a local optimum for that heuristic, our iterative improvement procedure returns a candidate solution that is a local optimum for each of the options. Even though this condition is not met for all heuristics in the *local search* categories, it leads to an elegant termination point.

2.2.4 Acceptance Condition

Algorithm 6 Fair-Share Iterated Local Search: Acceptance

```
function Accept
    $impr \leftarrow e_{current} - e_{proposed}$
    if $impr > 0$ then
        $n_{impr}$++
        $\mu_{impr} \overset{+}{\leftarrow} \frac{impr - \mu_{impr}}{n_{impr}}$
    end if
    return $random.\text{real}(0,1) < e^{\frac{impr}{T * \mu_{impr}}}$
end function
```

As mentioned in Section 2.2.2 the role of the acceptance condition is to control the diversification in the search process, i.e. whether or to accept a worsening proposal.[2] Never accepting worsening solutions typically results in the method getting stuck in a local optimum. While it is rare for an ILS scheme (given proper perturbation) to get stuck in a local optimum, we still risk losing a lot of time in a poor area of the search space.

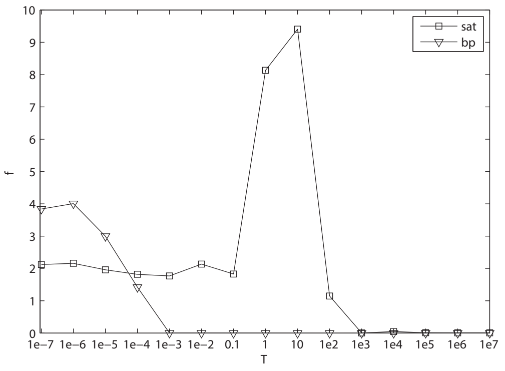

Figure 2: Performance of FS-ILS using the basic Metropolis acceptance condition with different T values.

The Metropolis acceptance condition accepts a worsening proposal with a probability $e^{\frac{e_{current} - e_{proposed}}{T}}$, where T is a positive parameter called the temperature. Under the Metropolis acceptance condition, the greater the worsening proposed, the smaller the likelihood the proposal will be accepted. Furthermore, the likelihood of accepting n times a worsening of a is equal to that of accepting a worsening of $n * a$ once. The higher the temperature, the higher the likelihood a worsening is accepted. More involved acceptance conditions vary this temperature over time (often referred to as simulated annealing [1]).

The main difficulty is choosing the temperature parameter T. Whether or not a worsening x is large is extremely domain (or even instance) dependent. E.g. in the MAX-SAT and Bin Packing domains, all evaluation function values lie in $[0, v]$ and $[0, 1]$ respectively (where v is the # variables in the formula). 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 worsenings to be accepted in Bin Packing, or (nearly) no worsenings at all in MAX-SAT. Figure 2 illustrates the problem, showing the performance (f, as defined in Section 3.1.1) of *FS-ILS* using the plain Metropolis condition on the MAX-SAT and Bin Packing domains for a wide range of temperature values.

In *FS-ILS*, we use a rather crude, yet effective way to make the choice of T less domain dependent by expressing worsening in terms of mean improvement, i.e. $e^{\frac{e_{current} - e_{proposed}}{T * \mu_{impr}}}$ (Algorithm 6). Here μ_{impr} is the (moving) mean improvement in improving iterations. Thus, we normalize the amount of worsening by dividing it by a metric that is equally domain-dependent. Whereas a worsening of x in MAX-SAT can hardly be compared to the same worsening in Bin Packing, $\frac{x}{\mu_{impr}}$ will be less domain dependent and more indicative of general quality. We do keep T as the only *FS-ILS* parameter, and will perform an extensive parameter sensitivity analysis in the experimental section.

[2]Most acceptance conditions accept all improving candidate solutions. Candidate solutions of equal quality are sometimes selectively refused (e.g. tabu-search) to effectively explore plateaus in the search space.

