Analysis of stochastic local search methods for the unrelated parallel machine scheduling problem

Haroldo G. Santos*[a], Túlio A.M. Toffolo[a,b], Cristiano L.T. Silva[a], and Greet Vanden Berghe[b]

[a]Federal University of Ouro Preto, Department of Computing - Brazil
[b]KU Leuven, Department of Computer Science, CODeS & iMinds-ITEC - Belgium

Abstract

This work addresses the unrelated parallel machine scheduling problem with sequence-dependent setup times, in which a set of jobs must be scheduled for execution by one of several available machines. Each job has a machine-dependent processing time. Furthermore, given multiple jobs, there are additional setup times, which vary based on the sequence and machine employed. The objective is to minimize the schedule's completion time (makespan). The problem is NP-hard and of significant practical relevance. The present paper investigates the performance of four different stochastic local search (SLS) methods designed towards solving the particular scheduling problem: Simulated Annealing, Iterated Local Search, Late Acceptance Hill-Climbing and Step Counting Hill-Climbing. The analysis focuses on design questions, tuning effort and optimization performance. Simple neighborhood structures are considered. All proposed SLS methods performed significantly better than two state-of-the-art hybrid heuristics, especially for larger instances. Updated best-known solutions were generated for 901 out of the 1,000 large benchmark instances considered, demonstrating that particular stochastic local search methods are simple yet powerful alternatives to current approaches for addressing the problem. Implementations of the contributed algorithms have been made available to the research community.

1 Introduction

The present paper investigates different stochastic local search (SLS) methods for the unrelated parallel machine scheduling problem with sequence-dependent setup times (UPMSP). In this problem, a set of jobs must be executed by a set of machines. Each job has a processing time associated with each machine and the setup time between two jobs is both sequence and machine-dependent. The objective is to minimize the completion time of all jobs, called makespan.

The UPMSP is a generalization of many classical parallel machine scheduling problems. It is highly relevant in production planning, when considering production lines with heterogeneous machines. The problem is extremely computationally challenging

*Corresponding author. E-mail: haroldo@iceb.ufop.br

and most instances in the literature are unsolved. Indeed, no proven optimal solutions are known for the 1,000 largest instances in the benchmark set proposed by Vallada and Ruiz (2011).

Recent literature indicates that hybrid methods are generally employed to approach the UPMSP. These methods incorporate ideas from several metaheuristics in the quest of producing better solvers. Such hybridized approaches have numerous disadvantages: : (*i*) they generally require many parameters to tune; (*ii*) by executing several optimization phases, these methods tend to be computationally expensive; (*iii*) it is difficult to identify which parts of the solver are the most effective and, most importantly, (*iv*) there is a well known statistical correlation (McConnell, 2004) in software industry where the number of bugs grows proportionally with respect to the number of lines of code: simple methods are more likely to be implemented without bugs than more complex ones, a very important consideration for operations research practitioners. The present paper investigates methods which contrast with this trend, by relying solely on simple local search. It is subsequently demonstrated how appropriately implemented and tuned SLS methods produce high quality solutions in a short length of time, outperforming other state of the art heuristic methods.

The investigation includes: (*i*) an analysis of parameter tuning challenges, (*ii*) a discussion and experiments on diversification and intensification strategies for the UPMSP, and (*iii*) improved best-known results.

The present paper is organized as follows. Section 2 presents the UPMSP in detail, including a literature review of heuristic methods. The neighborhoods used in all SLS methods implemented are described in Section 3. Section 4 details the constructive algorithm and the design of the four stochastic local search methods we tested: Simulated Annealing, Late Acceptance Hill-Climbing, Step Counting Hill-Climbing and Iterated Local Search. Section 5 presents an extensive computational study of parameter tuning for these methods and the obtained results. Finally, conclusions are discussed in Section 6.

2 Problem description and literature review

The UPMSP is denoted as $R|S_{ijk}|C_{max}$ in the $\alpha|\beta|\gamma$ notation introduced by Graham et al. (1979). It is a generalization of the parallel machine scheduling problem $P||C_{max}$, and therefore computationally challenging (NP-hard) even when only two machines are considered (Garey and Johnson, 1979). A complete survey on scheduling problems with setup times, including a discussion on the main algorithmic approaches, is presented by Allahverdi et al. (2008). The following notation is used to refer to the UPMSP:

Let $J = \{j_1, ..., j_n\}$ be the set of n jobs and $M = \{m_1, ..., m_k\}$ the set of k machines, such that:

1. each job j must be executed exactly once and by only one machine;
2. each job j has a processing time p_{mj} if executed by machine m;
3. machine m requires s_{mij} setup time to execute job j directly after job i;
4. machine m requires s_{mii} setup time to execute job i if i is its initial job.

The problem's objective is to allocate all jobs in J to the machines in M such that the completion time of the last job (makespan) is as early as possible. The machine executing the last job is hereafter called the makespan machine.

Kim et al. (2002) presented an interesting application of the UPMSP in the compound semiconductor wafer industry. The dicing operation of semiconductor wafers involves the processing of wafers on several machines with different technologies and manufacturers. Each machine must be prepared according to the type of wafer to be processed. There is a resulting setup time that is both machine and sequence-dependent.

Rabadi et al. (2006) implemented a Greedy Randomized Adaptive Search Procedure (GRASP) (Feo and Resende, 1995) style algorithm and also generated some instances. De Paula et al. (2007) and Avalos-Rosales et al. (2015) also proposed GRASP-like algorithms combined with Variable Neighborhood Search (Hansen and Mladenović, 1997).

Vallada and Ruiz (2011) proposed a set of challenging instances for the problem (SOA-2011). They designed a genetic algorithm (Goldberg, 1989) combined with local search, which they describe as capable of achieving good results within a small amount of time. Their results significantly improved upon those obtained by Rabadi et al. (2006). More recently, Cota et al. (2014) proposed a multi-neighborhood hybrid metaheuristic including path relinking (Glover et al., 2000) for polishing solutions. Their results generally improved the original results of Vallada and Ruiz (2011), presenting new best solutions for most instances.

