June 2009 *RR*-05.09

Massimiliano Caramia, Stefano Giordani

A Fast Metaheuristic for Scheduling Independent Tasks with Multiple Modes

A Fast Metaheuristic for Scheduling Independent Tasks with Multiple Modes

Massimiliano Caramia [*] Stefano Giordani [†]

Abstract

We consider the following multi-mode task scheduling problem. Given are a set of non-preemptive and independent tasks, and a set of single unit dedicated renewable resources. At any time each resource can be used by a single task at most. Each task has to be executed without preemption in one out of its possible execution modes, where each mode identifies a subset of resources simultaneously required by the task for its execution. The aim of the problem is to find a mode assignment for each task, and a non-preemptive schedule of the tasks to be executed in the assigned modes, such that the total amount of resources requested by tasks in any time period does not exceed the resource availability, and the schedule length, i.e., the makespan, is minimized. From the complexity viewpoint this problem is NP-hard. Heuristic algorithms for scheduling tasks with multiple modes for the minimum schedule length criterion involve in general two distinct phases, task mode assignment and task scheduling. In this paper we propose a novel two-phase approach metaheuristic based on strategic oscillation and on a randomized choice of the neighborhood of the local search to avoid being trapped in local optima. The proposed approach simplifies that one appeared in a previous work of the authors in which an interval T-coloring graph model and a metaheuristic approach based on strategic oscillation were provided. The performance of the proposed solution approach is compared to that of known multi-mode scheduling heuristics.

Keywords: independent task scheduling; multi-mode; metaheuristic.

[*]Dipartimento di Ingegneria dell'Impresa, Università di Roma "Tor Vergata", Via del Politecnico, 1 - 00133 Roma, Italy. e-mail: caramia@disp.uniroma2.it

[†]Dipartimento di Ingegneria dell'Impresa, Università di Roma "Tor Vergata", Via del Politecnico, 1 - 00133 Roma, Italy. e-mail: giordani@disp.uniroma2.it

1 Introduction

Multi-Mode Task Scheduling (MMTS) can be found in several real-life problems arising in production systems with scarce resources. Basically, MMTS models those situations in which a task can be executed by means of different resource combinations with consequently different processing times. For instance, if we consider an assembly cell, an operation can be executed either by using a fast machine and a robot, or, alternatively, by using a slower machine, a worker, and an assembly tool. The time required to execute the operation depends on the amount and the type of resources associated with its execution mode: the more powerful the resources, the lower the execution time.

In this work, we study the following MMTS problem: given are a set $\mathcal{T}= \{1, \ldots, n\}$ of n independent tasks (i.e., not related by precedence relations), and a set $\mathcal{R}= \{R_1, \ldots, R_r\}$ of r single unit dedicated renewable resources. Each task j has to be executed without preemption in one out of its possible $m_j \geq 1$ execution modes of the set $\mathcal{M}_j = \{M_j^1, \ldots, M_j^{m_j}\}$, where each mode $M_j^i = (\mathcal{R}_j^i; p_j^i)$ identifies the non empty subset $\mathcal{R}_j^i \subseteq \mathcal{R}$ of resources simultaneously required by task j for $p_j^i \geq 1$ time units if the task is executed with that particular mode M_j^i. The objective is to find a mode assignment for each task, and a schedule of the tasks to be executed in the assigned mode, such that at any time each resource is used by a single task at most, and the schedule length, i.e., the makespan, is minimized.

The problem has been proved to be NP-hard (see e.g. Bianco et al. 1995), and most of the previous contributions to this problem deal with the special case with only single unit renewable resources, like the one we address in this paper. The special case of the problem with prespecified resource allocation has been considered for example in Bianco et al. (1994). The more general case with multiple execution modes and with precedence relations among tasks has been analyzed e.g. in Bianco et al. (1998), Bianco et al. (1999).

Most of the approaches in the literature to solve MMTS are heuristics based on two distinct phases, i.e., task mode assignment and task scheduling. For instance, the approach proposed in Bianco et al. (1998) is a local search algorithm that, iteratively, first assigns a mode to every task by heuristically identifying the best among a restricted set of task execution modes, and then heuristically solves the task sequencing problem with the assigned modes. Before a new iteration starts, the subset of candidate execution modes

of each task is conveniently reduced.

By a study of the solution profile, we noted (see Section 3.4.1) that the latter approach ends up with a very few iterations, i.e., 2 or 3, and the CPU time for each iteration tends to become huge for instances with more than 300 tasks. Following these reasons, in this paper we propose to overcome these two drawbacks with a novel two-phase approach. The latter is a metaheuristic algorithm that extends a previous work by Caramia and Giordani (2007) on a heuristic approach to solve MMTS in which the mode assignment and the task scheduling features are managed in an integrated mechanism with mode assignment embedded in scheduling. In particular, in Caramia and Giordani (2007), the authors modelled MMTS as a graph interval T-coloring problem and proposed a solution approach based on strategic oscillation (Glover 2000). This algorithm (called $SOAR$) uses a multiple restart strategy to diversify the search, a randomized choice of the neighborhood of the local search to avoid being trapped in local optima, and strategic oscillation to intensify the search.