Algorithm 7 Fair-Share Iterated Local Search: Restart

```
function RESTART
    $t_{elapsed} \leftarrow$ time() $- t_{start}$
    $t^{run}_{elapsed} \leftarrow$ time() $- t^{run}_{start}$
    if $e_{current} < e^{run}_{best}$ then
        $e^{run}_{best} \leftarrow e_{current}$
        $wait_{max} \leftarrow$max($wait$,$wait_{max}$)
        $wait \leftarrow 0$
        if $e_{current} < e_{best}$ then
            $c_{best} \leftarrow c_{current}$
            $e_{best} \leftarrow e_{current}$
            $t_{best} \leftarrow t^{run}_{elapsed}$
        else if $e_{current} = e_{best}$ then
            $t_{best} \leftarrow$min($t^{run}_{elapsed}$,$t_{best}$)
        end if
    else
        $wait$++
    end if
    $patience \leftarrow \frac{t_{allowed}}{t_{elapsed}} * wait_{max}$
    return $wait > patience \wedge (t_{allowed} - t_{elapsed}) \geq t_{best}$
end function
```

2.2.5 Restart Condition

FS-ILS is restarted after it failed to find a new best candidate solution (i.e. improve e^{run}_{best}) for a certain amount of time (Algorithm 7). This method is inspired by the fact that a lot of methods 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 how to detect that the algorithm is stuck. We propose the following heuristic, which was inspired by a method used in CHeSC contestant *VNS-TW* [11] to detect a local optimum in the ILS process. *FS-ILS* is restarted if it hasn't found improvement for $a * wait_{max}$ iterations where $wait_{max}$ is the greatest number of iterations we had to wait for a new, best solution so far. The idea is that over time, $wait_{max}$ will approximate the longest time needed to still improve.

The choice of a is crucial. On the one hand to allow $wait_{max}$ to grow it is important that a is sufficiently large (at least > 1). On the other hand however, to prevent wasting too much time later on in the search (when $wait_{max}$ is somewhat accurate) a should be as small as possible. We therefore decided to lower a hyperbolically over time: $a = \frac{t_{allowed}}{t_{elapsed}}$, where $t_{allowed}$ is the time we are allowed to optimize and $t_{elapsed}$ the time we have already spent since the beginning of the optimization. Initially, a will be extremely 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 (t_{best}). This prevents restarts on instances on which a lot of time is required to obtain a high quality solution. E.g. given a $t_{allowed}$ of 10 minutes, when the best solution was obtained 9 minutes after initialization ($t^{run}_{elapsed}$), it is highly unlikely that a better solution is found after re-initialization.

Note that this restart condition is rather conservative and its core design concern was to prevent restarts as much as possible, and only restart when a method is obviously stuck and enough time is available to attain a solution of similar quality.

3. EXPERIMENTS

In this section, we experimentally analyze the performance of *FL-ILS*, concentrating first on its robustness to changes in its only numerical parameter. Then we analyze the accidental complexity of the method, comparing it with various simpler variants. Finally, we compare the method with the winner of the CheSC competition.

3.1 Parameter Sensitivity Analysis

3.1.1 Setup

The objective of the experiment described in this section is to analyze the choice of *FS-ILS*'s single (numerical) parameter T, i.e. the temperature used in its acceptance condition (see Section 2.2.4). To this purpose, we analyze *FS-ILS*'s performance using 15 different temperature values, ranging from 0.001 to 1000. In [2], T was intuitively chosen to be 0.5. To analyze the stability of this choice, 10 temperature values are chosen equidistantly in $[0.1, 1.0]$. For the sake of comparison we consider 3 further variants of *FS-ILS*

- A variant accepting all proposals (*aa*)
- A variant accepting no worsening proposals. (*anw*)
- A variant using the basic Metropolis acceptance condition with the temperature $t_i = 10^{i-7}$, $0 \leq i \leq 14$ that maximizes performance (*oracle*).

To simplify the comparison of different configurations of *FS-ILS* we want to quantify the performance of a certain configuration x. As in [2] we use the following evaluation function:

$$f(x) = \frac{1}{|P|} \sum_{\pi \in P} \frac{1}{n_{x,\pi}} \sum_{r \in R_{x,\pi}} s(r, \pi)$$

Here P is the set of benchmark instances on which x is tested. $R_{x,\pi}$ is the set of results obtained by method x over $n_{x,\pi}$ independent runs on the benchmark instance π. The scoring function s assigns points 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 obtained by a set of benchmark algorithms A on π.[3] In case of ties, s returns the average of the scores for the tied ranks.

