Automatic Design of Hybrid Stochastic Local Search Algorithms

Marie-Eléonore Marmion, Franco Mascia,
Manuel López-Ibáñez, and Thomas Stützle

IRIDIA, CoDE, Université Libre de Bruxelles (ULB), Brussels, Belgium
{mmarmion;fmascia;manuel.lopez-ibanez;stuetzle}@ulb.ac.be

Abstract. Many stochastic local search (SLS) methods rely on the manipulation of single solutions at each of the search steps. Examples are iterative improvement, iterated local search, simulated annealing, variable neighborhood search, and iterated greedy. These SLS methods are the basis of many state-of-the-art algorithms for hard combinatorial optimization problems. Often, several of these SLS methods are combined with each other to improve performance. We propose here a practical, unified structure that encompasses several such SLS methods. The proposed structure is *unified* because it integrates these metaheuristics into a single structure from which we can not only instantiate each of them, but we also can generate complex combinations and variants. Moreover, the structure is *practical* since we propose a method to instantiate actual algorithms for practical problems in a semi-automatic fashion. The method presented in this work implements a general local search structure as a grammar; an instantiation of such a grammar is a program that can be compiled into executable form. We propose to find the appropriate grammar instantiation for a particular problem by means of automatic configuration. The result is a semi-automatic system that, with little human effort, is able to generate powerful hybrid SLS algorithms.

Keywords: Stochastic local search, generalized local search structure, grammar, automatic algorithm design

1 Introduction

Many stochastic local search (SLS) methods manipulate a single solution at each of the search steps [11]. Examples of such SLS methods (also called metaheuristics) include classical iterative best- and first-improvement algorithms [20], iterated local search (ILS) [15], simulated annealing (SA) [13], variable neighborhood search (VNS) [10], random iterative improvement (RII) [20, 11], probabilistic iterative improvement (PII) [11], and iterated greedy (IG) [21], among others. Successful algorithms for hard combinatorial problems are often the result of an effective engineering of such SLS methods or of an appropriate combination of ideas from various of these methods. However, despite the plethora of possibilities, algorithm designers rarely consider but a few methods when tackling a new problem. We believe that this is due to two main reasons. First, the hybridization of such techniques is not a trivial task in terms of designing how the different parts should interact and implementing all possible interactions. Second, the effort required to design and analyze experiments that evaluate the different components and parameters of a hybrid algorithm is significant, thus considering a large number of hybrid algorithms may seem prohibitive at first.

In this paper, we propose a semi-automatic system that, with little human effort, is able to generate powerful hybrid SLS algorithms. We achieve this by combining two different proposals. First, we propose a unified structure that encompasses many of the above mentioned SLS methods proposed in the literature. We describe this unified structure as a grammar, from which we may instantiate not only SLS algorithms following one specific SLS methods, but also SLS algorithms that are composed of algorithmic components that are taken from various of these individual methods. Hence, our proposed grammar defines a very large space of possible hybrid SLS algorithms.

Our second proposal is to find the best instantiation of the grammar for a given problem by means of automatic configuration tools. Automatic configuration tools are typically used for tuning the parameters of optimization algorithms, given a set of training instances and a description of the parameter space [12, 14]. However, automatic configuration tools are also effective at instantiating heuristics from grammars [16].

In our approach, most human effort is devoted to implement and improve the problem-specific components (neighborhood moves, perturbations, heuristics), which are often the key to the success of an algorithm in a specific problem. Given these components and a set of training instances representative of the problem, the system takes care of generating a large number of hybrid SLS algorithms, and selects the best-performing on the training instances. This generation and selection process is mainly limited by the computation power available.

There are other two key characteristics of our proposal. First, the unified structure embodied by the grammar allows reusing the same few problem-specific components to generate a large number of different algorithms. The implementation of reusable components is based on ParadisEO [3], a framework that allows algorithm designers to reuse basic components to build their own algorithms. The system we propose in this work goes a step beyond and builds the algorithms automatically. The second key characteristic is that the algorithms instantiated from the grammar are stand-alone programs that are compiled to executable code. Therefore, the overhead introduced by the flexibility of the system is minimized, and the resulting automatically-crafted SLS algorithms are competitive with hand-crafted algorithms.

The paper is structured as follows. Section 2 introduces our unified structure for SLS algorithms and Section 3 how this structure can be implemented through a grammar and how SLS algorithms are configured. Section 4 describes our benchmark problem; experimental results are then given in Section 5 before we conclude and outline directions for extending our proposals in Section 6.

```
ILS Algorithm
1: $s_0$ := Initialization()
2: $s^*$ := ILS($s_0$)
3: return $s^*$
```

```
Function ILS($s_0$)
Require: perturbation, ls, acceptanceCriterion, stop
1: $s^*$ := ls($s_0$)
2: repeat
3:     $s'$ := perturbation($s^*$)
4:     $s''$ := ls($s'$)
5:     $s^*$ := acceptanceCriterion($s''$, $s^*$)
6: until termination criterion (stop) is satisfied
7: return $s^*$
```

Fig. 1. The iterated local search (ILS) algorithm

2 Generalized Local Search Structure

In this paper, we focus on SLS algorithms that work on a single solution at a time. The algorithm may internally keep a memory of multiple solutions, such as the best solution found so far, but there is the concept of the *current solution*, whose neighborhood is being explored.