The approach we propose in this work has two main motivations: firstly, even if $SOAR$ has an overall better performance than the two-phase approach in Bianco et al. (1998), that indeed, as stated above, was conceived for the more general situation where tasks are related by precedence relations, it tends to consume a large amount of computing time since the interval T-coloring problem is defined on a graph with $O(n \cdot m)$ vertices, being n the number of tasks and m the maximum number of modes required by a task; secondly, the performance of $SOAR$ is quite sensitive to the parameter setting. The new proposed approach tends to overcome these two drawbacks of $SOAR$ while preserving its overall good behavior patterns. On the one hand, we designed the new approach reformulating the strategic oscillation based algorithm without using the graph model, and, on the other hand, we made some of the parameters self-adaptive. For this reason, this approach can be interpreted as a simplification of the approach in Caramia and Giordani (2007).

The performance of the proposed new metaheuristic is compared to the algorithms in Bianco et al. (1998), and Caramia and Giordani (2007).

The remainder of the paper is organized as follows. In Section 2, we describe the proposed algorithm and, in Section 3, we show computational results of the proposed algorithm and the comparison to the two competing approaches discussed above.

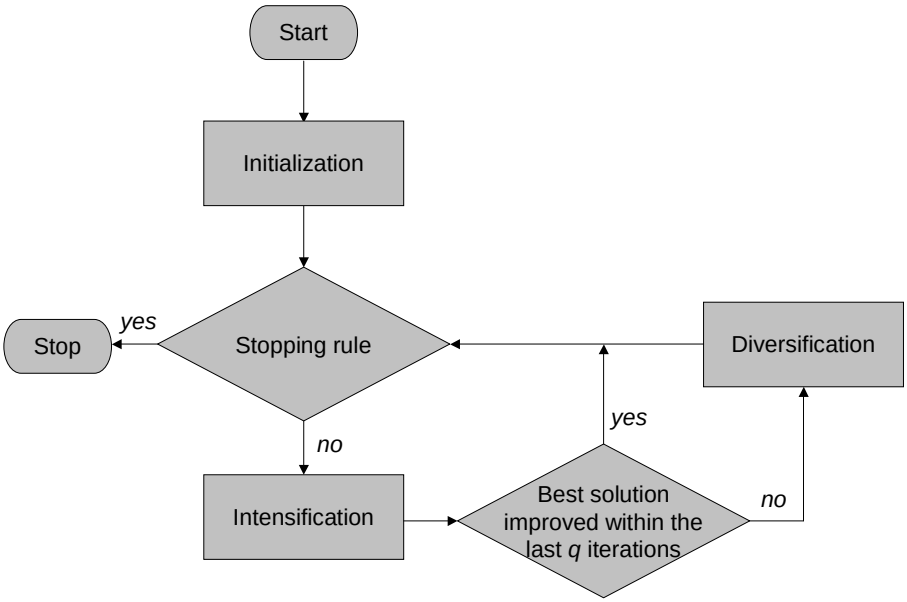

Figure 1: Flow chart of the functionalities of the proposed algorithm.

2 The proposed algorithm

The proposed metaheuristic approach is based on strategic oscillation (see, e.g., Glover 2000), and on a randomized choice of the neighborhood to escape from local minima. Strategic oscillation operates by alternating destructive and constructive phases, where each solution generated by a constructive phase is partially dismantled by a destructive phase, after which a new constructive phase builds a new solution starting from the partial solution obtained after the previous destructive phase. The main blocks of the algorithm are depicted in Figure 1.

In the *Initialization* phase, the algorithm finds a greedy starting solution. Then, in an iterative fashion, it applies the *Intensification* phase that is in charge to improve the current best solution by moving from the current solution to a new solution. In doing that it destroys part of the former solution removing at random some of the tasks, and then it completes this partial solution first by assigning new execution modes to the removed tasks and then by rescheduling these tasks with the new assigned modes as soon as possible in a greedy manner. If the current best solution is not improved within a given number q of iterations the current local search phase is interrupted, and the algorithm applies the *Diversification* phase, to avoid being trapped into a local optimum, by finding a new greedy solution, and by starting a new local search phase from this latter solution.

Table 1: Algorithm pseudocode

set $frequency(M_j^i) = 0, \forall j \in \mathcal{T}, i = 1, \ldots, m_j$; set $iteration = improving\_iteration = 0$
assign a mode $M_j^{h_j}$ to each task j arbitrarily
order tasks arbitrarily
schedule tasks according to the ordering and the modes assigned
let z^* be the current makespan
store the current solution
$z_{prec} = z^*$
Main loop
while not stopping rule **do** {INTENSIFICATION PHASE}
  choose a subset K of $\mathcal{T}$ at random
  increase $frequency(M_j^{h_j}), \forall j \in K$
  remove all tasks $j \in K$
  schedule tasks $j \in \mathcal{T} \setminus K$ respecting all incompatibilities
  for $j \in K$ **do**
    assign a new mode $M_j^{h_j}$ to task j at random, according to a given probability distribution
  end for
  schedule tasks $j \in K$ according to non-increasing values of their duration
  increase *iteration* count
  let z be the current makespan value
  if $z < z^*$ **then**
    update z^* and $improving\_iteration = iteration$
  else
    if $\frac{z - z_{prec}}{z_{prec}} \leq \beta$ **then**
      store the current solution
      $z_{prec} = z$
    else
      restore the previously stored solution
      $z = z_{prec}$
    end if
  end if
  if $improving\_iteration + intensify \leq iteration$ **then** {DIVERSIFICATION PHASE}
    for $j \in \mathcal{T}$ **do**
      assign mode $M_j^{h_j}$ to task j at random according to a probability distribution that gives more chance to modes with lowest $frequency(M_j^{h_j})$ values
    end for
    order tasks arbitrarily
    schedule tasks according to the ordering and the modes assigned
    $frequency(M_j^i) = 0, \forall j \in \mathcal{T}, i = 1, \ldots, m_j$
    if $z < z^*$ **then**
      update z^*
    end if
    set $improving\_iteration = iteration$
  end if
end while

The algorithm stops its execution if a certain stopping criterion is met (e.g., a given number of iterations has been executed).