In this experiment we choose the 30 instances used in CHeSC (2011) as benchmark instances P and the 20 contestants as benchmark algorithms A. 900 tests for each configuration were performed, 30 for each instance (n_π). For each run $t_{allowed}$ is chosen such that $t_{allowed}$ time on our machine (Intel Xeon E5320 1.86GHz) corresponds to 10 minutes on the machine used during the competition.[4]

3.1.2 Results

Figure 3 shows the performance of *FS-ILS* with different temperature values T on all 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

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

[4]Using the benchmark program provided on the CHeSC(2011) website

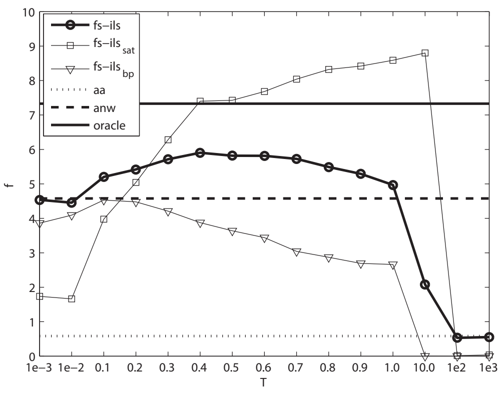

Figure 3: Performance of FS-ILS with different T values

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 $T = 0.5$ made in [2] seems like a good choice. Not only is it evaluated to be one of the best configurations (only 0.4 scored slightly better), this parameter choice is furthermore stable as *FS-ILS* performs similarly for similar temperature choices. Note that Figure 3 over-dramatizes the drop in performance for temperature choices beyond $[0.1, 1.0]$ because of the change in scale.

In comparing Figures 2 and 3 we find that our normalization of the Metropolis condition effectively solves the problem described in Section 2.2.2. Make no mistake, picking the best value for T is still about finding the best compromise. Some instances prefer more diversification and some less. E.g. On the MAX-SAT domain higher temperatures are preferred (10),while on the Bin Packing domain, lower temperatures are better (0.1). Furthermore, the *oracle* performs extremely well, illustrating that *FS-ILS* can be improved by making the choice of T more adaptive.

3.2 Accidental Complexity Analysis

3.2.1 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. To this purpose, we compare *FS-ILS* to simpler variants. We consider simpler alternatives for 4 design choices:

- The heuristic selection rule used (*SpeedNew*). Here we consider simpler variants selecting alternatives proportional to the rate at which they generate proposals (*Speed*) and variants using plain uniform selection (*Uniform*), see Section 2.2.2.
- The use of local search. Here we consider variants that do not perform local search, but use all perturbative heuristics (including the greedy) as options instead.
- The acceptance condition used (*APW*). Here we consider variants accepting only non-worsening proposals (*ANW*) and variants accepting all proposals (*AA*).
- The use of a restart condition. Here we consider variants that never restart.

Combining these simplifications gives rise to 35 simpler variants of *FS-ILS*. For each variant we perform 300 tests, 10 on each