We propose a generalized local search (GLS) structure modeled after iterated local search (ILS) [15]. ILS, as shown in Fig. 1, starts from an initial solution s_0, applies an improvement method (usually referred as local search, ls), and then three steps are repeated until the termination criterion is met: the current solution is perturbed to generate a new one, local search is applied to the new solution, and the acceptance criterion (in the simplest case of acceptance criteria) accepts the new solution or stays with the current one. ILS contains the most important elements of any hybrid LS algorithm, which are a *perturbation* operator, a subsidiary *local search*, and an *acceptance criterion*.

The *perturbation* is a transformation of the input solution. In ILS, this is typically a small random transformation of the solution but it may also be a random re-initialization. A perturbation may be one simple move in a neighborhood space, but it may also be composed of k applications of a simple move, and k may be even vary during the search either based on feedback of the search process or according to a pre-defined schedule. The *local search* can range from a simple iterative improvement over short runs of an SA algorithm to a full-fledged ILS. It could also be that no local search algorithm is used at all.

The *acceptance criterion* determines which solution will replace the current solution. The most basic acceptance criterion (*improveAccept*) accepts only solutions that are better (strictly or not) than the best solution found so far. Other acceptance criteria allow worse solutions to be accepted in order to increase the exploration of the search space. For instance, the *probAccept* criterion accepts a worsening solution with a probability $p \in [0, 1]$. For example, if $p = 1$, every new solution is accepted (*alwaysAccept*). A *thresholdAccept* criterion accepts a worsening solution if the relative deviation between the best and the current solution is below a threshold. In simulated annealing, a worse solution is accepted according to the Metropolis criterion (*metropolisAccept*). This criterion uses a cooling schedule that starts from an initial temperature, the temperature

Table 1. Classical SLS algorithms modeled after the ILS scheme.

Name	Perturbation	Local Search	Acceptance Criterion
SA [13]	one move	∅	Metropolis
PII [11]	one move	∅	prob.
RII [20, 11]	one move	∅	probRandom
VNS [10]	variable move	first-improv. descent	improvingStrictly
IG [21]	deconstruction-construction	"any"	"any"

is decreased according to the cooling schedule until the algorithm stops after reaching a final temperature. Often, the temperature is decreased periodically after a number of iterations (*span*).

By considering different alternatives for each of these components, we can replicate many of the SLS methods proposed in the literature. For instance, simulated annealing (SA) can be replicated by defining the perturbation operator as a simple move operator in a neighborhood, using no subsidiary local search, and using the Metropolis acceptance criterion. With these components, the scheme given for ILS above will actually replicate a classical SA algorithm.

Another example is variable neighborhood search (VNS) [10]. VNS executes an iterative improvement method at each iteration, and varies the strength of the perturbation depending on whether the resulting solution improves the best so far. This is equivalent to an ILS with a specialized *variable move* operator, iterative improvement as local search, and an *improveAccept* acceptance criterion. Other classical SLS algorithms can be modeled after ILS in a similar manner, as shown in Table 1.

In addition to replicating these classical SLS algorithms, the GLS structure proposed here can also reproduce more complex combinations of SLS algorithms. For example, an ILS algorithm can use a different ILS algorithm (with different perturbation and/or acceptance criterion) as a subsidiary local search, which, in turn may use SA as its own subsidiary local search. We call *recursion* the possibility of an ILS to embed another ILS []. The level of recursion is the number of embedded ILSs. This ability of combining simple components to generate hybrid local searches allows designing powerful algorithms. However, it raises the question of how to find high-performing algorithms for a particular problem, among all the possible combinations. The next section deals with this question.

3 Implementation

3.1 A practical implementation of the GLS structure

In this section, we describe how to implement the GLS structure proposed above in order to generate practical algorithms for a given problem. Our method consists of three parts. First, we use a generative grammar to describe the design space defined by the GLS structure. Second, we use a re-usable framework of source code components as the underlying implementation of the grammar. This implementation includes both problem-independent code, which can be re-used in any problem, and problem-specific components, which must be developed for each problem. Finally, we use automatic algorithm configuration tools to search the design space and generate high-performing instantiations of the grammar, given a set of training instances representative of a problem.

3.2 A grammar description of the GLS structure

A practical implementation of the GLS structure will contain many components that interact. Implementing such a GLS structure as a unique monolithic algorithm is a complex task. Moreover, the fact that a local search can be embedded within another in arbitrary ways complicates such implementation. The alternative that we propose here is to implement only the individual components, with clearly defined interfaces, and directly generate specific algorithms by combining these components. This is a typical problem in genetic programming, where grammars are often used to represent the design space of an algorithm [17].

A grammar is a set of derivation rules that describes how the symbols in a language can be combined to produce valid sentences. In our case, the valid sentences are local search algorithms encoded in `C++`, but for clarity we will describe the algorithms in pseudo-code. Fig. 2 shows the grammar that describes the GLS structure proposed in the previous section. Each line is a production rule of the form `<non-terminal> ::=` *expression* that describes how the non-terminal on the left-hand side can be replaced by the expression on the right-hand side. Expressions may contain terminal and/or non-terminals. Alternative expressions are separated with the symbol "`|`". The non-terminal symbol `<algorithm>` defines the starting point for instantiating an algorithm from the grammar.