Referring to Table 1 where a pseudocode description of the procedure is given, the algorithm operates in detail as follows. It starts with the *Initialization* phase with a twofold aim: one is to perform some initialization, and the other is to find a starting schedule. Referring to the former point, the algorithm initializes to zero three counters: one denoted as $frequency(M_j^i)$, that counts the number of times a mode i assigned to task j is discarded during the current local search phase; another counter is *iterations*, that counts the overall iterations performed by the algorithm during its progress. The last counter is *improving_iteration*, that accounts for the iteration at which the algorithm found the current best solution achieved so far. At Steps 2 and 3, the algorithm first assigns a mode $M_j^{h_j} \in \mathcal{M}_j$ to each task j at random, and then finds an ordering of the tasks, also in this case at random. At step 4, according to this ordering and the assigned modes, the algorithm schedules tasks greedily at their earliest starting time. At Step 5, the algorithm calculates the makespan z^* of the resulting schedule, being the best (unique) value found so far. At Steps 6 and 7, the current solution and its value are stored.

Steps from 9 to 42 represent the main body of the algorithm and are repeated until a stopping rule, represented by a prefixed number of iterations or a certain CPU time, is met. In particular, at Step 9, the algorithm chooses a subset K of tasks, whose cardinality is related to another parameter $0 \leq \alpha \leq 1$, i.e., $|K| = \alpha \cdot n$, that will be introduced later in the experimental analysis section.

The choice of the K tasks is made at random, and is based on a parameter $\Delta\_req(j)$ that considers, for each task $j \in \mathcal{T}$, the best improvement in terms of resource requirement with respect to the current mode assignment; that is, if task j is currently executed with mode $M_j^{h_j}$ we have

$$\Delta\_req(j) = \max_{M_j^i \in \mathcal{M}_j} (p_j^{h_j} \cdot |\mathcal{R}_j^{h_j}| - p_j^i \cdot |\mathcal{R}_j^i|).$$

Note that $\Delta\_req(j) \geq 0$. Now, $|K|$ tasks are chosen with a Montecarlo mechanism, by assigning to each task $j \in \mathcal{T}$ a probability equal to

$$prob(j) = \frac{\Delta\_req(j)}{\sum_{j \in \mathcal{T}} \Delta\_req(j)}.$$

Once these tasks have been selected, we increase by one the frequency counter of the

modes associated with them, i.e., $frequency(M_j^{h_j})$ where $j \in K$ and h_j is its currently assigned mode.

At Steps 11 and 12, the algorithm first removes from the current schedule the tasks in K, and then creates a partial schedule of the remaining $|\mathcal{T} \setminus K|$ tasks by scheduling the latter as soon as possible according to the ordering previously established.

For each task $j \in K$ (see Steps 13-15), the algorithm assigns a new mode $M_j^{h_j}$ based on the probability

$$prob(M_j^i) = \frac{\frac{1}{p_j^i \cdot |\mathcal{R}_j^i|}}{\sum_{M_j^{i'} \in \mathcal{M}_j} \frac{1}{p_j^{i'} \cdot |\mathcal{R}_j^{i'}|}},$$

meaning that, for task j, the lower the resource usage of a mode, the higher the probability with which it is assigned to j.

At Step 16, the tasks in K are greedily scheduled according to non-increasing values of their processing times. We note that the priority rule used to schedule tasks in K can be different from the shortest processing time rule adopted, and the choice we made is because this rule, on average, gave the best performance.

Once the algorithm finds a solution, it calculates the objective function value z (Step 18) and then if this represents an improvement on the best solution z^* found so far (Step 19), it updates $z^* = z$ (Step 20). The algorithm can accept or discard the current solution based on its relative distance from the previous accepted solution (see Steps 22-28) of value z_{prec}: indeed, the current solution is rejected if the current z value is very poor with respect to the previous solution value, i.e., when

$$\frac{z - z_{prec}}{z_{prec}} > \beta,$$

where $\beta \geq 0$ is a given acceptance threshold. In other words, $\beta = 0$ means that the algorithm moves only to non-worse solutions, and $\beta = +\infty$ indicates that the algorithm accepts all solutions regardless of their quality. Note that if $z \leq z_{prec}$, solution z is always accepted regardless of its value and the β parameter value.

If z^* has not been improved within the last $intensify$ iterations, the algorithm halts the current local search, and executes the *Diversification* phase; otherwise, the algorithm progresses in the current local search with a new *Intensification* phase. In the *Diversification* phase (see Steps from 30 to 41), the algorithm randomly determines a new greedy solution. In particular, a new mode is assigned to each task $j \in \mathcal{T}$ at random with

a Montecarlo mechanism, giving more chance to those modes the with lowest values of $frequency$. Following a random task order, and according to the assigned modes, the algorithm schedules tasks greedily at their earliest starting time. Then, from that greedy solution a new local search is started, after having reset the counters $frequency(M_j^i)$ to zero.