r	select	ls	accept	restart	f	p
1	**SpeedNew**	**yes**	**APW**	**yes**	**5.77**	**1.0**
2	SpeedNew	yes	APW	no	5.54	0.0048
3	Speed	yes	APW	yes	5.2	4.6E-5
4	Speed	yes	APW	no	4.85	2.0E-8
5	SpeedNew	yes	ANW	no	4.58	1.6E-6
6	SpeedNew	yes	ANW	yes	4.57	8.3E-7
7	Speed	yes	ANW	no	4.16	1.0E-9
8	Speed	yes	ANW	yes	3.87	4.2E-12
9	SpeedNew	no	APW	yes	3.23	3.4E-15
10	SpeedNew	no	APW	no	3.09	0.0
11	Uniform	yes	APW	no	2.99	0.0
12	Uniform	yes	APW	yes	2.96	0.0
13	Uniform	yes	ANW	yes	2.64	0.0
14	Uniform	yes	ANW	no	2.52	0.0
15	Speed	no	APW	yes	2.46	0.0
16	Speed	no	APW	no	2.42	0.0
17	Uniform	no	APW	no	2.18	0.0
18	Uniform	no	APW	yes	2.15	0.0
19	Speed	no	ANW	yes	1.84	0.0
20	Speed	no	ANW	no	1.74	0.0
21	SpeedNew	no	ANW	no	1.53	0.0
22	SpeedNew	no	ANW	yes	1.41	0.0
23	Uniform	no	ANW	yes	1.36	0.0
24	Uniform	no	ANW	no	1.35	0.0
25	SpeedNew	yes	AA	no	0.67	0.0
26	Speed	yes	AA	no	0.6	0.0
27	SpeedNew	yes	AA	yes	0.48	0.0
28	Speed	yes	AA	yes	0.48	0.0
29	Uniform	yes	AA	yes	0.41	0.0
30	Uniform	yes	AA	no	0.4	0.0
31	Uniform	no	AA	yes	0.16	0.0
32	Uniform	no	AA	no	0.12	0.0
33	SpeedNew	no	AA	yes	0.02	0.0
34	Speed	no	AA	yes	0.01	0.0
35	SpeedNew	no	AA	no	0.0	0.0
36	Speed	no	AA	no	0.0	0.0

Table 1: Comparison of FS-ILS to simpler variants

CHeSC (2011) benchmark instance to evaluate its performance. We test the significance of the differences in performance compared to *FS-ILS* using the Wilcoxon Signed-Rank test.

3.2.2 Results

Table 1 shows for each of the 36 configurations the design choices made, their performance (f, as defined in Section 3.1.1) and the p-value for the Wilcoxon Signed-Rank test in a pair-wise comparison with *FS-ILS*.

We find that *FS-ILS* (the best, in bold) is evaluated significantly better than its simpler variants, i.e. every design choice has a significant contribution. We note that some simplifications have a greater impact on performance than others. This allows us to measure the contribution of a certain design choice to the performance of *FS-ILS*.

The choice for speed proportional **selection** (*Speed*, *SpeedNew*) adds great value as the best configuration using uniform selection is only ranked 11th and in 20 out of 24 cases configurations using a speed proportional selection scheme outperform their version using plain uniform selection. In the 4 remaining cases the configuration has no control of diversification (*AA*), and therefore performs

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

Table 2: The results of the CHeSC 2011 competition with FS-ILS as competitor

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 *SpeedNew* outperform the version using *Speed* in 10 out of 12 cases. In the cases it does not, no local search is performed and all worsening proposals are rejected. *SpeedNew* selection becomes too greedy and fails to explore.

The use of **local search** is clearly important as the best configuration without is only ranked 9^{th}. Furthermore, all variants using local search outperform their version not using local search.

Also, the chosen **acceptance condition** is important. The top 4 configurations use the (normalized) Metropolis acceptance condition. Furthermore, variants using this acceptance condition outperform their version accepting no worsening proposals, which in turn outperforms the version accepting all worsening proposals.

The choice for the **restart condition** is clearly the most controversial. On the one hand using the restart condition does (significantly) improve the performance of the 2 top configurations. Therefore, we decided to describe it in this paper. On the other hand, not using the restart condition is the simplification with the smallest impact and using the restart condition is in no way consistently beneficial. Therefore, one might consider omitting the restart condition, which would further simplify implementation and analysis, as there is no need for re-initialization or run-specific variables.

3.3 Comparison to State of the Art

3.3.1 Setup

The objective of the experiment described in this section is to compare the performance of *FS-ILS* to the state of the art in selection hyperheuristics. To a large extent, this has already been done in [2]. It was shown that *FS-ILS* would have won the CheSC (2011) competition if it were a contestant[5] and performs best on 3 of the 6 domains (see Table 2). To verify *FS-ILS*'s generalization to other problems, it was tested on 28 new instances (not used during its design/CHeSC). Its performance was compared to that of a publicly available implementation of *AdapHH* (the competition's winner) and was shown to be on par if not slightly better.