The first three rules in the grammar describe the main structure of the GLS structure proposed earlier (see Fig. 1). The next three rules describe the basic components of our GLS structure, that is, the perturbation operator (`<perturb>`), local search (`<ls>`), and acceptance criterion (`<accept>`). Since the rule `<ls>` can expand to `<ils>` which contains again `<ls>`, a local search can be embedded within another local search (recursion). The other rules describe the alternatives available for the various components. Our grammar explicitly contains classical local search algorithms, but defined in terms of ILS, as detailed in Table 1. Moreover, the grammar also allows problem-specific components (`<pbs_...>`), which can be implemented for each problem tackled.

The possibility of adding problem-specific components is an advantage of our proposed method. Such components are critical for the success of SLS algorithms. For example, in this way problem specific construction and destruction mechanisms can be incorporated and be used in the destruction/construction phase (`<deconst-construct_perturb>`) of an IG algorithm. Hence, our grammar must account for such components. A practical implementation of our method also requires to define other problem-specific components in order to describe the representation of the problem, neighborhood operators, the objective function and how to read an instance of the problem. For simplicity, we do not include these in our exposition, but they are implemented in a similar fashion.

```
     <algorithm> ::= <initialization> <ils>
<initialization> ::= random | <pbs_initialization>
           <ils> ::= ILS(<perturb>, <ls>, <accept>, <stop>)

<perturb> ::= none | <initialization> | <pbs_perturb>
     <ls> ::= <ils> | <descent> | <sa> | <rii> | <pii> | <vns> | <ig> | <pbs_ls>
 <accept> ::= alwaysAccept | improvingAccept  <comparator>
            | prob(<value_prob_accept>) | probRandom | <metropolis>
            | threshold(<value_threshold_accept>) | <pbs_accept>

<descent> ::= bestDescent(<comparator>, <stop>)
            | firstImprDescent(<comparator>, <stop>)
    <sa>  ::= ILS(<pbs_move>, no_ls,  <metropolis>, <stop>)
    <rii> ::= ILS(<pbs_move>, no_ls, probRandom, <stop>)
    <pii> ::= ILS(<pbs_move>, no_ls, prob(<value_prob_accept>), <stop>)
    <vns> ::= ILS(<pbs_variable_move>, firstImprDescent(improvingStrictly),
                  improvingAccept(improvingStrictly), <stop>)
     <ig> ::= ILS(<deconst-construct_perturb>, <ls>, <accept>, <stop>)

<comparator> ::= improvingStrictly | improving
<value_prob_accept> ::= [0, 1]
<value_threshold_accept> ::= [0, 1]
<metropolis> ::= metropolisAccept(<init_temperature>, <final_temperature>,
                                  <decreasing_temperature_ratio>, <span>)
 <init_temperature> ::= {1, 2,..., 10000}
<final_temperature> ::= {1, 2,..., 100}
<decreasing_temperature_ratio> ::= [0, 1]
<span> ::= {1, 2,..., 10000}
```

Fig. 2. A simplified view of the grammar for the GLS structure.

Finally, each ILS in the proposed grammar has its own termination criterion (`<stop>`), which is typically a maximum computation time. If there is more than one level of ILS algorithms, the total computation time must be divided among them, such that the inner level does not consume all available time. We adopt here a simple scheme. The top-level ILS stops once the total time is consumed. Each subsequent level stops after consuming a ratio of the time allocated to its parent ILS. This ratio is controlled by a parameter $t_{\text{ls}} \in \{0.1, 0.2, \ldots, 1\}$ for each level of ILS. These ratios have to be tuned in order to generate an efficient hybrid SLS algorithm.

In practice, an instantiation of the grammar produces an algorithm that is mapped to source code implementing the individual components. In our case, the implementation of the components is done using Paradiseo [3], an open-source `C++` framework whose purpose is to facilitate the design of metaheuristics by providing a library of reusable components. The idea is that an algorithm designer can re-use the available algorithm components or implement her own components, and freely combine these components to design new algorithms. Our proposal goes a step beyond this idea, since in our proposed method the algorithm designer can focus on implementing problem-specific components, while the grammar takes care of describing the possible algorithm designs given the available components. The next section describes how to automatically find a high-performing SLS algorithm for a given problem, among all the possible algorithm designs that can be generated from the grammar.

3.3 Automatic generation of hybrid LS metaheuristics

Given a particular problem, our goal is to find the highest-performing instantiation of the grammar given above. As mentioned above, techniques such as genetic programming [17] and grammatical evolution [2] are often used for this task. Recently, we have shown how to instantiate IG algorithms from a grammar by means of a parametric representation [16]. The use of a parametric representation has certain advantages and enables the use of state-of-the-art automatic configuration tools for offline parameter tuning. In that work, we show that a parametric representation produced better IG algorithms than the representation used by grammatical evolution. Here we explore the much larger space of SLS algorithms defined by the proposed GLS structure.

We follow the method described in our previous work [16] to generate a parametric description of the grammar. This requires defining a maximum limit to the number of ILS levels in the final algorithm, that is, a maximum number of applications of the rule `<ls>` in the grammar. This limit has an influence on the number of parameters required to describe the grammar. In the next section, we explore the effect of this limit on the results.

From the parameter description and given a set of training instances representative of the problem, we apply an automatic configuration tool to search the space of possible algorithm designs. Here, we use irace [14], a publicly available implementation of Iterated F-Race [1]. Nonetheless, any automatic configuration tool that handles large numbers of categorical, numerical and conditional parameters with complex constraints would be appropriate.

