Solving the Unrelated Parallel Machine Scheduling Problem with Setup Times by Efficient Algorithms based on Iterated Local Search

Matheus N. Haddad[1], Luciano P. Cota[1], Marcone J. F. Souza[1], and Nelson Maculan[2]

[1] Universidade Federal de Ouro Preto
Departamento de Computação
35.400-000, Ouro Preto – MG, Brazil
{mathaddad,lucianoufop}@gmail.com,marcone@iceb.ufop.br
[2] Universidade Federal do Rio de Janeiro
Programa de Engenharia de Sistemas e Computação
21941-972, Rio de Janeiro – RJ, Brazil
maculan@cos.ufrj.br

Abstract. The *Unrelated Parallel Machine Scheduling Problem with Setup Times* (UPMSPST) is a problem that belongs to the $\mathcal{NP}$-Hard class and it is frequently found in many practical situations, like in textile and chemical industries. The objective in UPMSPST is to schedule jobs in machines in order to achieve the maximum completion time, known as *makespan*. In an attempt to solve this problem, it is proposed two algorithms: the AIV and the HIVP. Both algorithms are based on *Iterated Local Search* (ILS) and *Variable Neighborhood Descent* (VND). The difference between AIV and HIVP is that the first one generates a greedy initial solution, while the second applies a partially greedy procedure to construct the initial solution and it includes the *Path Relinking* (PR) technique. Neighborhoods based on swaps and multiple insertions are investigated in the developed algorithms. AIV and HIVP were tested on benchmark test problems from literature and statistical analysis of the computational results showed the superiority of them, outperforming the previously best known solutions for UPMSPST.

Key words: Unrelated Parallel Machine Scheduling, Iterated Local Search, Random Variable Neighborhood Descent, Makespan

1 Introduction

This paper deals with the *Unrelated Parallel Machine Scheduling Problem with Setup Times* (UPMSPST), which can be formally defined as follows. Let $N = \{1, ..., n\}$ be a set of jobs and let $M = \{1, ..., m\}$ be a set of unrelated machines. The UPMSPST consists of scheduling n jobs on m machines, satisfying the following characteristics: (i) Each job $j \in N$ must be processed exactly once by

only one machine $k \in M$. (ii) Each job $j \in N$ has a processing time p_{jk} which depends on the machine $k \in M$ where it will be allocated. (iii) There are setup times S_{ijk}, between jobs, where k represents the machine on which jobs i and j are processed, in this order. (iv) There is a setup time to process the first job, represented by S_{0jk}. The objective is to minimize the maximum completion time of the schedule, the so-called *makespan* or also denoted by $C_{\max}$. Because of such characteristics, UPMSPST is defined as $R_M \mid S_{ijk} \mid C_{\max}$ [1]. In this representation, R_M represents the unrelated machines, S_{ijk} the setup times and $C_{\max}$ the *makespan*.

Figure 1 illustrates a schedule for a test problem composed by two machines and seven jobs. In Table 1 are presented the processing times of these jobs in both machines. The setup times of these jobs in these machines are showed in Table 2.

It can be observed that in machine M1 the jobs 2, 1 and 7 are allocated in this order. In machine M2 the schedule of the jobs 5, 4, 6 and 3, in this order, is also perceived by this figure. The cross-hatched areas of the figure represent the setup times between jobs and the numbered areas the processing times. On the line below the schedule there is the timeline, in which the times 120 and 130 represent the completion times of each machine.

Table 1. Processing times in machines M1 and M2.

	M1	M2
1	20	4
2	25	21
3	28	14
4	17	32
5	43	38
6	9	23
7	58	52

Table 2. Setup times in machines M1 and M2

M1	1	2	3	4	5	6	7	M2	1	2	3	4	5	6	7
1	0	1	8	1	3	9	6	1	0	4	6	5	10	3	2
2	4	0	7	3	7	8	4	2	1	0	6	2	7	7	5
3	7	3	0	2	3	5	3	3	2	6	0	6	8	1	4
4	3	8	3	0	5	2	2	4	5	7	1	0	12	10	6
5	8	3	7	9	0	5	7	5	7	9	5	7	0	4	8
6	8	8	1	2	2	0	9	6	9	3	5	4	9	0	3
7	1	4	5	2	3	5	0	7	3	2	6	1	5	6	0

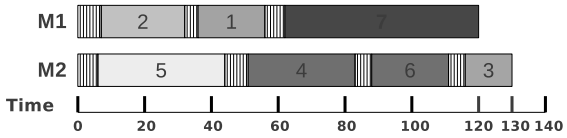

Fig. 1. An example of a possible schedule.

As the job 6 is allocated to machine M2 its processing time p_{62} will be 23. Its predecessor and its successor are the jobs 4 and 3, respectively. So, in this example, are computed the times $S_{462} = 5$ and $S_{632} = 5$. Thus, it can be calculated the completion time of machine M1 as $S_{021} + p_{21} + S_{211} + p_{11} + S_{171} + p_{71} = 120$. Equivalently it is also calculated the completion time of machine M2 as $S_{052} + p_{52} + S_{542} + p_{42} + S_{462} + p_{62} + S_{632} + p_{32} = 130$. After the calculation of the completion times of machines M1 and M2, it can be concluded that the machine M2 is the bottleneck machine. In other words, M2 is the machine that has the highest completion time, the *makespan*.