The application of exact integer programming solvers was evaluated in Avalos-Rosales et al. (2013), using integer programming reformulations. Improvements were obtained when solving the set of small instances with less than 50 jobs proposed by Vallada and Ruiz (2011).

3 Neighborhood structures

The fundamental parts of any local search method are the neighborhood structures. Before detailing any specifics, certain characteristics of the UPMSP should be mentioned:

- only changes involving the makespan machine can improve a solution;
- to escape from local optima, a sequence of modifications involving different machines may be needed, such as: swapping jobs between machines or altering the processing sequence on these machines;
- depending on the instance characteristics, some neighborhoods may be more important than others as the setup times of particular instances may be more significant than the processing times, and vice-versa.

Considering these characteristics, strategies to intensify and diversify the search were developed. The first aspect that defines each strategy is the selection of the main machine involved in the neighbor generation, denoted hereafter as m_x. Two possibilities are considered: (i) selecting a random machine as m_x and (ii) selecting the makespan machine as m_x. The second aspect that defines a neighborhood generation

strategy is the cardinality of the evaluated neighbor set. Again, two possibilities are considered:

1. ***regular* policy**: a single random neighbor solution is evaluated;
2. ***intensification* policy**: a small subset of neighbors is analyzed and the best one is returned.

All possible combinations of these aspects – selection of machine m_x and cardinality of the evaluated neighbor set – generate four different neighbor generation strategies. They all have the same probability of being chosen by the neighborhoods at each iteration.

Six neighborhood structures were developed to explore the search space. To simplify the final algorithm, all neighborhoods have the same probability of being selected during the local search methods. These neighborhoods are covered in detail in the following paragraphs.

Shift neighborhood

A neighbor in the *shift* neighborhood is generated by re-scheduling a random job from a machine m_x to another position on the same machine. The position is randomly chosen, except when the *intensification* policy is selected. In this case, the target position of the job is greedily selected, whereby the position incurring the shortest makespan for the machine is chosen. Figure 1 shows how a *shift* neighbor is generated.

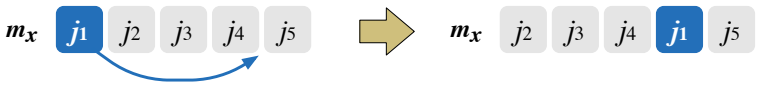

Figure 1: Example of a neighbor in the *shift* neighborhood.

Switch neighborhood

A neighbor in the *switch* neighborhood structure is generated by switching the order of two jobs on a machine m_x. The first job is always randomly chosen, while the second may be greedily chosen if the *intensification* policy is considered. In this case, the second job would be the one which when switched with the first results in the largest decrease (or smallest increase) in the machine's processing time. Both jobs are randomly selected if the *regular* policy is considered. Figure 2 shows an example of a neighbor in this structure.

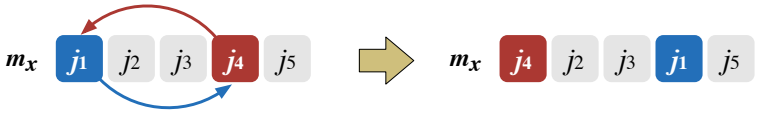

Figure 2: Example of a neighbor in the *switch* neighborhood.

Task move neighborhood

A neighbor in the *task move* neighborhood is generated by moving a random job from a source machine m_x to a random target machine m_y. If the *intensification* policy is considered, the job is positioned at the best position on the target machine, which is the position that minimizes the makespan of machine m_y. Otherwise, a random position is selected. Figure 3 shows an example of a *task move* neighbor generation.

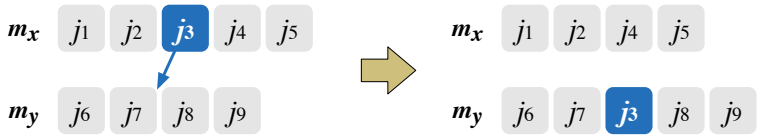

Figure 3: Example of a neighbor in the *task move* neighborhood.

Swap neighborhood

A neighbor in the *swap* neighborhood is generated by swapping two jobs between two machines, m_x and m_y. The jobs are randomly selected. Figure 4 presents an example of a neighbor in this structure. It is noteworthy that the swapped jobs are placed in random positions on the other machine. Only when applying the *intensification* policy, the positions are greedily selected to minimize the individual makespans of the machines.

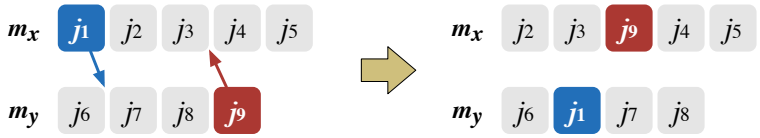

Figure 4: Example of a neighbor in the *swap* neighborhood.

2-shift neighborhood

A neighbor of the *2-shift* neighborhood is generated by shifting the position of two jobs executed on the same machine m_x. It is equivalent to applying the *shift* neighborhood move twice, respecting the same rules. Figure 5 shows an example of a *2-shift* neighbor.

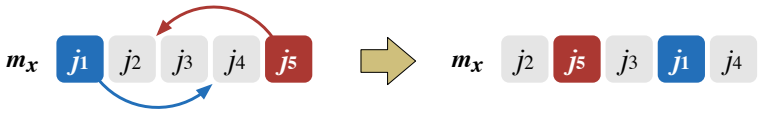

Figure 5: Example of a neighbor in the *2-shift* neighborhood.

Direct swap neighborhood

A neighbor in the *direct swap* neighborhood is generated by swapping two jobs between two machines, m_x and m_y, maintaining the previous positions on these machines. Jobs are randomly selected. When the *intensification* policy is employed, the first job is taken at random while the second job is greedily selected, such that the swap minimizes the makespan of machine m_x. Figure 6 presents an example of a neighbor in this structure.