3 Experimental analysis

The proposed algorithm is named $FAST$ because it is able to find very good solutions in very limited running times. An extensive experimentation on $FAST$ and on the algorithms used for comparisons, that is, the two-phase approach algorithm in Bianco et al. (1998), denoted as TPA, and algorithm $SOAR$ in Caramia and Giordani (2007) has been carried out. In this section, we provide our findings.

The section is divided into four subsections: in the first one, we give implementation details and the structure of the test problems used for the experimentation; in the second one, we analyze the performance of $FAST$ in order to set its parameters; the latter two subsections are devoted to the comparison of $FAST$ to $SOAR$ and TPA, respectively; in particular, we begin the last subsection analyzing the performance of different versions of algorithm TPA in order to select the best one, which will be used in the comparison to our algorithm.

3.1 Implementation details

All the algorithms have been implemented in the C language and executed on a PC with 2.93 GHz Intel Celeron processor and 256MB RAM. For testing the performance of the algorithms we have defined different instance classes characterized by the following parameters:

- Number of tasks, $n = 50, \ldots, 500$;
- Maximum number of task modes, $m = 2, \ldots, 5$;
- Number of available (single unit) resources, $r = 10$;
- Maximum number of resources requested by a task execution mode, $\rho = 2, 5, 8$;

- Maximum duration of a task, $p_{max} = 10$.

Each class is then identified by four parameters, i.e., (n, m, r, ρ). For instance, the class (100,4,10,5) collects instances with 100 tasks, at most 4 modes for each task, and each mode can require at most 5 resources out of 10. In particular, for each class type (n, m, r, ρ) we generated at random 5 instances with n tasks where the number m_j of execution modes for task j is uniformly distributed in the range $[1, m]$, the number $|\mathcal{R}_j^i|$ of resources required by task j in the mode M_j^i is uniformly distributed in the range $[1, \rho]$, and duration p_j^i is uniformly distributed in the range $[1, p_{max}]$. For each instance, the results of $FAST$ are obtained by running the algorithm 5 times and taking the average value.

3.2 Tuning the parameters of $FAST$

In this subsection we analyze the performance of $FAST$, with the aim of setting the values of its parameters.

We have carried out a wide variety of test runs to tune the algorithm parameters of the *Intensification* phase, which are:

- $intensify$: the maximum number of consecutive iterations of the *Intensification* phase without improvement w.r.t. the best solution found so far;
- α: the ratio (in percentage) between $|K|$ and n, that controls the neighborhood size in the local search of the *Intensification* phase;
- β: the acceptance threshold of a non-improving solution w.r.t. the previous accepted solution in the local search of the *Intensification* phase.

We start by analyzing the profile of the solution values found by the algorithm with different values of $intensify$ and show the behavior of the algorithm: we stopped the algorithm after 1500 iterations. In Figures 2 and 3, we show the makespan values achieved by $FAST$ over time when $\beta = 0$, $\alpha = 0.02$ and $intensify$ is fixed to 200 and 80, respectively, for a single run of $FAST$ over one instance in the class $(150, 5, 10, 5)$. In particular, the figures show the iteration count and the makespan values when the *Diversification* phase is executed and the algorithm restarts the local search phase with a new random initial solution; moreover, the minimum ("y min" in the figures), maximum ("y max" in the figures) and average ("y mean" in the figures) values of the schedule lengths found by the

algorithm during its execution are also shown ("x max" in the figures is the maximum number of iterations allowed). At the beginning, in both the two cases, the algorithm starts and executes a large amount of iterations without any diversification and finds a good solution: with $intensify = 200$, this phase lasts 828 iterations and the best solution found is not improved in the remaining iterations; with $intensify = 80$, this phase lasts a smaller number of iterations equal to 373, and the best solution found is successively improved after the first *Diversification* phase.

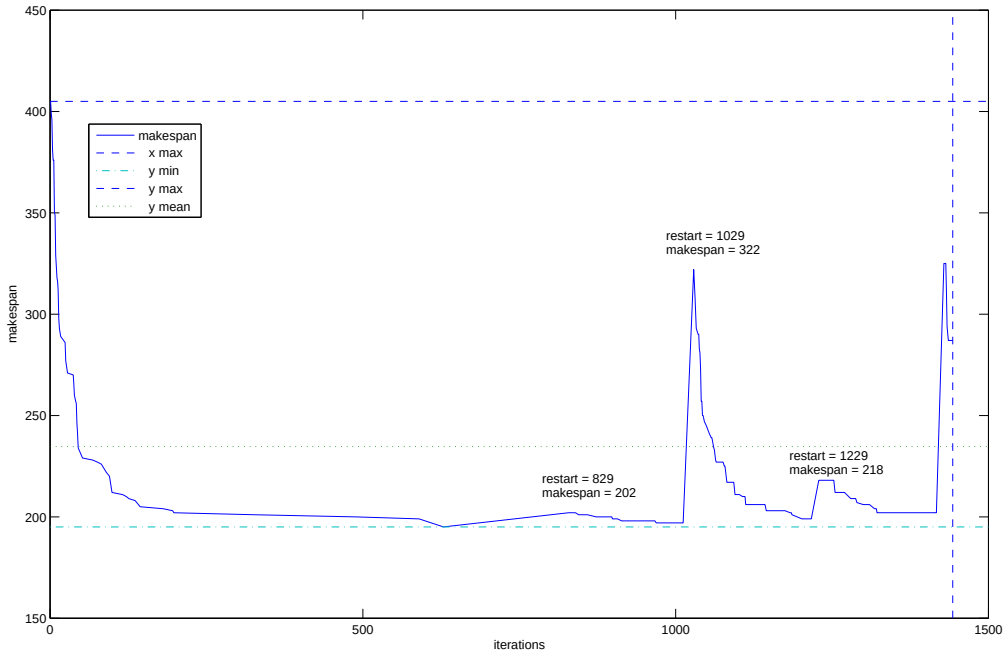

Figure 2: Solution values profile of the algorithm ($intensify = 200$, $\beta = 0$, $\alpha = 0.02$, $n = 150$).