Each parameter configuration tested by irace is an instantiation of the grammar, which is mapped to `C++` code and compiled into an executable. This executable is then run on various training instances by irace in order to determine its performance. A complete description of the irace procedure is beyond the scope of the paper. It suffices to say that the irace procedure stops after exhausting a given budget of algorithm runs, and that it returns the SLS algorithm configuration that it identified as the best performing one during the tuning.

4 Experimental Setup

We test our proposed method on the permutation flowshop scheduling problem with weighted tardiness (*PFSP-WT*). In contrast to our previous work [16], our aim here is to automatically generate a hybrid SLS algorithm that matches or outperforms the current state-of-the-art algorithm for the *PFSP-WT*. First, we briefly describe the *PFSP-WT*. Then, we summarize the state-of-the-art algorithm and the problem-specific components added to our grammar. Finally, we describe the experimental setup.

4.1 The *PFSP-WT*

The permutation flowshop scheduling problem (PFSP) encompasses a variety of problems that are typical of industrial production environments. The common goal of various PFSPs is to schedule n jobs on m machines with the condition that all jobs must be processed in the same order and jobs are not allowed to pass each other. Each job i requires, on each machine j, a fixed, non-negative processing time p_{ij}.

In the *PFSP-WT*, we are asked to determine a schedule that minimizes the total weighted tardiness. Each job i has a due date d_i, which denotes the desired completion time of the job on the last machine, and a priority weight w_i, which denotes its importance. The *tardiness* of a job i is defined as $T_i = \max\{C_i - d_i, 0\}$, where C_i is the completion time of job i on the last machine, and the *total weighted tardiness* is given by $\sum_{i=1}^{n} w_i \cdot T_i$. This problem is *NP*-hard even for a single machine [5].

4.2 Local search components for the *PFSP-WT*

To the best of our knowledge, a state-of-the-art algorithm for the *PFSP-WT* was proposed by Dubois-Lacoste et al. [7]. This algorithm, henceforth called soa-IG, is an iterated greedy algorithm that works as follows. An initial solution is constructed using a modified version of the well-known NEH algorithm [19] called NEH-WSLACK. In NEH-WSLACK [6], the WSLACK heuristic provides the initial order for the NEH algorithm, and the jobs are inserted in the solution in the order that minimizes the partial objective function, i.e., computed using the jobs present in the partially constructed solution. The local search in soa-IG is a first-improvement descent using a swap neighborhood and with a maximum number of swaps, fixed to $2 \cdot (n - 1)$ swaps, where n is the number of jobs. The perturbation operator consists in removing d jobs randomly from the solution. These jobs are re-inserted one by one to minimize the partial objective function. Finally, in the acceptance criterion, a new solution that is worse than the current one is accepted with a probability given by $\exp(100 \cdot (f(\pi) - f(\pi'))/(f(\pi) \cdot T_c))$, where T_c is a user-defined parameter, $f(\pi)$ is the objective value of the current solution and $f(\pi')$ is the objective value of the new one. Dubois-Lacoste et al. [7] suggest the settings $d = 5$ and $T_c = 1.2$.

We add the aforementioned components to the grammar of our GLS structure as additional problem-specific components (Fig. 3). In particular, we add two initialization methods, NEH with and without the WSLACK heuristic (`NEH` and `NEH-WSLACK`). In addition to the random destruction-construction perturbation used by soa-IG, we add further problem-specific perturbations based on classical neighborhood move operators (insert, exchange and swap) and a strength parameter k that controls the number of random moves applied per perturbation. The value of k may be fixed or vary during the run (`var_`) as in VNS. The problem-specific local search used by SOA is added to the grammar (`soa_ls`). Moreover, the `pbs(_variable)_move` used in the grammar (see Fig 2) are set to the insert move. Note that, the descents also use the insert move to define the neighborhood. Finally, we add the acceptance criterion of soa-IG as an additional acceptance criterion.

```
<pbs_initialization> ::= NEH | NEH-WSLACK
<pbs_perturb> ::= <deconst-construct_perturb>
                | <perturb_move>(<k>)
                | var_<perturb_move>(<k>)
<perturb_move> ::= insert | swap | exchange
<k> ::=  {1,2,...,10}
<deconst-construct_perturb> ::= soa_ig_perturb(<d>)
<d> ::= {1,2,...,10}
<pbs_ls> ::= soa_ig_ls

<pbs_variable_move> ::= var_<pbs_move>(<k>)
<pbs_move> ::= insert
<pbs_accept> ::= soa_ig_accept(<Tc>)
<Tc> ::= [0,1]
```

Fig. 3. The extended grammar for the *PFSP-WT*.

4.3 Experimental Protocol

We assess the potential of the proposed method by generating three hybrid SLS algorithms for the *PFSP-WT*, and comparing them with soa-IG. In particular, we generate three algorithms (ALS1, ALS2, and ALS3) for tackling the *PFSP-WT* from our GLS structure, by allowing different levels of recursion.

The procedure for generating these three algorithms is as follows. We consider the grammar presented in Fig. 2 and the *PFSP-WT*-specific extensions discussed above (Fig. 3). For each level of recursion, we automatically generate a parameter description. Indeed, the recursion leads to an increasing number of parameters. With one level of recursion, i.e., a single ILS, the grammar is represented by 80 parameters. Of these 80 parameters, 27 are categorical and represent possible algorithmic choices, 25 are integer-valued, and 28 are real-valued. With two or three levels of recursion, the number of parameters increases to 127 and 174, respectively. Any combination of the values that can be assumed by these parameters defines a different hybrid SLS algorithm implemented in C++ and compiled with GCC 4.7.2 with options "`-Ofast -flto -march=native`". All experiments are carried out on a single core of AMD Opteron 6272 processors (2.1GHz) running CentOS 6.2 Linux.