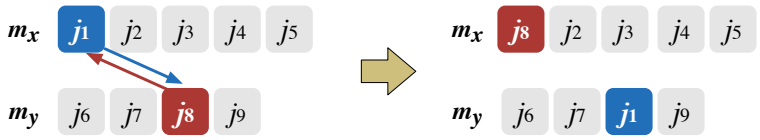

Figure 6: Example of a neighbor in the *direct swap* neighborhood.

4 Stochastic Local Search methods

This work proposes introducing stochastic local search (SLS) techniques (Hoos and Stützle, 2005) to the challenging UPMSP. It is demonstrated how great results can be obtained using simple and easy to implement SLS methods, such as the traditional Simulated Annealing (Kirkpatrick et al., 1983) and Iterated Local Search (Lourenço et al., 2003) methods, as well as the more recent Late Acceptance Hill-Climbing (Burke and Bykov, 2012) and Step Counting Hill-Climbing (Bykov and Petrovic, 2013) methods. These methods are used to control the exploration of the neighborhoods presented in Section 3. By employing different mechanisms for accepting neighbor solutions, different intensification and diversification strategies are implemented.

Before engaging in the SLS methods, the procedure for generating initial solutions is presented in Section 4.1. Next, the four implemented SLS methods are described.

4.1 Constructive method

Robust local search methods should produce equally good solutions regardless of the quality of the initial solution received as input. Thus, the implemented algorithms employ a fast and simple randomized constructive algorithm: each task is sequentially assigned to a randomly selected machine.

4.2 Simulated Annealing

Proposed by Kirkpatrick et al. (1983), Simulated Annealing (SA) is a probabilistic metaheuristic based on an analogy with a thermodynamics simulation of the cooling of a set of heated atoms. The main procedure consists of a loop that randomly generates, at each iteration, one neighbor S' of the current solution S. Let Δ be the variation of the objective function value incurred when moving from S to S', such that $\Delta = f(S') - f(S)$. The method immediately accepts the candidate solution if $\Delta \leq 0$. If $\Delta > 0$, the candidate may be accepted with a probability $e^{-\Delta/T}$, where

T is a parameter called temperature, which regulates the probability of accepting worsening solutions.

The temperature is initially set to a value t_0. After a fixed number of iterations sa_{max}, the temperature is gradually lowered by a cooling rate α. The temperature in a stage q is given by $t_q \leftarrow \alpha \times t_{q-1}$, with $0 < \alpha < 1$ (geometric cooling). With this procedure, a greater chance of avoiding local optima occurs at the initial iteration and as t approaches zero the algorithm behaves like a descent method, reducing the likelihood of accepting worsening solutions (Henderson et al., 2003). To prevent stagnation, reheating may be performed when the temperature reaches a minimum threshold ϵ.

Algorithm 1 shows the implemented SA's pseudo-code, where $f(.)$ is the objective function and $N_k(.)$ is a function that returns a random neighbor solution of neighborhood structure k. The algorithm requires the following arguments:

S : initial solution;

t_0 : initial temperature;

α : cooling rate;

sa_{max} : number of iterations processed at each temperature.

Algorithm 1: Simulated Annealing (SA)

```
   Input: S, t_0, α, sa_max
 1 S* ← S
 2 t ← t_0
 3 while time limit is not reached do
 4     for i ← 1 to sa_max do
 5         S' ← random neighbor of a random neighborhood k, S' ∈ N_k(S)
 6         Δ ← f(S') − f(S)
 7         if Δ ≤ 0 then
 8             S ← S'
 9             if f(S') < f(S*) then S* ← S';
10         else
11             take a random x ∈ [0, 1]
12             if x < e^{−Δ/t} then S ← S';
13     t ← α × t
14     if t < ε then t ← t_0;
15 return S*
```

4.3 Iterated Local Search

Iterated Local Search (ILS) (Lourenço et al., 2003) is based on the idea that local search procedures achieve better results by optimizing different solutions generated by perturbing locally optimal solutions.

The implemented ILS algorithm (Algorithm 3) begins from an initial solution S and applies perturbations of size p_{size} to S, followed by a descent method. A perturbation is the unconditional acceptance of a neighbor generated by any of the neighborhoods presented in Section 3. The perturbations are initially small, generating solutions in the current solution's vicinity. As better solutions become harder to

find, the perturbation increases so that solutions farther than the current one can be reached, resulting in diversification.

The descent phase employs a straightforward random non-ascendant (RNA) method which only accepts improving neighbors or sideways moves (those producing different solutions with the same cost). Evidently, this is only one of a large set of design options for the descent phase. Other methods may consider different acceptance criteria. The implemented RNA method is presented by Algorithm 2. The parameter rna_{max} establishes a limit on the number of consecutive non-improving iterations in the RNA.

Algorithm 2: Random non-ascendant method (RNA)

```
   Input: S, rna_max
1  i ← 1
2  while i < rna_max do
3  |  S' ← random neighbor of a random neighborhood k, S' ∈ N_k(S)
4  |  if f(S') ≤ f(S) then
5  |  |  if f(S') < f(S) then i ← 0;
6  |  |  S ← S'
7  |  i ← i + 1
8  return S
```

The descent phase produces a solution that is accepted only if it improves the best solution, in which case the perturbation size p_{size} is reset to p_0. If the non-improvement ILS iterations counter i reaches a limit $iter_{max}$, perturbation size p is incremented until it exceeds bound p_{max}, at which point it is reset to its initial size (p_0). Algorithm 3 presents the ILS procedure implemented. Five arguments are required:

S : initial feasible solution;

p_0 : number of moves made in the initial perturbation level;

p_{max} : maximum number of moves made in one perturbation, multiplier of p_0;

rna_{max} : maximum number of consecutive non-improving moves made in the RNA descent phase;

$iter_{max}$: maximum number of consecutive non-improving iterations at the same perturbation level.