As it can be inferred, with $intensify = 80$ the *Diversification* phase is more frequent than for the case with $intensify = 200$.

Figure 4 plots the trends of the average solution values of the instances with $n = 100, 200$ tasks, $intensify = 200$, $\alpha = 0.02, 0.1, 0.5, 0.9$ and $\beta = 0, 0.02, 0.1, +\infty$; the figure shows that, experimentally, the best choice for α and β are 0.02 and 0, respectively.

With this latter choice, we analyzed the effectiveness and the efficiency of the algorithm, by varying the $intensify$ parameter. We experimented with $intensify = 80, 200, 600, 2000,$

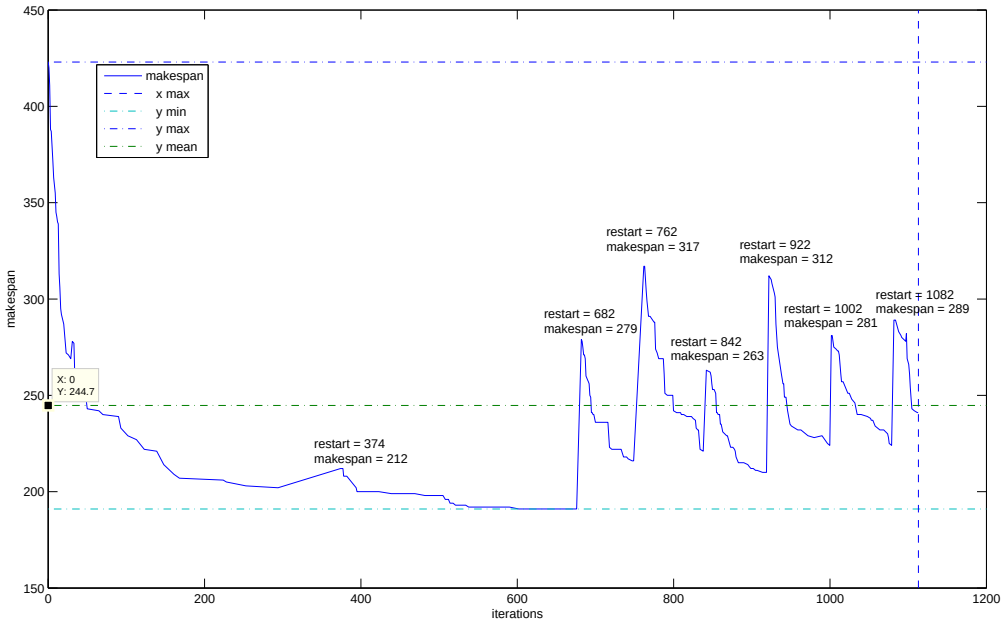

Figure 3: Solution values profile of the algorithm ($intensify = 80$, $\beta = 0$, $\alpha = 0.02$, $n = 150$).

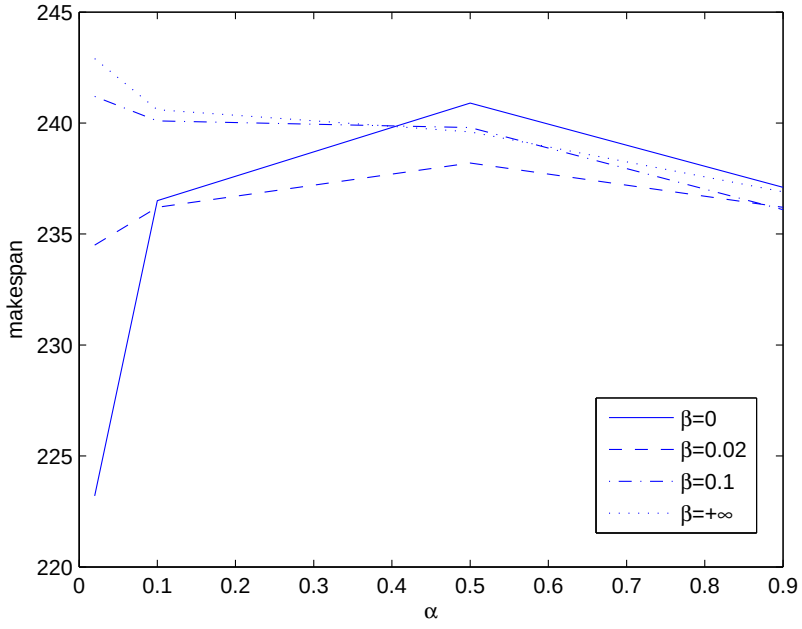

Figure 4: Average makespan values for different α and β values, and $intensify = 200$.

and the results are listed in Table 2 where we report the average makespan and the average CPU time (in seconds) among the instances of the same class (CPU times reported are the times when the best solution values were found); moreover, the average values among all the experimented instances over increasing *intensify* values are plotted in Figure 5. The figure shows the trade-off between the effectiveness and the efficiency of the algorithm, and how for increasing *intensify* values the effectiveness increases while the efficiency decreases.