The UPMSPST appears in many practical situations, like in textile and chemical industries [2]. On the other hand, the UPMSPST is in $\mathcal{NP}$-Hard class, as it is a generalization of the *Parallel Machine Scheduling Problem with Identical Machines and without Setup Times* [3, 4]. The theoretical and practical importance instigate the study of the UPMSPST. Under these circumstances, finding the optimal solution for UPMSPST using exact methods can be computationally infeasible for large-sized problems. Thus, metaheuristics and local search heuristics are usually developed to find good near optimal solutions.

In order to find these near optimal solutions for the UPMSPST, this paper proposes two algorithms based on *Iterated Local Search* – ILS [5] and *Variable Neighborhood Descent* – VND [6]. The first algorithm is called AIV, it starts from an initial solution constructed on a greedy way by the *Adaptive Shortest Processing Time* – ASPT rule. Then, this initial solution is refined by ILS, using as local search the Random VND procedure. In this procedure, here called RVND, there is no fixed sequence of neighborhoods, because they are sorted on each application of the local search. In [7] the authors showed the effectiveness of RVND over the conventional VND. The second algorithm named HIVP is an endeavour to upgrade AIV, constructing the solution by a partially greedy procedure and including the *Path Relinking* – PR technique.

Both algorithms were tested using benchmark instances from [8] and the computational results showed that they are able to produce better solutions than the ones found in literature, with lower variability and setting new upper bounds for the majority of instances.

The rest of this paper is structured as follows. Firstly, works that inspired the development of this paper are described. Then, the methodology used for the deployment of this paper is presented. The computational results are shown on sequence. Finally, this paper is concluded and possible proposals to be explored are described.

2 Literature Review

In literature are found several works that seek to address the UPMSPST and similar problems. These approaches were inspirations for the development of this paper.

[9] propose the development of seven heuristics with the objective of minimizing the weighted mean completion time. In [10], a problem with common due dates is addressed and four heuristics are implemented for minimizing the total weighted tardiness. [11] aim to minimize the total weighted tardiness, considering dynamic releases of jobs and dynamic availability of machines and they used four dispatching rules in order to generate initial solutions and a *Tabu Search* as the basis for the development of six search algorithms. This problem is also addressed in [2], where a *Branch-and-Price* algorithm is developed.

More recent references are found when dealing with the UPMSPST. [12] created a *Three-phase Partitioning Heuristic*, called PH. In [13] it is proposed a *Metaheuristic for Randomized Priority Search* (Meta-RaPS). [14] bet in Tabu Search for solving the UPMSPST. [15] implement the *Ant Colony Optimization* (ACO), considering its application to problems wherein the ratio of jobs to machines is large. In [16] it is implemented a *Restricted Simulated Annealing* (RSA), which aims to reduce the computational effort by only performing movements that the algorithm consider effective. In [17] is defined and proved a set of proprieties for the UPMSPST and also implemented an *Genetic Algorithm* and a *Simulated Annealing* using these proprieties. A hybridization that joins the *Multistart* algorithm, the VND and a mathematical programming model is made in [18]. [19] solve the UPMSPST using *Genetic Algorithms*, with two sets of parameters, yielding two algorithms, GA1 and GA2. In [19], the authors created and provided test problems for the UPMSPST [8]. Also in [8] are presented the best known solutions to the UPMSPST so far.

3 Methodology

3.1 Representation and Evaluation of a Solution

To represent a solution s it is used a vector v of m positions, with each position representing a machine. In addition, each machine is associated with a list of jobs allocated to it. The order of the jobs in this list represents the sequence of execution.

The evaluation of a solution s is done by calculating the processing time of the machine that will be the last to conclude the execution of its jobs. In other words, the evaluation value is the *makespan*.

3.2 Proposed Algorithms

In this section two algorithms, AIV and HIVP, are proposed for solving UPMSPST. The details of operation of these algorithms are described below.

The AIV Algorithm The first proposed algorithm, named AIV, combines the heuristic procedures *Iterated Local Search* (ILS) and *Random Variable Neighborhood Descent* (RVND). The main structure of AIV is based on ILS, using the RVND procedure to perform the local searches.

The pseudo-code of AIV is presented in Algorithm 1.

Algorithm 1: AIV

```
   input : timesLevel, executionTime
 1 currentTime ← 0;
 2 Solution s, s', bestSol;
 3 s ← ASPT() ;                                          /* see subsection 3.3 */
 4 s ← RVND(s) ;                                         /* see subsection 3.6 */
 5 bestSol ← s;
 6 level ← 1;
 7 Update currentTime;
 8 while currentTime < executionTime do
 9 |   s' ← s;
10 |   times ← 0;
11 |   maxPerturb ← level + 1;
12 |   while times < timesLevel do
13 |   |   perturb ← 0;
14 |   |   s' ← s;
15 |   |   while perturb < maxPerturb do
16 |   |   |   perturb ++;
17 |   |   |   s' ← perturbation(s') ;                   /* see subsection 3.8 */
18 |   |   end
19 |   |   s' ← RVND(s') ;                               /* see subsection 3.6 */
20 |   |   if f(s') < f(s) then
21 |   |   |   s ← s';
22 |   |   |   updateBest(s, bestSol);
23 |   |   |   times ← 0;
24 |   |   end
25 |   |   times ++;
26 |   |   Update currentTime;
27 |   end
28 |   level ++;
29 |   if level ≥ 4 then
30 |   |   level ← 1;
31 |   end
32 end
33 return bestSol;
```

The Algorithm 1 has only two input parameters: 1) *timesLevel*, which represents the number of times in each level of perturbation; 2) *executionTime*, the time in milliseconds that limits the execution of the algorithm.