Algorithm 3: Iterated Local Search (ILS)

Input: S, p_0, p_{max}, rna_{max}, $iter_{max}$

```
S* ← S ← RNA(S, rna_max)
i ← 0
p ← p_0
p_max ← p_0 × p_max
while time limit is not reached do
    for j ← 1 to p do
        S ← random neighbor S' of a random neighborhood k, S' ∈ N_k(S)
    S ← RNA(S, rna_max)
    if f(S) < f(S*) then
        S* ← S
        i ← 0
        p ← p_0
    else
        S ← S*
        i ← i + 1
    if i ≥ iter_max then
        i ← 0
        p ← p + p_0
        if p > p_max then p ← p_0;
return S*
```

4.4 Late Acceptance Hill-Climbing

Late Acceptance Hill-Climbing (LAHC) is a metaheuristic introduced by Burke and Bykov (2008). It is an adaptation of the classical Hill-Climbing heuristic that considers the last l solutions when accepting or rejecting a neighbor. Note that a candidate solution may be accepted even if it is worse than the current solution, since it is compared against the solution obtained l iterations ago. The algorithm has only one parameter: the number l of most recent solutions considered. Both LAHC and Step Counting Hill-Climbing, which will be presented in the next subsection, share similarities with the Threshold Accepting (TA) metaheuristic, proposed by Dueck and Scheuer (1990). In fact, they can be interpreted as specializations of the TA metaheuristic, employing different mechanisms to update the threshold.

LAHC was created with three goals in mind: (*i*) to be a one-point search procedure that does not employ an artificial cooling schedule as SA does; (*ii*) to effectively use the information collected during previous iterations of the search; and (*iii*) to employ an acceptance mechanism almost as simple as Hill-Climbing's (Burke and Bykov, 2012).

Algorithm 4 presents the pseudo-code of the LAHC implemented. A list $F = \{f_0, ..., f_{l-1}\}$ of solution costs is stored. This list is initially filled with the cost of the initial solution S: $f_v \leftarrow f(S)\ \forall v \in \{0, ..., l-1\}$. At each iteration, a candidate solution S' is generated. The candidate solution is accepted if it improves the current solution or if its cost is smaller than or equal to the cost stored at position v of list F. If this solution improves upon the best solution found, it is updated: $S^* \leftarrow S'$. Next, the cost at position v is updated: $f_v \leftarrow f(S)$. This process repeats until the

stopping criterion is reached.

Algorithm 4: Late Acceptance Hill-Climbing (LAHC)

```
Input: initial solution S and list size l
f_v ← f(S) ∀v ∈ {0, ..., l - 1}
S* ← S
v ← 0
while time limit is not reached do
    S' ← random neighbor of a random neighborhood k, S' ∈ N_k(S)
    if f(S') ≤ f(S) or f(S') ≤ f_v then
        S ← S'
        if f(S) < f(S*) then S* ← S;
    f_v ← f(S)
    v ← (v + 1) mod l
return S*
```

4.5 Step Counting Hill-Climbing

The Step Counting Hill-Climbing method (SCHC) introduced by Bykov and Petrovic (2013), much like LAHC, was designed to perform a one point search combining diversification and intensification search strategies. Its advantage is that it is even simpler to implement than LAHC: the threshold control uses a scalar, instead of a list, to control the acceptance of worsening moves.

The SCHC (Algorithm 5) maintains a bound b that limits the acceptance of worsening moves. This bound remains unchanged for a number of iterations, and is only updated at an interval c, which is the algorithm's single parameter.

Algorithm 5 presents the pseudo-code of the implemented SCHC method.

Algorithm 5: Step Counting Hill-Climbing (SCHC)

```
Input: S, c
Output: Best solution S* found.
S* ← S
b ← f(S)
i ← 0
while time limit is not reached do
    S' ← random neighbor of a random neighborhood k, S' ∈ N_k(S)
    if f(S') ≤ f(S) or f(S) < b then
        S ← S'
        if f(S) < f(S*) then S* ← S;
    if i ≥ c then
        b ← f(S)
        i ← 0
    i ← i + 1
return S*
```

5 Computational experiments

This section presents an extensive experimental evaluation of the proposed SLS methods. Experiments were conducted on the set of 1,640 benchmark instances proposed by Vallada and Ruiz (2011), for which best known solutions are recorded in SOA-ITI (2013). These instances have $n \in \{6, 8, 10, 12, 50, 100, 150, 200, 250\}$ jobs and $k \in \{2, 3, 4, 5, 10, 15, 20, 25, 30\}$ machines. Vallada and Ruiz (2011) divide these instances into two groups: small ($n \leq 12$ and $k \leq 5$) and large ($n \geq 50$ and $k \geq 10$). Processing times are uniformly distributed between $\{1, \ldots, 99\}$. Considering the setup times, there are four groups of instances, with setup costs uniformly distributed in the following ranges: $\{1, \ldots, 9\}$, $\{1, \ldots, 49\}$, $\{1, \ldots, 99\}$ and $\{1, \ldots, 124\}$.

The algorithms were coded in Java 1.8 and the experiments were executed on an Intel® Core$^{\text{TM}}$i7-4790 3.6Ghz computer with 16Gb of RAM memory, running the openSUSE 13.2 linux operating system. This computer is approximately 2.5 times faster (PassMark software, 2015) at running sequential applications than the computer equipped with an Intel® Core 2$^{\text{TM}}$duo E6600 @ 2.4Ghz used by Vallada and Ruiz (2011) in their experiments.

Time limits in Vallada and Ruiz (2011) were computed as $n \times (\frac{k}{2}) \times t$ ms, with $t = 10, 30$ and 50. To achieve a fair comparison between all algorithms, the time limits of Vallada were scaled down by the factor suggested in the benchmarks of PassMark software (2015). Consequently, all algorithms were executed with an approximately equal amount of processing power. When $t = 50$, for example, a time limit of 75 seconds is imposed, whereas the execution time of Vallada and Ruiz (2011) was 187.5 seconds. The authors wish to adhere to the guidelines of good laboratory practice for optimization research (Kendall et al., 2016) and thus the source code and all solutions are provided at `http://www.goal.ufop.br/software/upmsp`.