Table 2: Average makespan and CPU time values over the instance classes ($\alpha = 0.02$; $\beta = 0$).

instance class				$intensify = 80$		$intensify = 200$		$intensify = 600$		$intensify = 2000$	
				makespan	*CPU*	*makespan*	*CPU*	*makespan*	*CPU*	*makespan*	*CPU*
n	m	r	ρ		(sec)		(sec)		(sec)		(sec)
50	2	10	2	48.0	0.8	48.2	1.0	48.0	1.4	48.0	1.6
50	2	10	5	83.8	2.2	87.0	3.2	85.8	5.8	87.4	4.8
50	2	10	8	158.6	4.0	156.8	3.6	156.6	5.6	157.8	6.6
50	4	10	2	37.4	2.2	36.2	3.6	36.0	6.4	36.0	4.4
50	4	10	5	64.2	4.8	62.4	6.0	62.4	13.8	60.6	14.4
50	4	10	8	91.8	21.6	90.6	19.0	88.2	12.8	89.0	28.4
100	2	10	2	93.8	0.6	90.6	4.0	89.0	4.6	89.0	8.2
100	2	10	5	175.4	13.6	170.6	6.4	167.4	19.6	165.8	33.2
100	2	10	8	270.4	8.4	268.0	26.6	261.2	25.4	261.4	39.0
100	4	10	2	66.4	7.6	63.8	18.6	61.8	24.0	61.2	37.0
100	4	10	5	139.6	15.6	135.0	48.0	131.6	76.2	129.2	114.0
100	4	10	8	195.6	46.2	188.2	79.4	182.2	119.4	182.4	181.2
200	2	10	2	159.2	24.8	155.8	32.6	153.4	28.6	151.4	67.8
200	2	10	5	338.6	26.2	334.2	71.6	327.0	128.2	323.8	241.8
200	2	10	8	514.8	21.2	507.0	123.4	499.2	231.4	495.0	406.4
200	4	10	2	126.2	120.4	123.4	43.0	119.0	118.2	116.6	222.0
200	4	10	5	268.2	96.2	258.4	157.0	252.0	539.8	246.2	944.4
200	4	10	8	381.0	129.8	376.8	260.4	372.2	858.4	370.4	1537.2

3.2.1 Changing parameter values dynamically

From the effectiveness point of view, we experimentally obtained better results with $\alpha = 0.02$ and $\beta = 0$, that is, with a very narrow neighborhood and only improvement moves in the *Intensification* phase. Nevertheless, we also experimented that with a wider neighborhood (i.e., with $\alpha = 0.3$) and accepting also non-improvement moves within a given threshold (i.e., with $\beta = 0.03$), we were able to reach a good solution in a shorter time. From this consideration, we decided to vary α and β dynamically within the algorithm run, adopting a sort of variable neighborhood search with a variable threshold accepting strategy. The idea is to divide each local search phase into 2 sub-phases: in

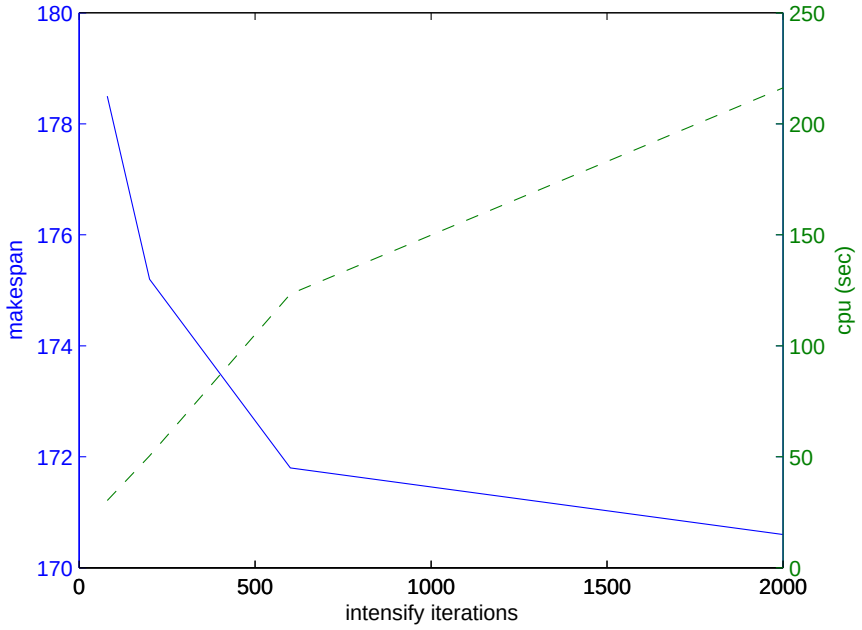

Figure 5: Average makespan values (solid line) and CPU times (dotted line) over increasing $intensify$ values ($\alpha = 0.02$; $\beta = 0$).

the first one (i.e., sub-phase 1) the algorithm searches new solutions over a wide neighborhood (e.g., $\alpha_1 = 0.3$) and also accepts non-improving solutions (e.g., $\beta_1 = 0.05$); in the next sub-phase 2, the algorithm changes the search strategy by searching into a narrow neighborhood (e.g., $\alpha_2 = 0.02$) and without accepting non-improving solutions (e.g., $\beta_2 = 0$).