First of all, AIV begins initializing the variable that controls the time limit, *currentTime* (line 1). Next, it initializes three empty solutions: the current solution s, the modified solution s' and the solution that will store the best solution found *bestSol* (line 2).

In line 3 a new solution is created based on the *Adaptive Shortest Processing Time* (ASPT) rule (see subsection 3.3). Then, this new solution passes through local searches at line 4, using the RVND module (see subsection 3.6).

In the next step, the current best known solution, *bestSol*, is updated (line 5) and the level of perturbations is set to 1 (line 6).

After all these steps, the execution time is recalculated in line 7.

The iterative process of ILS is situated in lines 8 to 32 and it finishes when the time limit is exceeded. A copy of the current solution to the modified solution is made in line 9.

In lines 10 and 11 the variable that controls the number of times in each level of perturbation (*times*) is initialized, as well as the variable that limits the maximum number of perturbations (*maxPerturb*). The following loop is responsible to control the number of times in each level of perturbation (lines 12-27).

The next loop, lines 15 to 18, executes the perturbations (line 17) in the modified solution (see subsection 3.8). The number of times this loop is executed depends on the level of perturbation. With the perturbations accomplished, the new solution obtained is evaluated and the RVND procedure is applied in this new solution until a local optimum is reached, in relation to all neighborhoods adopted in RVND.

In lines (20-24) it is verified if the changes made in the current solution were good enough to continue the search from it. When the time is up, in *bestSol* will be stored the best solution found by AIV.

Each module of AIV will be detailed afterwards.

The HIVP Algorithm The second proposed algorithm, HIVP, is an attempt to improve the AIV algorithm, differing only by two procedures: the construction of the initial solution and the incorporation of an intensification/diversification technique, the Path Relinking – PR [20]. The Algorithm 2 presents the pseudo-code of HIVP.

Since HIVP is an upgrade of AIV, only the different lines will be described next. The elite set is initialized at line 3. At line 4 the solution s receives a solution generated by the partially greedy procedure CPG_{HBSS} (see subsection 3.4). Following, s passes through local searches using the RVND procedure (see subsection 3.6) and *bestSol* receives the resulting solution from RVND. This solution is inserted in the elite set (line 7). In the iterative loop of HIVP, after the application of the RVND in the modified solution (line 21) the elite set is updated and a random real number between 0 and 1 is generated (lines 22 and 23). If this number is less than or equal to 0.05 and the elite set is full (with 5 solutions) it is performed an intensification/diversification of the search using the PR technique - lines 23 to 27 (see subsection 3.7). The modified solution and a random solution from the elite set are the input parameters of PR. If the modified solution is considered an improvement (line 28), then the elite set is updated at line 32.

3.3 ASPT: The Adaptive Shortest Processing Time Rule

The *Adaptive Shortest Processing Time* (ASPT) rule is an extension of the *Shortest Processing Time* rule [21].

In ASPT, firstly, it is created a set $N = \{1, ..., n\}$ containing all jobs and a set $M = \{1, ..., m\}$ that contains all machines.

Algorithm 2: HIVP

```
   Input  : timesLevel, executionTime
   Output : bestSol
 1 currentTime ← 0;
 2 Solution s, s', bestSol;
 3 elite ← {};
 4 s ← CPG_HBSS() ;                                   /* see subsection 3.4 */
 5 s ← RVND(s) ;                                      /* see subsection 3.6 */
 6 bestSol ← s;
 7 elite ← elite ∪ {bestSol};
 8 level ← 1;
 9 Update currentTime;
10 while currentTime ≤ executionTime do
11 |   s' ← s;
12 |   times ← 0;
13 |   maxPerturb ← level + 1;
14 |   while times < timesLevel do
15 |   |   perturb ← 0;
16 |   |   s' ← s;
17 |   |   while perturb < maxPerturb do
18 |   |   |   perturb ++;
19 |   |   |   s' ← perturbation(s') ;                /* see subsection 3.8 */
20 |   |   end
21 |   |   s' ← RVND(s') ;                            /* see subsection 3.6 */
22 |   |   elite ← update(s');
23 |   |   pr ← random(0,1);
24 |   |   if pr ≤ 0.05 e |elite| ≥ 5 then
25 |   |   |   el ← random(1,5);
26 |   |   |   s' ← PR(elite [el], s') ;              /* see subsection 3.7 */
27 |   |   end
28 |   |   if f(s') < f(s) then
29 |   |   |   s ← s';
30 |   |   |   times ← 0;
31 |   |   |   updateBest(s, bestSol);
32 |   |   |   elite ← update(s);
33 |   |   end
34 |   |   times ++;
35 |   |   Update currentTime;
36 |   end
37 |   level ++;
38 |   if level ≥ 4 then
39 |   |   level ← 1;
40 |   end
41 end
42 return bestSol ;
```

From the set N, the jobs are classified according to an evaluation function g_k. This function is responsible to obtain the completion time of the machine k. Given a *Candidate List* (CL) of jobs, it is evaluated, based on the g_k function, the insertion of each of these jobs in all positions of all machines. The aim is to obtain in which position of what machine that the candidate job will produce the lowest completion time, that is, the $g_{\min}$.

If the machine with the lowest completion time has not allocated any job yet, its new completion time will be the sum of the processing time of the job to be inserted with the initial setup time for such job.