The evaluation of solutions employs a metric insensitive to scales: the relative percentage deviation (RPD), presented in Equation 1. Vallada and Ruiz (2011) employed the same metric[1].

$$\text{RPD} = 100 \times \frac{\text{Method solution} - \text{Best known solution}}{\text{Best known solution}} \tag{1}$$

The remainder of this section is organized as follows. A detailed parameter tuning investigation is discussed in Section 5.1. Section 5.2 analyses the impact of the various neighborhoods. Finally, Section 5.3 presents the final results obtained by the SLS methods and compares them with the state of the art methods in the literature.

5.1 Parameter tuning

A large set of experiments was conducted to discover the best parameter configuration for each algorithm. The parameter tuning phase considered a separate training set of 164 randomly selected instances[2], to avoid overfitting. All experiments, including

[1] Best known solutions were updated with the results obtained with the algorithms proposed by the present paper.

[2] The instances in the training set were selected using the default random numbers generator in the bash shell script programming language, version 4.2.

those considered during the parameter tuning phase, respected the same time limits previously detailed.

Firstly, a manually-selected set of values for each parameter was evaluated considering its many combination with others. The objective is understanding the possible correlations between parameters, while simultaneously evaluating each algorithm's sensitivity to various parameter settings.

Secondly, the iRace package was used. This package implements the iterated racing procedure (López-Ibánez et al., 2011; López-Ibánez and Stützle, 2014), which is an extension of iterated F-race (I/F-Race) proposed by Birattari et al. (2007). The primary function of iRace is the automatic configuration of optimization algorithms or, in other words, determining the most appropriate parameter settings for an optimization method. iRace is implemented as an R package (R Development Core Team, 2008) and builds upon the race package.

5.1.1 Manual tuning

Figure 7 depicts the RPD obtained using different parameters settings for each of the four implemented SLS methods.

Both LAHC and SCHC have only one parameter to tune. These methods were tested with $l, c \in \{10, 100, 1000, 5000, 10000, 50000, 100000, 500000, 1000000\}$. As presented in Figure 7, SCHC appears less sensitive to parameter changes than LAHC. SCHC proves superior to LAHC given that it is not only easier to implement, but also requires no additional memory structure.

Both SA and ILS have multiple parameters. During testing, various combinations of parameter values were evaluated. SA was evaluated with 27 triples of the following parameters $t_0 \in \{1, 1.5, 3\}, \alpha \in \{0.9, 0.95, 0.99\}, sa_{max} \in \{1000, 10000, 100000\}$. In ILS 27 triples with $rna_{max} \in \{1000, 500000, 1000000\}, p_0 \in \{5, 10, 100\}, p_{max} \in \{2, 4, 8\}$ ($iter_{max}$ was fixed in 1000) were tested. Figure 7 illustrates how this initial sampling of different parameters produced many low average RPDs for SA. After evaluating many iterations with different parameter combinations for the SA algorithm, it is observed that most combinations of different parameter values performed equally well. The implemented ILS was more sensitive to parameter changes: some parameter values significantly degenerated the quality of the produced RPDs. These conclusions, however, may be biased by the initial parameter sampling and by our different expertise in tuning these algorithms. To explore the search in the parameter space more systematically, all algorithms were submitted to an automated tuning procedure, thus enabling an evaluation of how well a state of the art tuning algorithm discovers good parameter settings. These results are discussed in the following section.

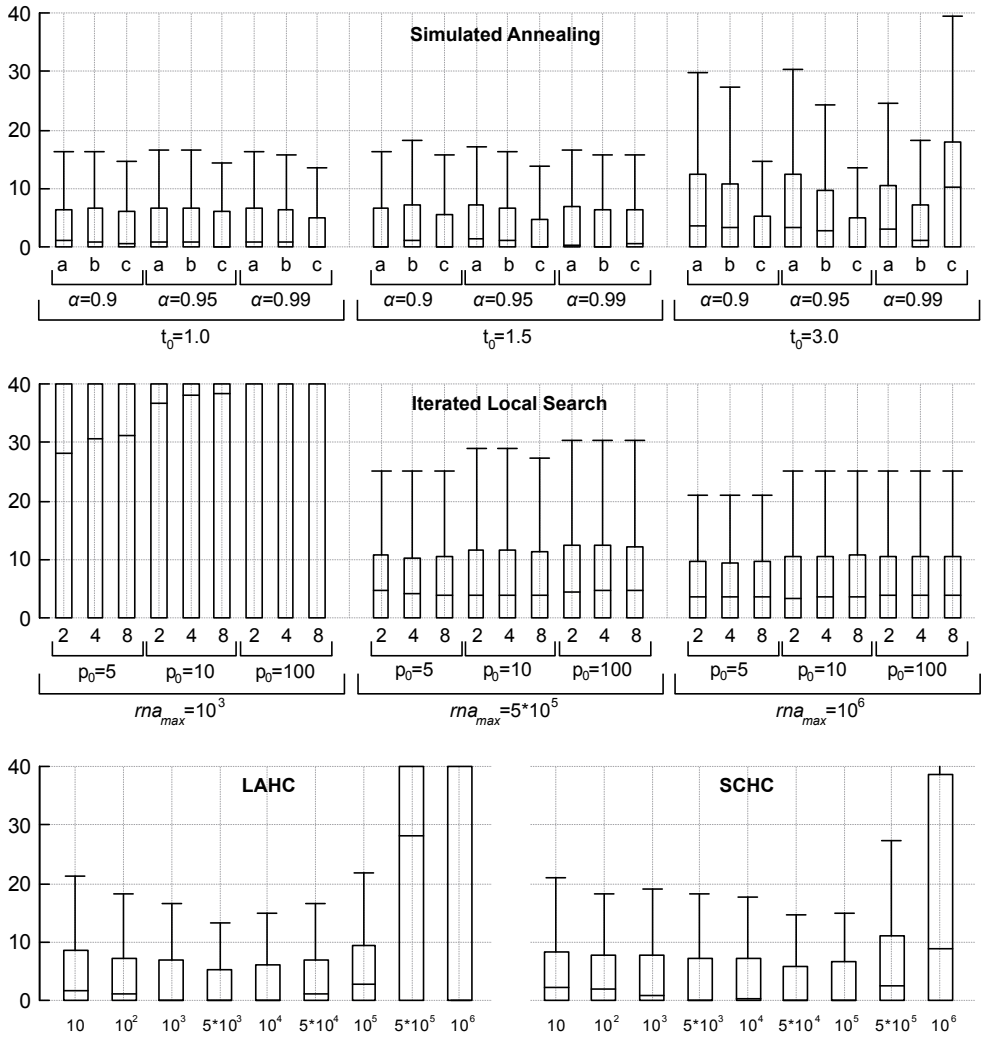

Figure 7: Parameter influence on the manual tuning of each SLS method.