The parameter description is given to irace together with a number of training instances. As training instances, we generated 10 random *PFSP-WT* instances for each number of jobs in $\{50, 60, 70, 80, 90, 100\}$ and with 20 machines, following the procedure described by Minella et al. [18]. Within irace, a specific algorithm, i.e., a specific parameter configuration, is evaluated by running it on a training instance with a time limit of 30 CPU-seconds. A single run of irace stops after exhausting a given budget of evaluations. Since the number of parameters is different according to the level of recursion, we used different budgets for the different runs of irace; concretely, 30 000 evaluations for generating ALS1, 40 000 for generating ALS2, and 50 000 for generating ALS3.

The three algorithms ALS1, ALS2 and ALS3 generated by irace are then run on a set of test instances of size 50x20 and 100x20, different from the set of training instances. Also soa-IG is run on the same instances. To avoid differences due to implementation details, we have instantiated soa-IG as one

```
ALS1 Algorithm: ILS(IG)
  $s_0$ := NEH-WSLACK()
  $s^*$ := ILS(perturb_move_insert($k = 6$),
              ILS(soa_ig_perturb($d = 9$),
                  firstImprDescent(strict,
                                   $t_{\mathrm{ls}} = 0.5$),
                  soa_ig_accept($T_c = 0.8956$),
                  $t_{\mathrm{ls}} = 0.8$)
              improvingAccept,
              $t_{\mathrm{ls}} = maxTime$)
  return $s^*$
```

```
ALS2 Algorithm: ILS(ILS(VNS)) :
  $s_0$ := NEH()
  $s^*$ := ILS(perturb_none,
              ILS(perturb_none,
                  ILS(variable_move_insert($k = 1$),
                      firstImprDescent(strict,
                                       $t_{\mathrm{ls}} = 0.4$),
                      improvingStrictlyAccept,
                      $t_{\mathrm{ls}} = 0.4$),
                  metropolisAccept($1548, 56, 0.7447, 7401$),
                  $t_{\mathrm{ls}} = 0.8$),
              improvingAccept,
              $t_{\mathrm{ls}} = maxTime$)
  return $s^*$
```

```
ALS3 Algorithm: ILS(ILS(VNS)) :
  $s_0$ := NEH-WSLACK()
  $s^*$ := ILS(perturb_move_exchange($k = 7$)),
              ILS(soa_ig_perturb($d = 5$),
                  ILS(variable_move_insert($k = 3$),
                      firstImprDescent(strict, $t_{\mathrm{ls}} = 0.3$),
                      improvingStrictlyAccept, $t_{\mathrm{ls}} = 0.4$),
                  metropolisAccept($4969, 48, 0.8356, 8954$), $t_{\mathrm{ls}} = 0.8$),
              alwaysAccept, $t_{\mathrm{ls}} = maxTime$)
  return $s^*$
```

Fig. 4. Hybrid LS algorithms automatically generated for *PFSP-WT*.

specific SLS algorithm through our grammar, taking care that the algorithm is implemented correctly in this way. These test instances were generated by Minella et al. [18] from well-known PFSP instances [22]. Each run is repeated 30 times with different random seeds.

5 Experimental Results

The three algorithm (ALS1, ALS2 and ALS3) generated by irace are shown in Fig. 4. The first one (ALS1) is an IG algorithm within a classical ILS. It uses the NEH-WSLACK initialization, then executes a classical ILS with a k-`insert` move as perturbation, IG as the subsidiary local search, and an `improving` acceptance criterion. The IG has a time limit of $0.8 \cdot maxTime$, and it is represented by an ILS with the construction/deconstruction operator of soa-IG as the perturbation, a first-improvement descent as the subsidiary local search, and the IG acceptance criterion. The first-improvement descent has a time limit of $0.5 \cdot 0.8 \cdot maxTime$. (Note that the first-improvement descent will actually terminate much before its maximum time limit upon finding a local optimum; in fact, the time limits mentioned here and in the following do actually not restrict the computation times of iterative improvement algorithms.)

ALS2 is a VNS algorithm included in an ILS that is itself included in an ILS. ALS2 uses the NEH initialization, then executes a classical ILS without perturbation, an ILS as the subsidiary local search, and an `improving` acceptance criterion. The subsidiary ILS has a time limit of $0.8 \cdot maxTime$, again no perturbation, a VNS as the subsidiary local search, and a `Metropolis` acceptance criterion. The VNS has a time limit of $0.4 \cdot 0.8 \cdot maxTime$, and it is represented as an ILS with a *variable* `insert` move perturbation, a first-improvement descent as the subsidiary local search, and the `improvingStrictly` acceptance criterion. The first-improvement descent has a time limit of $0.4 \cdot 0.4 \cdot 0.8 \cdot maxTime$.