Currently, *FS-ILS* remains untested on 10 HyFlex instances,[6] which were excluded from [2] since no benchmark algorithms were available to evaluate its performance.[7] In this experiment we test *FS-ILS* and *AdapHH* 31 times on all 68 instances currently available in HyFlex. Rather than using some summary statistic, the comparison is done per instance, based on the median solution quality obtained. In addition we use a Mann-Whitney U test to determine whether the observed differences are significant (5% confidence).

[5]Please note that this comparison is not entirely fair. To design FS-ILS information was used that was not available to the other contestants, i.e. the benchmark instances used during the competition and the performance of the other contestants.

algorithm	**chesc**	**other**	**total**
FS-ILS	19 (11)	20 (13)	39 (24)
=	1	4	5
AdapHH	10 (8)	14 (8)	24 (16)

Table 3: Comparison of FS-ILS to AdapHH

3.3.2 Results

Table 3 shows on how many of the HyFlex instances *FS-ILS* performs better than (or tied with) *AdapHH* ($total$), 30 of which were used during the CHeSC (2011) competition ($chesc$) and 38 that weren't ($other$). The number of significant differences are shown in brackets.

We see that neither of the methods performs best on all instances. *FS-ILS* performs better than *AdapHH* on 39 out of the 68 instances and ties for 5 instances. Furthermore, not only does it perform better for the majority (19) of the 30 $chesc$ instances (on which *FS-ILS* was tested during design) it also performs better on the majority (20) of the 38 $other$ instances. We find 40 of the 63 observed differences to be significant. *FS-ILS* performs significantly better than *AdapHH* on more instances than vice versa, for both $chesc$ and $other$ instance sets. These results provide further evidence of *FS-ILS*'s competitiveness.

4. RELATED RESEARCH

The term hyperheuristic was first used to denote a (meta-)heuristic iteratively selecting and applying heuristics from a given set of low-level heuristics in [8]. Since then a lot of research has been performed on hyperheuristics. [6] provides an extensive discussion of the origin and recent developments within the domain.

While the potential of hyperheuristics in more general problem-independent search was recognized early on [4], due to practical difficulties most methods focused on a single domain. The creation of the HyFlex framework [15] in 2010 has greatly simplified cross-domain benchmarking and has been used in the implementation of many selection hyperheuristics ever since. Most notably, HyFlex was used to support the CHeSC 2011 competition, and its winner *AdapHH* [14] can be considered state-of-the-art. Contemporary work [18] reviews 16 of the 20 CHeSC contestants and draws conclusions supporting the choices made in *FS-ILS*'s design. A downside of the HyFlex framework is that it is rather restrictive. In particular, too little information is available about candidate solutions/heuristics to properly apply reinforcement learning [9]. Various extensions to Hyflex [16, 17] have been proposed and implemented.

[6]5 for both TSP and VRP domains

[7]For the 28 instances results for 8 benchmark algorithms were made available before the competition

Hyperheuristics follow a wide variety of designs. Traditionally they are iterative search strategies with a heuristic selection and solution acceptance stage ([3, 14]). However, other high-level search strategies have been developed recently. Iterated Local Search methods [3, 5, 7, 9] have been shown to outperform traditional search strategies in a similar setting on multiple occassions [3, 18]. Also evolutionary approaches [16], evolving a population of candidate solutions (vs. a single incumbent solution), and even (co-)evolving a population of hyperheuristics have been explored [13]. While evolutionary approaches present many interesting ideas, they tend to be complex and little evidence is provided to show they perform significantly better than simpler alternatives.

5. CONCLUSION

In this paper, we presented *FS-ILS*, a simple state-of-the-art Iterated Local Search selection hyperheuristic discovered using a semi-automated design approach [2]. We provide a detailed explanation and motivation of the design choices. In our experimental section, we first performed a parameter sensitivity analysis, showing that *FL-ILS* is largely robust to changes in its sole parameter. Subsequently we analyzed the presence of accidental complexity, showing that *FS-ILS* performs significantly better than simpler variants, motivating every design decision. Finally, we compared the performance of *FS-ILS* to that of the winner of the CHeSC competition on all HyFlex instances, providing further evidence of its competitiveness on this benchmark.