If this machine has some job, its new completion time will be the previous completion time plus the processing time of the job to be inserted and the setup times involved, if it has sequenced jobs before or after.

This allocation process ends when all jobs are assigned to some machine, thus producing a feasible solution, s. This solution is returned by the heuristic. The algorithm is said to be adaptive because the choice of a job to be inserted depends on the preexisting allocation.

3.4 CPG_{HBSS}: The Partially Greedy Procedure

The CPG_{HBSS} procedure is a partially greedy constructive method which also uses the evaluation function g_k to classify the jobs. The same Candidate List (CL) created in ASPT is generated here. Notwithstanding, instead of deterministically choosing the job that produces the $g_{\min}$, the CPG_{HBSS} procedure chooses a candidate in the CL based on the *Heuristic-Biased Stochastic Sampling* (HBSS) [22]. The jobs in the CL are chosen according to a probability arising from a *bias* function.

In Table 3 are presented the following *bias* functions: Random, Logarithmic, Polynomial of degree 2, Polynomial of degree 3, Polynomial of degree 4 and Exponential. The first column shows the index of the candidates. In each cell of the table there is the probability of the candidate i be chosen using the respective *bias* function. For example, if the *bias* function selected is the Exponential, the fifth element of the CL will have 1.2% chance of being chosen.

Table 3. Table containing *bias* functions. Source: [22]

	Rand	*Log*	*Linear*	$Poly^2$	$Poly^3$	$Poly^4$	Exp
1	0.033	0.109	0.250	0.620	0.832	0.924	0.632
2	0.033	0.069	0.125	0.155	0.104	0.058	0.233
3	0.033	0.055	0.083	0.069	0.031	0.011	0.086
4	0.033	0.047	0.063	0.039	0.013	0.004	0.031
5	0.033	0.042	0.050	0.025	0.007	0.001	0.012
6-30	0.033	0.678	0.429	0.092	0.013	0.002	0.006

The *bias* function chosen in this work was the Exponential, this choice was made based on empirical tests. With the *bias* function selected, it can be explained how the CPG_{HBSS} procedure constructs a solution.

Initially, as in ASPT, all the jobs and machines are stored in the sets N and M, respectively. Then, all jobs are inserted in the CL.

While there are jobs in the CL: firstly, jobs are classified according to the g_k function (see subsection 3.3), analyzing their inclusion at the last position of a machine and creating a list named *Rank*. *Rank* is composed by combinations of (j, k), with $j \in LC$ and $k \in M$, ordered according to the value calculated by the g_k function.

A probability according to the Exponential *bias* function values, as presented in Table 3, is associated to each pair in *Rank*. Secondly, the Wheel Selection method is performed, in order to select which job will be inserted in the partial solution. Then, the selected job is removed from LC and all pairs containing this

job are removed from *Rank*. This loop is executed until all jobs are inserted in the solution, generating a feasible solution.

3.5 Neighborhood Structures

Three neighborhood structures are used to explore the solution space. These structures are based on swap and multiple insertion movements of the jobs.

Multiple Insertion The Multiple Insertion (MI) movement consists in reallocating a job from a position and inserting it on any position of the entire schedule, including the machine that it was previously allocated. The MI movement belongs to the $N^{MI}(.)$ neighborhood. To exemplify this movement, Figure 2 illustrates the transfer of the job 4 from the second position of machine $M2$ to the second position of machine $M1$.

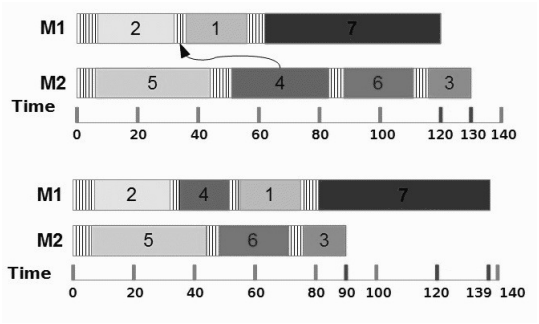

Fig. 2. Example of a Multiple Insertion movement

Swap in the Same Machine The Swap in the Same Machine (SSM) movement, as the name suggests, is done by swapping the positions of two jobs presented in the same machine. The SSM movement belongs to the $N^{SSM}(.)$ neighborhood. Figure 3 shows the swap of jobs 5 and 6 in machine $M2$.

Swap Between Different Machines The Swap Between Different Machines (SDM), which belongs to the $N^{SDM}(.)$ neighborhood, consists in swapping two jobs that are allocated in different machines. This movement can be better exemplified by Figure 4, where the swap of the jobs 7 ($M1$) and 5 ($M2$) is perceived.

3.6 RVND: The Random Variable Neighborhood Descent Procedure

The *Random Variable Neighborhood Descent* – RVND procedure [7, 23] is a variant of the VND procedure [6].