ALS3 is also a VNS algorithm included in an ILS that is itself included in an ILS. Although three levels of recursion were allowed when generating ALS3, this algorithm only has two levels as ALS2. ALS3 uses the NEH-WSLACK initialization, then executes a classical ILS with a k-`exchange` move as perturbation, an ILS as the subsidiary local search, and an acceptance criterion that always accepts a new solution. The subsidiary ILS has a time limit of $0.8 \cdot maxTime$, and uses the construction/deconstruction operator of soa-IG as the perturbation, a VNS as the subsidiary local search, and a `Metropolis` acceptance criterion. The VNS has a time limit of $0.4 \cdot 0.8 \cdot maxTime$ and it is represented as an ILS with a *variable* `insert` move perturbation, a first-improvement descent as the subsidiary local search, and the `improvingStrictly` acceptance criterion. The first-improvement descent has a time limit of $0.3 \cdot 0.4 \cdot 0.8 \cdot maxTime$ seconds.

Comparison with the state-of-the-art algorithm. To assess the performance of the three automatically generated algorithms, we run them 30 times on the test instances and compare them with soa-IG. Fig. 5 and 6 show the solution cost reached by each algorithm on each instance. Table 2 gives the best and mean solution. The behavior of the algorithms is slightly different depending on the instance size. The performance of the automatically generated SLS algorithms on the `50x20` instances matches the quality obtained by soa-IG in most instances, and they are noticeably better on a few. On the `100x20` instances, the automatic SLS algorithms clearly outperform soa-IG.

In order to assess the performance over each set of instances, we perform a statistical analysis based on the Friedman test for analyzing non-parametric unreplicated complete block designs, and its associated post-hoc test for multiple comparisons [4]. First, we pair the runs performed on the same instance using the same random seed. This is the blocking factor, and the different algorithms are the treatment factor. Algorithms are ranked within each block, lower solution cost corresponds to lower rank. If the Friedman test rejects the hypothesis that the different algorithms obtain the same mean rank, then we calculate the difference (ΔR) between the sum of ranks of each algorithm and the best ranked one (with the lowest sum of ranks). We also calculate the minimum difference between the sum of ranks of two algorithms that is statistically significant (ΔR_α), given a significance level of $\alpha = 0.05$. Table 3 gives the results of this analysis, applied separately to the two sets of instances of size `50x20` and `100x20`. We indicate in bold face the best strategy (the one having the lowest sum of ranks) and those that are not significantly different from the best one. In both cases, the best ranked algorithm is significantly better than the rest. However, the best algorithm is different in each case. Notably, soa-IG is consistently ranked as the worst by a large margin, especially on the `100x20` instances. These results are consistent with the observations above. Therefore, our conclusion is that the current state of the art can be matched and outperformed by the automatically generated algorithms.

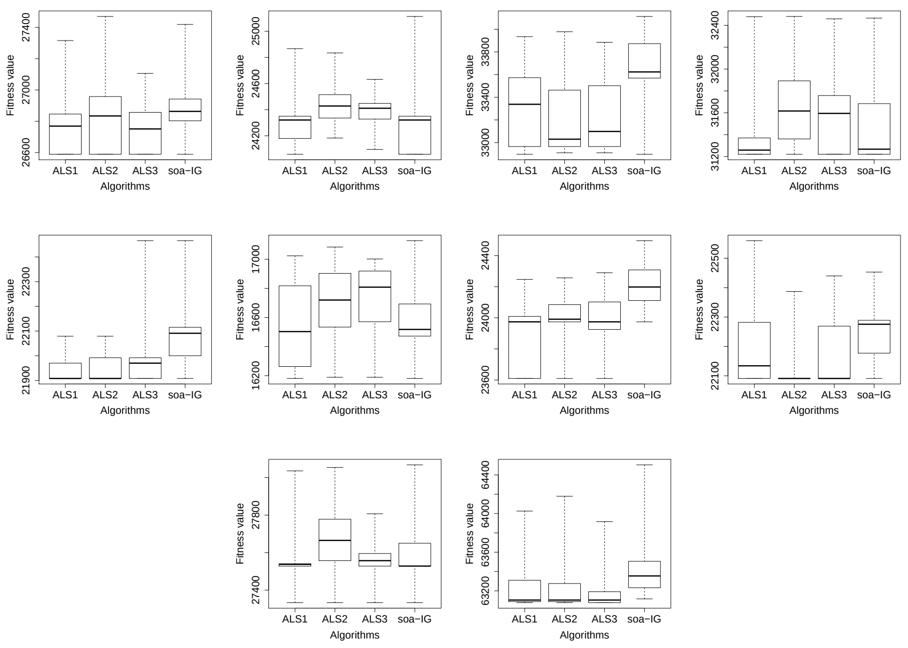

Fig. 5. Solution costs obtained by the three automatic SLS algorithms (ALS1, ALS2 and ALS3) and soa-IG on the `50x20` instances.

6 Conclusion

Hybridizations of stochastic local search (SLS) methods that manipulate a single solution at each step of the search are among the most effective non exact algorithms for tackling hard combinatorial optimization problems [11]. Nonetheless, designing such hybrid SLS algorithms is an arduous task that requires a significant amount of effort in implementation, experimental setup and analysis. In practice, algorithm designers only consider a few ad-hoc combinations of SLS algorithms. In this paper, we have shown that the process of designing such algorithms can become mostly automatic. In particular, we have proposed a unified and practical generalized local search (GLS) structure.