Accordingly, with sub-phase 1 the algorithm is able to look for a new solution over a large neighborhood space accepting also non-improving moves, in order to avoid to be trapped soon in a local optimum; sub-phase 2 is successively used to improve the solution values by intensifying the search on a small neighborhood, if for a certain time period we were unable to find good solutions w.r.t. the best solution found. In particular, the transition from sub-phase 1 to sub-phase 2 is done when, within a certain number of iterations or CPU time t_0, the algorithm fails in finding a new solution of value z not greater than $(1 + \Delta)z^*$, where z^* is the best solution value found so far by the algorithm, and Δ is an arbitrarily small parameter.

In Tables 3-5, we show the best makespan values and the corresponding running times, obtained with $t_0 = 20$ seconds and $\Delta = 0.1$, for different values of α_1 and β_1, and with

Table 3: Average makespan and CPU time values for different α_1 values and $\beta_1 = 0$.

instance class				$\beta_1 = 0$					
				$\alpha_1 = 0.1$		$\alpha_1 = 0.3$		$\alpha_1 = 0.6$	
n	m	r	ρ	$makespan$	CPU(sec)	$makespan$	CPU(sec)	$makespan$	CPU(sec)
300	2	10	2	235.6	87.4	235.8	166.0	236.0	103.0
300	2	10	5	492.4	160.8	483.2	187.2	494.6	93.2
300	2	10	8	769.4	187.0	767.2	186.0	782.4	116.0
300	4	10	2	187.0	92.0	186.2	97.0	187.0	101.0
300	4	10	5	389.8	123.0	393.8	131.8	397.0	106.6
300	4	10	8	593.6	208.0	605.0	134.0	606.4	118.0
500	2	10	2	379.8	150.0	389.4	145.0	394.0	95.4
500	2	10	5	862.2	117.2	841.2	117.2	855.4	66.4
500	2	10	8	1318.4	138.0	1311.0	138.0	1327.2	91.4
500	4	10	2	328.0	119.0	317.6	89.0	323.6	94.2
500	4	10	5	674.2	137.2	672.6	154.2	673.6	153.8
500	4	10	8	1021.0	157.0	1007.0	86.2	988.8	120.0

Table 4: Average makespan and CPU time values for different α_1 values and $\beta_1 = 0.05$.

instance class				$\beta_1 = 0.05$					
				$\alpha_1 = 0.1$		$\alpha_1 = 0.3$		$\alpha_1 = 0.6$	
n	m	r	ρ	$makespan$	CPU(sec)	$makespan$	CPU(sec)	$makespan$	CPU(sec)
300	2	10	2	235.2	105.0	240.2	93.4	233.6	110.0
300	2	10	5	474.6	190.4	469.0	142.6	476.8	121.4
300	2	10	8	778.4	158.0	779.0	155.0	780.4	167.0
300	4	10	2	186.0	117.0	182.0	115.0	186.8	113.0
300	4	10	5	392.6	133.0	388.0	141.8	390.6	139.6
300	4	10	8	602.6	202.0	608.8	132.0	606.8	146.0
500	2	10	2	387.6	191.0	362.5	119.8	388.2	129.0
500	2	10	5	841.2	202.4	851.6	120.6	856.0	108.4
500	2	10	8	1327.2	97.8	1310.8	82.6	1312.0	124.0
500	4	10	2	317.6	116.0	279.8	132.2	310.6	103.0
500	4	10	5	671.6	167.4	626.6	90.4	691.8	142.4
500	4	10	8	1035.0	116.0	1005.8	84.0	1023.8	45.0

$\alpha_2 = 0.02$ and $\beta_2 = 0$. By analysing the results in these tables, we note that most of the best results are obtained when $\alpha_1 = 0.3$ and $\beta_1 = 0.05$.

In the following, we compare the algorithm version with dynamic parameters (with $t_0 = 20$ seconds, $\Delta = 0.1$, $\alpha_1 = 0.3$, $\beta_1 = 0.05$, $\alpha_2 = 0.02$ and $\beta_2 = 0$) and the version with fixed parameters, where we do not vary α and β, that is, when only sub-phase 2 is

Table 5: Average makespan and CPU time values for different α_1 values and $\beta_1 = 0.2$.

instance class				$\beta_1 = 0.2$					
				$\alpha_1 = 0.1$		$\alpha_1 = 0.3$		$\alpha_1 = 0.6$	
n	m	r	ρ	$makespan$	CPU(sec)	$makespan$	CPU(sec)	$makespan$	CPU(sec)
300	2	10	2	235.0	99.2	235.2	120.4	231.6	117.6
300	2	10	5	490.8	145.8	497.4	149.4	490.8	114.6
300	2	10	8	789.4	184.0	783.4	258.2	785.6	182.0
300	4	10	2	187.6	120.0	186.6	103.6	185.4	118.4
300	4	10	5	395.4	139.4	393.0	140.8	389.6	159.6
300	4	10	8	607.8	120.6	601.6	142.6	605.2	199.4
500	2	10	2	389.6	159.4	386.6	106.0	388.0	115.0
500	2	10	5	848.2	147.2	847.2	191.8	839.4	162.8
500	2	10	8	1334.2	132.4	1322.6	113.4	1315.6	91.6
500	4	10	2	317.4	135.2	314.6	116.4	315.4	127.6
500	4	10	5	668.0	125.4	677.0	84.8	677.8	43.0
500	4	10	8	1008.0	121.2	997.4	145.8	1016.8	93.2

executed (i.e., $t_0 = 0$, $\alpha_2 = 0.02$ and $\beta_2 = 0$).