5.1.2 iRace tuning

Table 1 presents the best parameters indicated by iRace on a budget of 20,000 runs for each SLS method. The range considered for each parameter is also shown. These parameters were used in the final, complete tests, to produce the results displayed in Figure 8. The quality of the results obtained with the parameters recommended by iRace are comparable to the best ones obtained manually (Figure 7). The automated process used by iRace was both easier and faster than the iterative process of manual tuning.

SLS	Param.	Range	Value
SA	t_0	$1 \rightarrow 10^3$	1
	α	$0.90 \rightarrow 0.99$	0.96
	sa_{max}	$100 \rightarrow 10^7$	1176628
LAHC	l	$10 \rightarrow 10^6$	135
SCHC	c	$10 \rightarrow 10^6$	810

SLS	Param.	Range	Value
ILS	p_0	$1 \rightarrow 100$	1
	p_{max}	$1 \rightarrow 10^3$	54
	rna_{max}	$10^5 \rightarrow 10^7$	8013591
	$iter_{max}$	$1 \rightarrow 10^4$	8642

Table 1: Parameters indicated by iRace for SA, ILS, LAHC and SCHC.

5.2 Neighborhood structures tuning

iRace was also used to infer the impact of the six neighborhood structures considering each of the four neighbor generation strategies (Section 3). The activation (or deactivation) of each neighborhood was set as a parameter of the algorithm. In total 24 parameters were added to the solver, one for each neighborhood with a specific strategy. iRace was executed to select the best parameter setting – or neighborhood selection – for each SLS method. A budget of 30,000 runs and the training set of 164 instances (Section 5.1) were considered.

iRace produced a pool with the 20 best configurations found. Results in Table 2 indicate how often each neighborhood was activated in these various configurations. The rows represent the neighbor generation strategies and the columns represent the neighborhoods. Each cell shows, among the 20 best configurations of all SLS methods, the percentage of times a neighborhood is active with a particular strategy.

Strategy	Shift	Switch	Task move	Swap	2-shift	Direct swap
Regular policy and makespan machines	60.0%	25.0%	60.0%	60.0%	5.0%	45.0%
Regular policy and random machine	45.0%	70.0%	5.0%	70.0%	20.0%	45.0%
Intensification policy and makespan machine	30.0%	20.0%	90.0%	100.0%	35.0%	45.0%
Intensification policy and random machine	95.0%	5.0%	100.0%	70.0%	10.0%	35.0%

Table 2: Neighborhoods selection by iRace.

Table 2 enables one to conclude that neighborhoods *2-shift* and *direct swap* have the most negligible positive impact since they were rejected by most configurations suggested by iRace. Nevertheless, all combinations of neighborhoods and strategies were selected at least once. For simplicity, all neighborhoods are activated in the final algorithm.

5.3 Results and discussion

Once the best parameters for each method were defined (Table 1), based on the training set of instances, the proposed SLS methods were evaluated considering the full set of 1,640 instances. Each method ran five independent executions (with different random seeds) on each instance.

Table 3 reports the average RPD produced by different algorithms in the time limit stipulated by Vallada and Ruiz (2011) ($t = 50$), scaled to the computational power of the author's computer, as detailed in Section 5. The four groups of instances have varying setup time ranges and 10 different instances for each combination of size and setup time range. Therefore each average value was computed considering $5 \times 10 \times 4 = 200$ executions.

Column `AIRP` indicates the algorithm of Cota et al. (2014), which was also implemented in Java. Given that the AIRP solver was obtained, this solver executed not only in the same hardware, but also the same Java Virtual Machine software. Column `Vallada` presents the RPDs computed by the best algorithm in Vallada and Ruiz (2011), scaled considering the updated best known solutions. The instances are grouped by size, with each row representing 40 instances.

Table 3 indicates how solvers `AIRP` and `Vallada` produce solutions with increasingly large RPDs for large instances. Vallada and Ruiz (2011) also ran additional experiments with more restricted execution times than considered in this experiment (multipliers $t = 10, 20$). The updated best known solution shows that these restricted times were excessively short for their algorithm to produce reasonable results, since even with the larger times their RPD remains high. The updated upper bounds in the benchmark instances explicitly stress the hardness of these problems, showing that even with the improvements achieved all the heuristics studied still have significant room for improvement.

For small instances, both `AIRP` and LAHC produced the best results. LAHC, however, is more robust than AIRP which generates, for some instance sizes, RPDs of more than 10%. All proposed SLS methods perform significantly better than the other algorithms for the remaining instances. The SA's results are highlighted and demonstrate how it consistently produced better average RPDs for all large instances. In addition, the best known solution was improved for 901 of the 1,000 large instances.

A comparison of the RPDs produced by each algorithm in the entire set of instances is presented in Figure 8. Each boxplot aggregates 8,200 executions. The four implemented SLS methods significantly outperform the other algorithms in terms of average solution quality.