We have shown that the GLS structure *unifies* the formulation of various simple SLS methods and their possible combinations (hybridizations) into a single structure. In fact, the best algorithms generated when applied to the *PFSP-WT* are complex hybrids that combine ILS with IG, VNS and even a different ILS. Our proposal is also *practical*, in the sense that it generates algorithms that are as efficient as if they were hand-crafted by a competent programmer. Two properties of our proposal are key for obtaining such efficiency. First, instead of a complex algorithmic framework with many parameters, our system generates specific algorithms from a grammar description of the GLS structure.

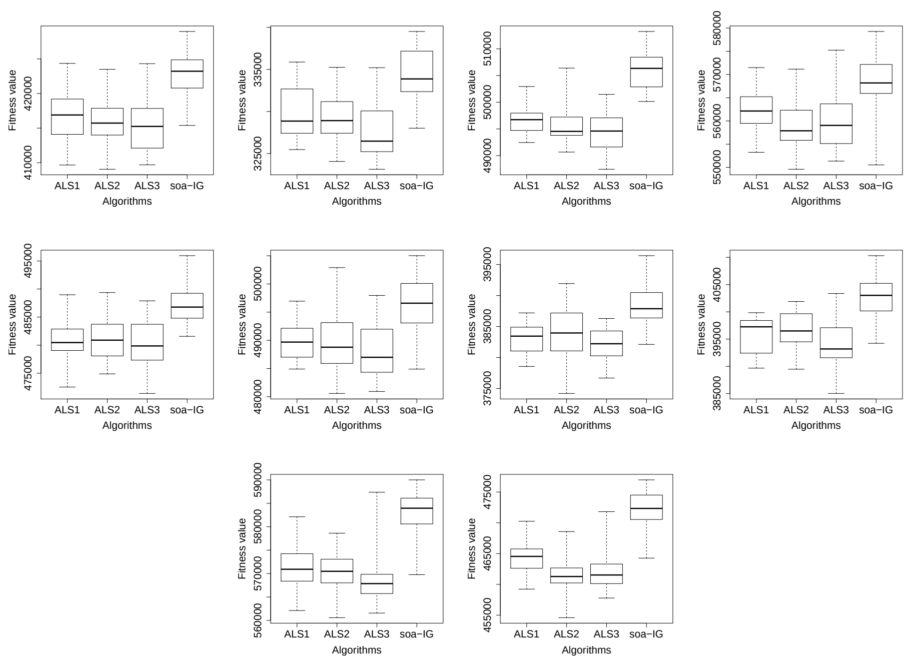

Fig. 6. Solution costs obtained by the three automatic SLS algorithms (ALS1, ALS2 and ALS3) and soa-IG on the `100x20` instances.

These specific algorithms, which contain only a small fraction of all the algorithmic components available in the grammar, are generated directly as `C++` code and compiled. Second, our grammar description allows algorithm designers to include problem-specific components, which are often crucial for obtaining high-performing SLS algorithms. The system takes care of combining, testing and selecting (or discarding) these problem-specific components among all the available algorithmic components.

We have evaluated our proposal by applying our method to the *PFSP-WT* comparing it the a state-of-the-art IG algorithm. Our experimental results showed that the three automatically generated SLS algorithms are able to outperform it on well-known *PFSP-WT* instances from the literature.

Despite this initial success, there is considerably room for improvement. First, we used a moderate amount of computing effort for the automatic generation of algorithms. Therefore, the real potential of the proposed GLS structure may not have been exhausted. Second, the definition of the GLS structure is a work in progress. Future work will extend the GLS structure presented here, and its implementation, to include additional SLS algorithms, for example, greedy randomized adaptive search procedure (GRASP) [8] and Tabu Search [9]. Additionally, possible re-designs of specific aspects of the GLS structure and its implementation may be considered once more computational results are gathered.

Table 2. Solution costs obtained by the three automatic SLS algorithms and soa-IG on the test instances.

Instances		ALS1		ALS2		ALS3		soa-IG	
		Best	Avg	Best	Avg	Best	Avg	Best	Avg
50x20	ta051	26589	26806.2	26589	26824.9	26589	**26756.1**	26589	26899.6
	ta052	24059	**24273.2**	24183	24443.2	24096	24390.8	24059	24333.5
	ta053	32897	33307.8	32910	**33183.7**	32910	33206.8	32897	33634.6
	ta054	31221	**31470.2**	31221	31663.3	31221	31572.7	31221	31488.9
	ta055	21908	**21936.2**	21908	21948.3	21908	21975.9	21908	22094.7
	ta056	16181	**16516.4**	16189	16711.6	16189	16740.7	16181	16556.7
	ta057	23610	**23869**	23610	23990.4	23610	23953.9	23974	24211.2
	ta058	22091	22207.7	22091	**22131.9**	22091	22166.8	22091	22262.1
	ta059	27333	**27521.3**	27333	27685.1	27333	27573.1	27333	27577.9
	ta060	63078	63286.3	63078	63235.9	63078	**63179.9**	63117	63456
100x20	ta081	409667	416932.6	409052	415941.5	409697	415306.3	415388	422625
	ta082	325472	329803.8	324060	329161.3	323133	**327466.6**	328014	334437.1
	ta083	492455	496922.6	490669	495669.7	487450	**494569.8**	500142	505772
	ta084	553249	562380.8	549600	**558824.5**	551359	559419.9	550536	568600.5
	ta085	472546	480861	474883	481147.7	471402	**479941.3**	481576	487291.6
	ta086	484905	490357.9	480575	489379	480926	**488144.7**	484892	496511.1
	ta087	378567	382931.6	374208	384024.5	376694	**382277.9**	382122	388511.8
	ta088	389673	395809.4	389475	396729.7	385029	**394056.2**	394226	402836.5
	ta089	562109	571495.2	560593	570489.5	561570	**568465.7**	569769	582829.9
	ta090	459232	464206.4	454597	**461262.9**	457784	462177.3	464264	471961.3

Table 3. Statistical analysis based on the Friedman-test. The second column gives the minimum difference in the sum of ranks that is statistically significant (ΔR_α), given a significance level of $\alpha = 0.05$. For each instance set, algorithms are ordered according to the rank obtained. The numbers in parenthesis are the difference of ranks relative to the best algorithm. The algorithm that is significantly better than the other ones is indicated in bold face.