Each neighborhood of the set $\{N^{MI}, N^{SSM}, N^{SDM}\}$ described in section 3.5 defines one local search. Unlike VND, the RVND explores the solution space

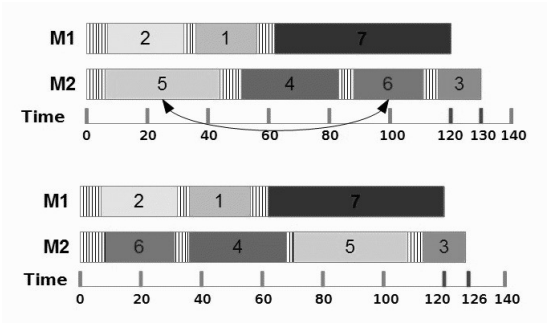

Fig. 3. Example of a Swap in the Same Machine movement

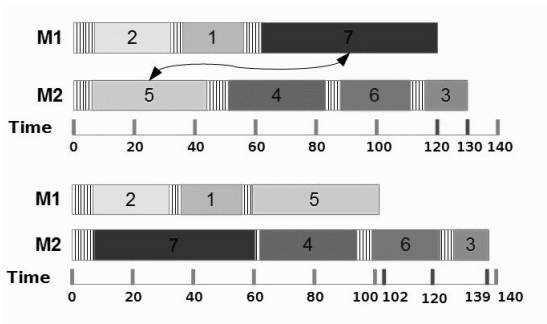

Fig. 4. Example of a Swap Between Different Machines movement

using these three neighborhoods in a random order each time the procedure is triggered. The RVND is finished when it is found on a local optimum with relation to the three considered neighborhoods.

Following are described the local searches procedures used in RVND.

***FI*$^{1}_{MI}$: Local Search with Multiple Insertion** The first local search uses multiple insertions movements ($N^{MI}(.)$ neighborhood) with the *First Improvement* strategy. In this search, each job of each machine is inserted in all positions of all machines.

The selection of the jobs to be removed respects the allocation order in the machines. That is, initially, the first job is selected to be removed, then the second job until all jobs from a machine are chosen. The machines that will have their jobs removed are selected based on their completion times. The search starts with machines with higher completion times to machines with lower completion times.

By contrast, the insertions are made from machines with lower completion times to machines with higher completion times. The jobs are inserted starting from the first position and stopping at the last position.

The movement is accepted if the completion times of the machines involved are reduced. If the completion time of a machine is reduced and the completion time of another machine is added, the movement is also accepted. However, in this case, it is only accepted if the value of reduced time is greater than the value of time increased.

It is noteworthy that even in the absence of improvement in the value of *makespan*, the movement can be accepted. Upon such acceptance of a movement, the search is restarted and only ends when it is found a local optimum, that is, when there is no movement that can be accepted in the neighborhood of multiple insertion.

FI_{SMD}: Local Search with Swaps Between Different Machines The second local search makes swap movements between different machines, exploring the $N^{SMD}(.)$ neighborhood. For each pair of existing machines every possible swap of jobs between them are analyzed.

Exchanges are made from machines that have higher completion times to machines with lower completion times. The acceptance criteria are the same as those applied in the first local search. If there are reductions in completion times on two machines involved, then the movement is accepted. If the reduced value of the completion time of a machine is larger than the completion time plus another machine, the movement is also accepted. Once a movement is accepted, the search stops.

BI_{SSM}: Local Search with Swaps on the Same Machine The third local search examines the $N^{SSM}(.)$ neighborhood and uses the strategy *Best Improvement*.

The machines are ordered from the machine that has the highest value of completion time to the machine that has the lowest value of completion time.

For each machine, starting from the first, all possible swaps involving pairs of jobs are investigated. The best movement is accepted if the completion time of the machine is reduced and, in this case, the local search is repeated from this solution; otherwise, the next machine is analyzed.

This local search only ends when no improvements is found in 30% of the machines.

3.7 Path Relinking

The Path Relinking – PR [20] technique makes a balance between intensification and diversification of the search. Its objective is to explore existing paths between high quality solutions. These high quality solutions are stored in an elite set.

In order to a solution s' enter in this elite set, one of the following conditions must be satisfied: i) be better than the best solution of the elite set, in terms of the *makespan* value; ii) be better (lower *makespan*) than the worst solution of the elite set and differentiate itself from all solutions of the elite set at least by 10%. The diversity criterion, when comparing two solutions, is the percentage of jobs allocated in different positions.

The *Backward Path Relinking* (BkPR) strategy is used. According [20] BkPR usually outperforms forward path relinking. Thus, a path is constructed from a base solution to a guide solution, being the best solution as the base solution and the worst solution as the guide solution. In this work, this strategy is applied over the following solutions: 1) a solution chosen randomly in the elite set; 2) the solution returned by the RVND local searches.

The attribute chosen for building the path is the position of a job. Initially, the jobs positions of the guide solution are inserted in a list named Φ. In each iteration is analyzed the insertion, in the base solution, of an attribute that belongs to the guide solution. Following, the other copy of this job is removed from the base solution. Moreover, if the machine that receives this job has another different job at the same position, then this job is relocated to another position that has not set its attribute from the guide solution yet.

With all attributes from guide solution analyzed, it is included to base solution the attribute which produces the lower cost. This cost is given by the sum of all completion times of all machines in base solution. After the insertion of an attribute, to this new base solution is applied the FI^2_{MI} local search, defined next. It is important to highlight that once an attribute is inserted in base solution, this attribute can not be changed.

Following, the selected attribute is removed from Φ. These steps are repeated until Φ is empty. At the end of the algorithm the base solution will have the same scheduling as the guide solution and the best solution found during this procedure is returned.

FI^2_{MI}: Local Search with Multiple Insertion As the local search FI^1_{MI}, the local search FI^2_{MI} also explores the $N^{MI}(.)$ neighborhood with the *First Improvement* strategy. But it differs by two characteristics: i) the only acceptance criterion is the improvement of the *makespan*; ii) when an improvement occurs, the method is stopped and the new solution is returned.