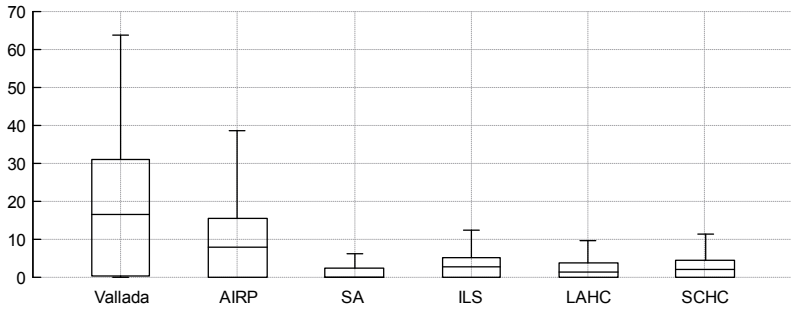

Figure 8: Boxplots with the RPDs produced by each algorithm for all 1,640 instances.

Table 3: Average RPDs produced by each algorithm for different instance sizes.

n	k	Vallada	AIRP	SA	ILS	LAHC	SCHC
6	2	**0.00**	0.87	0.86	2.04	**0.00**	**0.00**
	3	0.08	0.08	1.28	2.08	**0.00**	0.01
	4	0.27	3.80	0.75	2.25	**0.04**	0.05
	5	0.21	10.25	0.32	1.33	**0.00**	**0.00**
8	2	0.03	**0.00**	2.01	2.76	0.04	0.13
	3	0.24	**0.00**	1.33	4.28	0.12	0.19
	4	0.39	0.48	1.16	2.93	**0.10**	0.17
	5	0.20	0.38	0.12	1.64	**0.00**	0.55
10	2	0.17	**0.00**	1.35	3.23	0.05	0.33
	3	0.20	**0.00**	1.21	2.93	0.09	0.44
	4	0.32	**0.13**	1.18	3.83	0.30	0.96
	5	1.15	**0.27**	0.56	3.33	1.05	1.41
12	2	**0.09**	0.11	1.54	2.63	0.15	0.72
	3	0.08	**0.03**	1.52	3.94	0.22	0.95
	4	0.75	**0.12**	0.99	3.62	0.51	1.67
	5	1.67	**0.63**	1.28	5.10	1.66	2.53
50	10	12.42	5.97	**2.38**	5.37	6.43	6.87
	15	20.83	7.10	**1.58**	3.14	5.33	5.32
	20	25.65	9.30	**0.94**	1.96	3.90	4.26
	25	30.21	11.54	**1.26**	1.82	4.14	4.52
	30	32.83	12.80	**1.63**	1.74	3.24	3.61
100	10	13.96	8.79	**2.06**	3.81	3.92	5.20
	15	22.41	10.78	**2.15**	3.48	5.34	5.40
	20	29.63	13.42	**2.15**	3.51	5.33	5.58
	25	36.39	15.40	**1.97**	2.97	4.22	4.82
	30	41.64	19.22	**2.13**	3.52	4.20	3.87
150	10	15.81	10.03	**1.78**	3.77	3.08	4.39
	15	22.83	12.57	**1.47**	3.46	3.67	4.37
	20	30.62	14.78	**2.33**	3.68	4.09	4.42
	25	37.45	17.19	**2.19**	3.78	3.84	3.78
	30	43.27	20.06	**2.70**	4.49	3.80	4.15
200	10	16.14	10.47	**1.55**	3.39	2.92	3.98
	15	24.76	13.64	**1.78**	3.96	3.37	4.27
	20	31.89	15.89	**2.03**	4.18	3.49	3.92
	25	39.33	18.99	**2.42**	4.57	3.58	3.56
	30	44.81	21.08	**2.32**	5.11	3.24	3.36
250	10	17.14	10.52	**1.57**	3.44	2.67	3.64
	15	25.09	13.44	**1.72**	3.98	3.25	4.09
	20	33.38	16.18	**2.04**	4.45	3.45	3.93
	25	39.79	19.15	**2.37**	5.26	3.39	3.72
	30	44.95	21.29	**2.10**	5.74	2.81	3.32
	min	**0.00**	**0.00**	0.12	1.33	**0.00**	**0.00**
	avg	18.03	8.95	**1.61**	3.48	2.46	2.89
	max	44.95	21.29	**2.70**	5.74	6.43	6.87

6 Conclusions

This work detailed the design, computational experiments and performance analysis of four heuristic approaches to the unrelated parallel machine scheduling problem with sequence-dependent setup times. The proposed algorithms employ six different neighborhoods. The application of these neighborhoods and the algorithmic parame-

ters were extensively tuned, both applying a manual selection of parameters and the iRace package to automate the search for the best parameter configuration.

The proposed SLS algorithms consistently produced good results, outperforming two of the current best algorithms for the UPMSP when considering the solution quality produced in restricted computation times. The bounds of 901 out of 1000 large instances were improved over the previous best known solutions. All proposed algorithms have the advantage of being much simpler to implement than recently proposed hybrid heuristics.

After comparing the implemented metaheuristics, one very interesting observation became apparent. When tuning optimization algorithms, the method with fewer parameters is not necessarily the easiest to tune. For instance, Simulated Annealing (SA) has more parameters than Step Counting Hill Climbing (SCHC). However, extensive tuning revealed that many different parameter configurations for SA presented similarly good results. Despite the extensive set of experiments conducted, SCHC was unable to replicate SA's impressive results.

The results open up opportunities for future research regarding the development of fundamental heuristics to be applied withing SLS algorithms for scheduling and other combinatorial optimization problems. Given the unraveled correlation between the number of parameters and the robustness of SLS algorithms, it is worthwhile revising the application range of the algorithms belonging to this class.

Acknowledgements

We acknowledge FAPEMIG and CNPq for supporting the development of this research, as well as the Belgian Science Policy Office (BELSPO) in the Interuniversity Attraction Pole COMEX (http://comex.ulb.ac.be) and the Leuven Mobility Research Center. The computational resources and services used in this work were provided by the VSC (Flemish Supercomputer Center), funded by the Hercules Foundation and the Flemish Government – department EWI.

References

Allahverdi, A., Ng, C., Cheng, T., Kovalyov, M. Y., 2008. A survey of scheduling problems with setup times or costs. European Journal of Operational Research 187 (3), 985–1032.