Instances	ΔR_α	**Algorithms** (ΔR)
50x20	57.92	**ALS1**, ALS3 (75), ALS2 (115.5), soa-IG (221.5)
100x20	47.04	**ALS3**, ALS2 (100), ALS1 (143), soa-IG (573)

Third, additional techniques may prove to be useful for avoiding too complex SLS algorithm designs. Finally, we will apply the proposed method to other hard combinatorial problems, with the aim of improving the state of the art.

References

1. Balaprakash, P., Birattari, M., Stützle, T.: Improvement strategies for the F-race algorithm: Sampling design and iterative refinement. In: Bartz-Beielstein, T., et al. (eds.) Hybrid Metaheuristics, LNCS, vol. 4771, pp. 108–122. Springer (2007)
2. Burke, E.K., Hyde, M.R., Kendall, G.: Grammatical evolution of local search heuristics. IEEE Transactions on Evolutionary Computation 16(7), 406–417 (2012)
3. Cahon, S., Melab, N., Talbi, E.G.: ParadisEO: A framework for the reusable design of parallel and distributed metaheuristics. Journal of Heuristics 10(3), 357–380 (2004)

4. Conover, W.J.: Practical Nonparametric Statistics. John Wiley & Sons, New York, NY, third edn. (1999)
5. Du, J., Leung, J.Y.T.: Minimizing total tardiness on one machine is NP-hard. Mathematics of Operations Research 15(3), 483–495 (1990)
6. Dubois-Lacoste, J.: A study of Pareto and Two-Phase Local Search Algorithms for Biobjective Permutation Flowshop Scheduling. Master's thesis, IRIDIA, Université Libre de Bruxelles, Belgium (2009)
7. Dubois-Lacoste, J., López-Ibáñez, M., Stützle, T.: A hybrid TP+PLS algorithm for bi-objective flow-shop scheduling problems. Computers & Operations Research 38(8), 1219–1236 (2011)
8. Feo, T.A., Resende, M.G.C.: Greedy randomized adaptive search procedures. Journal of Global Optimization 6, 109–113 (1995)
9. Glover, F.: Tabu search – Part I. INFORMS Journal on Computing 1(3), 190–206 (1989)
10. Hansen, P., Mladenovic, N.: Variable neighborhood search: Principles and applications. European Journal of Operational Research 130(3), 449–467 (2001)
11. Hoos, H.H., Stützle, T.: Stochastic Local Search—Foundations and Applications. Morgan Kaufmann Publishers, San Francisco, CA (2005)
12. Hutter, F., Hoos, H.H., Leyton-Brown, K., Stützle, T.: ParamILS: an automatic algorithm configuration framework. Journal of Artificial Intelligence Research 36, 267–306 (Oct 2009)
13. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. Science 220, 671–680 (1983)
14. López-Ibáñez, M., Dubois-Lacoste, J., Stützle, T., Birattari, M.: The irace package, iterated race for automatic algorithm configuration. Tech. Rep. TR/IRIDIA/2011-004, IRIDIA, Université Libre de Bruxelles, Belgium (2011),
15. Lourenço, H.R., Martin, O., Stützle, T.: Iterated local search: Framework and applications. In: Gendreau, M., Potvin, J.Y. (eds.) Handbook of Metaheuristics, chap. 9, pp. 363–397. Springer, 2 edn. (2010)
16. Mascia, F., López-Ibáñez, M., Dubois-Lacoste, J., Stützle, T.: From grammars to parameters: Automatic iterated greedy design for the permutation flow-shop problem with weighted tardiness. In: Learning and Intelligent Optimization, 7th International Conference, LION 7. LNCS, Springer (2013), to appear
17. Mckay, R.I., Hoai, N.X., Whigham, P.A., Shan, Y., O'Neill, M.: Grammar-based genetic programming: A survey. Genetic Programming and Evolvable Machines 11(3-4), 365–396 (Sep 2010)
18. Minella, G., Ruiz, R., Ciavotta, M.: A review and evaluation of multiobjective algorithms for the flowshop scheduling problem. INFORMS Journal on Computing 20(3), 451–471 (2008)
19. Nawaz, M., Enscore, Jr, E., Ham, I.: A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. OMEGA 11(1), 91–95 (1983)
20. Papadimitriou, C.H., Steiglitz, K.: Combinatorial Optimization – Algorithms and Complexity. Prentice Hall, Englewood Cliffs, NJ (1982)
21. Ruiz, R., Stützle, T.: A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. European Journal of Operational Research 177(3), 2033–2049 (2007)
22. Taillard, É.D.: Benchmarks for basic scheduling problems. European Journal of Operational Research 64(2), 278–285 (1993)