3.8 Perturbations

A perturbation is characterized by applying an insertion movement in a local optimum, but this movement differs when inserting the job in another machine. The job will be inserted into its best position, that is, in the position that will produce the lowest completion time. Doing so, sub parts of the problem are optimized after each perturbation. The machines and the job involved are chosen randomly.

In both AIV and HIVP, the number of perturbations applied to a solution is controlled by the level of perturbation. A level l of perturbation consists in the application of $l + 1$ insertion movements. The maximum level allowed for the perturbations is set to 3.

If *timeslevel* perturbed solutions are generated without an improvement in the current solution the perturbation level is increased. If an improvement of the current solution is found, the level of perturbation is set to its lowest level ($l = 1$).

3.9 Efficient Evaluation of the Objective Function

The evaluation of an entire solution after every movement, multiple insertion or swap, demands a large computational effort. Aiming to avoid this situation, it was created a procedure that evaluates only the processing and setup times involved in the movements. In this way, in order to obtain the new completion time of each machine it is necessary few additions and subtractions.

Taking the example of the multiple insertion movement illustrated in Figure 2, the new completion time of machine M2 is obtained by subtracting from its previous value the processing time of job 4 p_{42} and also subtracting the setup times involved, S_{542} and S_{462}. The setup time S_{562} also needs to be added to the completion time of machine M2. In machine M1, the processing time of job 4 p_{41} and the setup times S_{241} and S_{411} are included in the new completion time. Then, the new completion time of machine M1 is $M1 = 120-4+3+17+3 = 139$ and the new completion time of machine M2 is $M2 = 130-7-32-5+4 = 90$. Although the given example is for the multiple insertion movement, it is trivial to apply the same procedure for a swap movement.

4 Computational Results

Using a set of 360 test problems from [8] the computational tests were performed. This set of test problems involves combinations of 50, 100 and 150 jobs with 10, 15 and 20 machines. There are 40 instances for each combination of jobs and machines. The best known solutions for each of these test problems are also provided in [8].

AIV was developed in C++ language and HIVP was developed in Java language. All experiments were executed in a computer with *Intel Core i5 3.0 GHz* processor, 8 GB of RAM memory and in *Ubuntu 12.04* operational system.

The parameters used in both AIV and HIVP are: i) the number of iterations on each level of perturbation: $timeslevel = 15$; ii) the stop criterion: $Time_{\max}$, which is the maximum time of execution, in milliseconds, obtained by Eq. 1. In this equation, m represents the number of machines, n the number of jobs and t is a parameter that was tested with three values for each instance: 10, 30 and 50. It is observed that the stop criterion, with these values of t, was the same adopted in [19].

$$Time_{\max} = n \times (m/2) \times t \;\; ms \tag{1}$$

With the objective to verify the variability of final solutions produced by the algorithms it was used the metric given by Eq. 2. This metric is used to compare algorithms. For each algorithm Alg applied to a test problem i is calculated the *Relative Percentage Deviation* RPD_i of the solution found $\bar{f}_i^{Alg}$ in relation to the best known solution f_i^*.

In this paper, the algorithms AIV and HIVP were executed 30 times, for each instance and for each value of t, calculating the *Average Relative Percentage Deviation* RPD_i^{avg} of the RPD_i values found. In [19] the algorithms were executed 5 times for each instance and for each value of t.

$$RPD_i = \frac{\bar{f}_i^{Alg} - f_i^*}{f_i^*} \tag{2}$$

In Table 4 are presented, for each set of instances, the RPD_i^{avg} values obtained for each value of $t = 10, 30, 50$ by AIV and HIVP, and also it contains the RPD_i^{avg} values obtained by GA2, a genetic algorithm developed in [19]. To our knowledge, the results reported in the literature for this set of test problems are only presented in [19] and the best algorithm tested by the authors was GA2.

There are three values of RPD_i^{avg} separated by a '/' for each set of instances in the table. Each separation represents test results with different values of t, 10/30/50. If a negative value is found, it means that the reached result outperformed the best value found in [19] on their experiments.

Table 4. Average Relative Percentage Deviation of the algorithms AIV, HIVP and GA2 with $t = 10/30/50$.

Set of Instances	AIV[1]	HIVP[1]	GA2[2]
50 x 10	3.69/1.83/1.30	**2.27/0.25/-0.44**	7.79/6.92/6.49
50 x 15	1.52/-0.77/-1.33	**-0.6/-2.78/-3.47**	12.25/8.92/9.20
50 x 20	5.26/2.01/1.65	**-2.14/-4.06/-4.68**	11.08/8.04/9.57
100 x 10	5.06/2.93/2.00	**4.45/2.04/1.23**	15.72/6.76/5.54
100 x 15	**1.80**/-0.40/-1.29	2.17/**-0.64/-1.78**	22.15/8.36/7.32
100 x 20	**0.52**/-1.64/-2.89	0.58/**-2.44/-3.92**	22.02/9.79/8.59
150 x 10	**3.77/1.99**/1.07	4.21/2.15/**0.98**	18.40/5.75/5.28
150 x 15	**1.83/-0.24/-1.04**	3.37/0.42/-0.44	24.89/8.09/6.80
150 x 20	**-1.04/-3.10/-4.00**	1.22/-2.19/-3.39	22.63/9.53/7.40
RPD^{avg}	2.49/0.29/-0.50	**1.72/-0.8/-1.77**	17.44/8.02/7.35

[1]Executed on Intel Core i5 3.0 GHz, 8 GB of RAM, 30 runs for each instance
[2]Executed on Intel Core 2 Duo 2.4 GHz, 2 GB of RAM, 5 runs for each instance

The best values of RPD^{avg} are highlighted in bold. It is remarkable that HIVP is the algorithm that found the best results. Not only it improved the majority of best known solutions, but also it won in 63% of the sets of instances. The algorithm AIV found the best results in 37% of the remainder sets of instances.