Avalos-Rosales, O., Alvarez, A. M., Angel-Bello, F., 2013. A reformulation for the problem of scheduling unrelated parallel machines with sequence and machine dependent setup times. In: Twenty-Third International Conference on Automated Planning and Scheduling. pp. 278–282.

Avalos-Rosales, O., Angel-Bello, F., Alvarez, A., 2015. Efficient metaheuristic algorithm and re-formulations for the unrelated parallel machine scheduling problem with sequence and machine-dependent setup times. The International Journal of Advanced Manufacturing Technology 76 (9-12), 1705–1718.

Birattari, M., Balaprakash, P., Dorigo, M., 2007. The ACO/F-race algorithm for combinatorial optimization under uncertainty. In: Doerner, K., Gendreau, M., Greistorfer, P., Gutjahr, W., R.F., H., Reimann, M. (Eds.), Metaheuristics–Progress in Complex Systems Optimization. Operations Research/Computer Science Interfaces Series. Springer, Berlin, Germany, pp. 189–203.

Burke, E. K., Bykov, Y., 2008. A Late Acceptance Strategy in Hill-Climbing for Exam Timetabling Problems. In: PATAT '08 Proceedings of the 7th International Conference on the Practice and Theory of Automated Timetabling.

Burke, E. K., Bykov, Y., 2012. The Late Acceptance Hill-Climbing Heuristic. Tech. Rep. CSM-192, Department of Computing Science and Mathematics, University of Stirling.

Bykov, Y., Petrovic, S., 2013. An initial study of a novel Step Counting Hill Climbing heuristic applied to timetabling problems. In: Kendall, G., Berghe, G. V., McCollum, B. (Eds.), MISTA 2013: Proceedings of the 6th Multidisciplinary International Conference on Scheduling: Theory and Applications. Gent, Belgium, pp. 691–693.

Cota, L., Haddad, M., Souza, M., Coelho, V., 2014. AIRP: A heuristic algorithm for solving the unrelated parallel machine scheduling problem. In: Evolutionary Computation (CEC), 2014 IEEE Congress on. IEEE, pp. 1855–1862.

De Paula, M. R., Ravetti, M. G., Mateus, G. R., Pardalos, P. M., 2007. Solving parallel machines scheduling problems with sequence-dependent setup times using variable neighbourhood search. IMA Journal Management Mathematics 18 (2), 101–115.

Dueck, G., Scheuer, T., 1990. Threshold accepting: a general purpose optimization algorithm appearing superior to simulated annealing. Journal of computational physics 90 (1), 161–175.

Feo, T. A., Resende, M. G. C., 1995. Greedy randomized adaptive search procedures. In: Journal of Global Optimization. Vol. 6. Springer, pp. 109–133.

Garey, M. R., Johnson, D. S., 1979. Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Company.

Glover, F., Laguna, M., Martí, R., 2000. Fundamentals of scatter search and path relinking. Control and cybernetics 29 (3), 653–684.

Goldberg, D. E., 1989. Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley. Berkeley.

Graham, R., Lawler, E., Lenstra, J., Rinnooy Kan, A., 1979. Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey. Annals of Discrete Mathematics 5 (2), 287–326.

Hansen, P., Mladenović, N., 1997. Variable Neighborhood Search. Computers and Operations Research 24 (11), 1097–1100.

Henderson, D., Jacobson, S. H., Johnson, A. W., 2003. The theory and practice of simulated annealing. In: Glover, F., Kochenberger, G. (Eds.), Handbook of Metaheuristics. International Series in Operations Research & Management Science. Kluwer Academic Publishers, pp. 287–319.

Hoos, H. H., Stützle, T., 2005. Stochastic Local Search: Foundations and Applications. Morgan Kaufmann, Ch. Empirical Analysis of SLS Algorithms, pp. 149–201.

Kendall, G., Bai, R., Błazewicz, J., De Causmaecker, P., Gendreau, M., John, R., Li, J., McCollum, B., Pesch, E., Qu, R., Sabar, N., Vanden Berghe, G., Lee, A., 2016. Good laboratory practice for optimization research. Journal of the Operational Research Society 67 (4), 676–689.

Kim, D., Kim, K., Jang, W., Chen, F., 2002. Unrelated parallel machine scheduling with setup times using simulated annealing. Robotics and Computer-Integrated Manufacturing 18 (3-4), 223–231.

Kirkpatrick, S., Gelatt, C. D., Vecchi, M. P., 1983. Optimization by simulated annealing. Science 220 (4598), 671–680.

López-Ibánez, M., Dubois-Lacoste, J., Stützle, T., Birattari, M., 2011. The irace package: Iterated racing for automatic algorithm configuration. IRIDIA, Université Libre de Bruxelles, Belgium, Tech. Rep. TR/IRIDIA/2011-004.

López-Ibánez, M., Stützle, T., 2014. Automatically improving the anytime behaviour of optimisation algorithms. European Journal of Operational Research 235 (3), 569 – 582.

Lourenço, H. R., Martin, O. C., Stützle, T., 2003. Iterated local search. In: Glover, F., Kochenberger, G. (Eds.), Handbook of Metaheuristics. Kluwer Academic Publishers, Boston, Ch. 11.

McConnell, S., 2004. Code complete. Pearson Education.

PassMark software, 2015. CPU benchmarks.
URL `https://www.cpubenchmark.net/`

R Development Core Team, 2008. R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria.
URL `http://www.R-project.org`

Rabadi, G., Moraga, R., Ameer, A., 2006. Heuristics for the unrelated parallel machine scheduling problem with setup times. Journal of Intelligent Manufacturing 17, 85–97.

SOA-ITI, Nov. 2013. Grupo de investigación SOA: Sistemas de optimización aplicada.
URL `http://soa.iti.es/`

Vallada, E., Ruiz, R., 2011. A genetic algorithm for the unrelated parallel machine scheduling problem with sequence dependent setup times. European Journal of Operational Research 211 (3), 612–622.