There are a vast number of metaheuristic methods. What is the value of yet another method? As a selection hyperheuristic, *FL-ILS* is an *off-the-peg* solution, and therefore readily applicable to new domains. Unlike other state-of-the-art methods, it is rather simple and therefore understandable and reproducible. E.g. using the information provided in this paper, *FS-ILS* can be reproduced in only a few hours, using about 200 lines of code.[8] In comparison, the publicly available implementation of *AdapHH* counts over 3000 lines of code.[9] These properties make *FS-ILS* an interesting starting point for future hyperheuristic research.

6. ACKNOWLEDGMENTS

The *FS-ILS* method was developed during Steven Adriaensen's master thesis, wherein Prof. Ann Nowé was promotor and Tim Brys supervisor. Currently, Steven Adriaensen and Tim Brys are funded by a Ph.D grant of the Research Foundation Flanders (FWO).

7. REFERENCES

[1] E. H. Aarts, J. H. Korst, and P. J. Van Laarhoven. Simulated annealing. *Local search in combinatorial optimization*, pages 91–120, 1997.

[2] S. Adriaensen, T. Brys, and A. Nowé. Designing reusable metaheuristic methods: A semi-automated approach. In *Evolutionary Computation (CEC), 2014 IEEE Congress on*. IEEE, 2014.

[3] E. Burke, T. Curtois, M. Hyde, G. Kendall, G. Ochoa, S. Petrovic, J. Vazquez-Rodriguez, and M. Gendreau. Iterated local search vs. hyper-heuristics: Towards general-purpose search algorithms. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–8. IEEE, 2010.

[4] 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*, pages 457–474, 2003.

[5] E. K. Burke, M. Gendreau, G. Ochoa, and J. D. Walker. Adaptive iterated local search for cross-domain optimisation. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 1987–1994. ACM, 2011.

[6] 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*, 64(12):1695–1724, 2013.

[7] C.-Y. Chan, F. Xue, W. Ip, and C. Cheung. A hyper-heuristic inspired by pearl hunting. In *Learning and Intelligent Optimization*, pages 349–353. Springer, 2012.

[8] P. Cowling, G. Kendall, and E. Soubeiga. A hyperheuristic approach to scheduling a sales summit. In *Practice and Theory of Automated Timetabling III*, pages 176–190. Springer, 2001.

[9] L. Di Gaspero and T. Urli. A reinforcement learning approach for the cross-domain heuristic search challenge. In *Proceedings of the 9th Metaheuristics International Conference (MIC 2011), Udine, Italy*, 2011.

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

[11] P.-C. Hsiao, T.-C. Chiang, and L.-C. Fu. A variable neighborhood search-based hyperheuristic for cross-domain optimization problems in chesc 2011 competition. In *Fifty-Third Conference of OR Society (OR53), Nottingham, UK*, 2011.

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

[13] D. Meignan. An evolutionary programming hyper-heuristic with coevolution for chesc'11. In *CHeSC 2011*.

[14] 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.

[15] 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*, pages 136–147, 2012.

[16] G. Ochoa, J. Walker, M. Hyde, and T. Curtois. Adaptive evolutionary algorithms and extensions to the hyflex hyper-heuristic framework. In *Parallel Problem Solving from Nature-PPSN XII*, pages 418–427. Springer, 2012.

[17] W. Van Onsem and B. Demoen. Parhyflex: A framework for parallel hyper-heuristics.

[18] W. Van Onsem, B. Demoen, and P. De Causmaecker. Hyper-criticism: A critical reflection on todays hyper-heuristics. In *Proceedings of the 28th Anual Conference of the Operational Research Society*, volume 28, 2014.

[19] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *Evolutionary Computation, IEEE Transactions on*, 1(1):67–82, 1997.

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

[9]To be fair, this does include some code non-essential to the optimization process. E.g. logging.