A table with all results found by both algorithms and also the previous best known values for the UPMSPST can be found in `http://www.decom.ufop.br/prof/marcone/projects/upmsp/Experiments_UPMSPST_AIV_HIVP.ods`.

The box plot, Figure 5, contains all RPD^{avg} values for each algorithm. It is notable that 100% of the RPD values encountered by both AIV and HIVP outperformed the ones obtained by GA2 algorithm. By the way, it is observed

that 75% of solutions found by the developed algorithms are near the best known solutions. Besides, more than 50% of solutions found by the HIVP algorithm are better than the best known so far. With AIV this percentage drops to 25%.

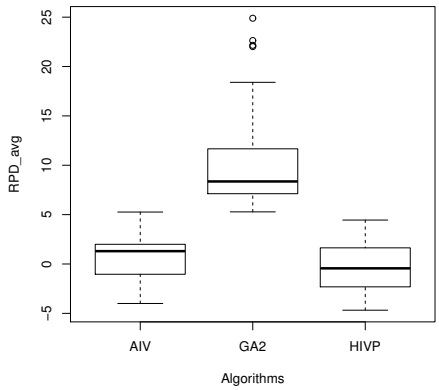

Fig. 5. Box plot showing the RPD^{avg} of the algorithms.

The results were submitted to the Shapiro-Wilk test [24] to verify if the sample satisfies the normality test, so that it can be decided which test to use for analyzing statistical differences between all algorithms. The Shapiro-Wilk returned, with significance level of 5%, $W = 0.9261$ and $p = 0.2692$. As $p = 0.2692 > 0.05$ then it can be concluded with 95% of confidence level that the sample are taken from a normal distribution.

Thus, in order to verify if exist statistical differences between the RPD values, it was applied an analysis of variance (ANOVA) [25]. This analysis returned, with 95% of confidence level and $threshold = 0.05$, that $p = 2 \times 10^{-16}$. As $p < threshold$, it is possible to ensure that exist statistical differences between the RPD values.

A Tukey HSD test, with 95% of confidence level and $threshold = 0.05$, was used for checking where are these differences. Table 5 contains the differences in the average values of RPD (diff), the lower end point (lwr), the upper end point (upr) and the *p-value* (p) for each pair of algorithms.

Table 5. Results from Tukey HSD test.

Algorithms	diff	lwr	upr	p
GA2-AIV	10.077407	7.495481	12.659333	0.0000000
HIVP-AIV	-1.041481	-3.623408	1.540445	0.6018344
HIVP-GA2	-11.118889	-13.700815	-8.536963	0.0000000

The *p-value* shows that when comparing AIV to GA2 there is a statistical difference between them, because it was less than the *threshold*. The same conclusion can be achieved when comparing HIVP to GA2. However, when AIV is compared to HIVP they are not statistically different from each other, since the *p-value* was greater than the *threshold*.

By plotting the results from the Tukey HSD test (Fig. 6) it is more noticeable that HIVP and AIV are statistically different from GA2, as their graphs do not pass through zero. Comparing algorithms HIVP and AIV it can be perceived that they are not statistically different from each other, because the graph passes through zero. Thus, with a statistical basis it can be concluded, within the considered instances, that AIV and HIVP are the best algorithms on obtaining solutions for UPMSPST.

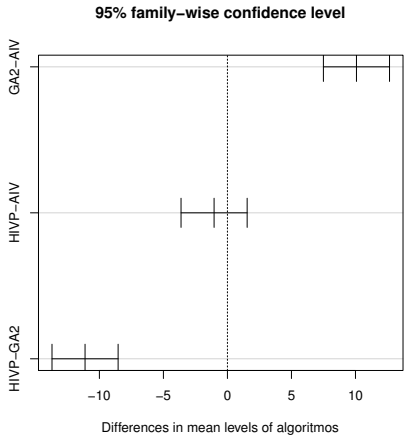

Fig. 6. Graphical results from Tukey HSD test.

5 Conclusions

This paper studied the *Unrelated Parallel Machine Scheduling Problem with Setup Times* (UPMSPST), aiming to the minimization of the maximum completion time of the schedule, the *makespan*.

In order to solve the UPMSPST it was proposed two algorithms based on *Iterated Local Search* (ILS) and *Variable Neighborhood Descent* (VND). The first algorithm was named AIV and it implements the *Adaptive Shortest Processing Time* (ASPT) rule to create an initial solution. The *Random Variable Neighborhood Descent* (RVND) procedure was used to perform the local searches, randomly exploring the solution space with insertions and swap movements. A perturbation in AIV is an application of an insertion movement. The second algorithm, called HIVP, is an attempt to upgrade AIV, constructing the solution

using a partially greedy procedure and incorporating the *Path Relinking* (PR) technique in order to intensify and diversify the search.

The two developed algorithms were applied to instances taken from literature and the results were compared the genetic algorithm GA2, developed in [19]. Statistical analysis of the computational results showed that AIV and HIVP are able to produce 100% of better solutions than GA2. HIVP and AIV were also able to generate new upper bounds for these test problems. Although HIVP seems to be better than AIV, statistically this was not proved. Nevertheless, it can be concluded that both AIV and HIVP are two efficient algorithms when dealing with the UPMSPST.

For future works it is proposed the application of both algorithms on the entire set of test problems available in [8]. An improvement that will be studied is an incorporation of a Mixed Integer Programming (MIP) model to AIV or HIVP for solving related sub problems.

Acknowledgements

The authors thank the Brazilian agencies FAPEMIG and CNPq, and the Universidade Federal de Ouro Preto (UFOP) for the financial support on the development of this work.

References

1. Graham, R., Lawler, E., Lenstra, J., Kan, A.: Optimization and approximation in deterministic sequencing and scheduling: a survey. Annals of discrete Mathematics **5**(2) (1979) 287–326
2. Pereira Lopes, M.J., de Carvalho, J.M.: A branch-and-price algorithm for scheduling parallel machines with sequence dependent setup times. European Journal of Operational Research **176** (2007) 1508–1527
3. Karp, R.M.: Reducibility among combinatorial problems. Complexity of Computer Computations **40**(4) (1972) 85–103
4. Garey, M., Johnson, D.: Computers and intractability: A guide to the theory of np-completeness. WH Freeman & Co., San Francisco **174** (1979)
5. Lourenço, H.R., Martin, O., Stützle, T.: Iterated local search. In Glover, F., Kochenberger, G., eds.: Handbook of Metaheuristics. Volume 57 of International Series in Operations Research & Management Science. Kluwer Academic Publishers, Norwell, MA (2003) 321–353
6. Mladenovic, N., Hansen, P.: Variable neighborhood search. Computers and Operations Research **24**(11) (1997) 1097–1100
7. Souza, M., Coelho, I., Ribas, S., Santos, H., Merschmann, L.: A hybrid heuristic algorithm for the open-pit-mining operational planning problem. European Journal of Operational Research **207**(2) (2010) 1041–1051
8. de Optimización Aplicada, S.: (2011) A web site that includes benchmark problem data sets and solutions for scheduling problems. Available at `http://soa.iti.es/problem-instances`.

9. Weng, M.X., Lu, J., Ren, H.: Unrelated parallel machine scheduling with setup consideration and a total weighted completion time objective. International Journal of Production Economics **70** (2001) 215–226
10. Kim, D.W., Na, D.G., Frank Chen, F.: Unrelated parallel machine scheduling with setup times and a total weighted tardiness objective. Robotics and Computer-Integrated Manufacturing **19** (2003) 173–181
11. Logendran, R., McDonell, B., Smucker, B.: Scheduling unrelated parallel machines with sequence-dependent setups. Computers & Operations research **34**(11) (2007) 3420–3438
12. Al-Salem, A.: Scheduling to minimize makespan on unrelated parallel machines with sequence dependent setup times. Engineering Journal of the University of Qatar **17**(1) (2004) 177–187
13. Rabadi, G., Moraga, R.J., Al-Salem, A.: Heuristics for the unrelated parallel machine scheduling problem with setup times. Journal of Intelligent Manufacturing **17**(1) (2006) 85–97
14. Helal, M., Rabadi, G., Al-Salem, A.: A tabu search algorithm to minimize the makespan for the unrelated parallel machines scheduling problem with setup times. International Journal of Operations Research **3**(3) (2006) 182–192
15. Arnaout, J., Rabadi, G., Musa, R.: A two-stage ant colony optimization algorithm to minimize the makespan on unrelated parallel machines with sequence-dependent setup times. Journal of Intelligent Manufacturing **21**(6) (2010) 693–701
16. Ying, K.C., Lee, Z.J., Lin, S.W.: Makespan minimisation for scheduling unrelated parallel machines with setup times. Journal of Intelligent Manufacturing **23**(5) (2012) 1795–1803
17. Chang, P., Chen, S.: Integrating dominance properties with genetic algorithms for parallel machine scheduling problems with setup times. Applied Soft Computing **11**(1) (2011) 1263–1274
18. Fleszar, K., Charalambous, C., Hindi, K.: A variable neighborhood descent heuristic for the problem of makespan minimisation on unrelated parallel machines with setup times. Journal of Intelligent Manufacturing **23**(5) (2011) 1949–1958 doi:10.1007/s10845-011-0522-8.
19. Vallada, E., Ruiz, R.: A genetic algorithm for the unrelated parallel machine scheduling problem with sequence dependent setup times. European Journal of Operational Research **211**(3) (2011) 612–622
20. Resende, M.G.C., Ribeiro, C.C.: 11. In: GRASP: Greedy Randomized Adaptive Search Procedures. 2 edn. Springer (2013) 287–312
21. Baker, K.R.: Introduction to Sequencing and Scheduling. John Wiley & Sons (1974)
22. Bresina, J.L.: Heusistic-biased stochastic sampling. Proceedings of the thirteenth national conference on Artificial intelligence **1** (1996) 271–278
23. Subramanian, A., Drummond, L., Bentes, C., Ochi, L., Farias, R.: A parallel heuristic for the vehicle routing problem with simultaneous pickup and delivery. Computers & Operations Research **37**(11) (2010) 1899–1911
24. Shapiro, S.S., Wilk, M.B.: An analysis of variance test for normality (complete samples). Biometrika **52** (1965) 591–611
25. Montgomery, D.: Design and Analysis of Experiments. fifth edn. John Wiley & Sons, New York, NY (2007)