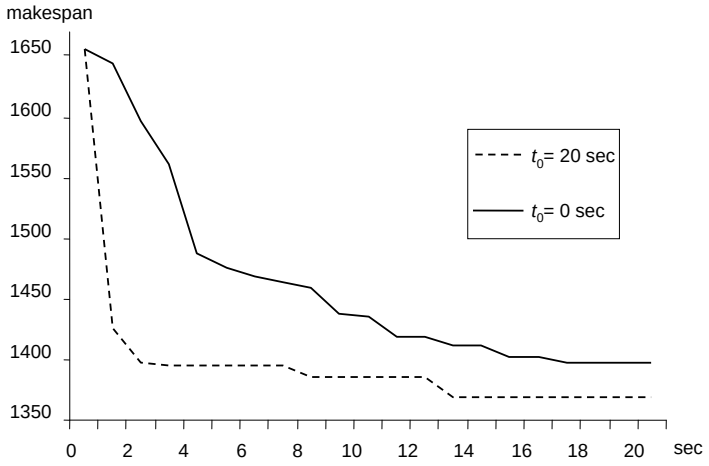

Figure 6: Best solution values found with $t_0 = 0$ and $t_0 = 20$ seconds, respectively, on an instance in the class $(500, 2, 10, 8)$.

Figure 6 shows the makespan profile of the two versions during the first stages of their execution obtained by running once the two algorithms on one large instance belonging to the class $(500, 2, 10, 8)$; the figure shows that in the first 20 seconds, the algorithm implemented with dynamic parameters is able to achieve better solutions w.r.t. the fixed parameters version, and, moreover, that a good solution is obtained very soon (within

Table 6: Average makespan values and CPU times comparison using fixed vs dynamic values of α and β.

instance class				fixed parameters		dynamic parameters	
n	m	r	ρ	*makespan*	CPU(sec)	*makespan*	CPU(sec)
300	2	10	2	232.8	106.8	240.2	93.4
300	2	10	5	488.2	144.8	469.0	142.6
300	2	10	8	790.6	146.0	779.0	154.8
300	4	10	2	191.4	107.4	182.0	115.4
300	4	10	5	396.6	144.4	388.0	141.8
300	4	10	8	609.2	148.0	608.8	131.8
500	2	10	2	390.2	128.6	362.5	119.8
500	2	10	5	848.2	154.2	851.6	120.6
500	2	10	8	1320.8	141.0	1310.8	82.6
500	4	10	2	323.2	117.4	279.8	132.2
500	4	10	5	685.0	172.6	626.6	90.4
500	4	10	8	996.0	174.4	1005.8	84.0

3 seconds). For instances with 300 and 500 tasks, Table 6 shows the comparison of the results obtained by the two algorithms with 200 seconds of time limit over all the instances.

3.3 The comparison between $FAST$ and $SOAR$

In this subsection we compare our algorithm $FAST$ with the dynamic parameters discussed in the previous subsection to algorithm $SOAR$ in Caramia and Giordani (2007). The comparison has been carried using the same set of test instances (the 10 random instances used in Caramia and Giordani 2007) and the same machine. Table 7 shows the average values of the best solutions (column denoted *makespan*) and the corresponding computing times, in seconds, to find them (column denoted CPU), achieved by the two approaches on instances with $n = 50$, 100 and 150 tasks.

The experimentation is made by running $SOAR$ and $FAST$ with a time limit equal to 500 seconds and 100 seconds, respectively, to assess that $FAST$ is indeed often able to improve $SOAR$ solutions in reduced computing time. By the results listed in the table it can be seen that $FAST$ is able to improve the quality of the solution given by $SOAR$ in many cases and especially for the instances with a large number of tasks; also the CPU time needed to find the best solutions within the given time limits shows that $FAST$ on average is faster than $SOAR$. In particular, for instances with 50 tasks we have an average makespan of 73.7 for both the algorithms and an average CPU time of 34.7 seconds for

Table 7: Makespan comparison between $FAST$ and $SOAR$ algorithms on instances ranging from 50 to 150 tasks.

instance class				$SOAR$		$FAST$	
n	m	r	ρ	$makespan$	CPU(sec)	$makespan$	CPU(sec)
50	5	10	2	33.8	29.4	33.8	82.4
50	4	10	2	31.4	194.2	33.7	11.3
50	3	10	2	36.2	110.1	36.2	36.7
50	2	10	2	51.3	73.4	52.0	4.5
50	5	10	5	73.4	279.7	73.4	27.6
50	4	10	5	61.2	79.4	59.1	4.0
50	3	10	5	55.4	114.7	56.3	19.9
50	2	10	5	86.5	2.3	84.2	33.2
50	5	10	8	85.7	3.4	85.7	34.4
50	4	10	8	127.4	16.6	129.1	41.0
50	3	10	8	120.2	61.2	119.8	39.6
50	2	10	8	122.1	18.1	121.0	82.2
100	5	10	2	59.8	92.5	59.8	31.6
100	4	10	2	68.3	57.4	68.3	10.0
100	3	10	2	64.8	175.3	62.2	34.3
100	2	10	2	105.0	21.4	104.2	14.0
100	5	10	5	155.2	101.3	153.7	31.4
100	4	10	5	133.0	67.0	131.2	26.7
100	3	10	5	150.6	21.4	151.8	25.4
100	2	10	5	161.3	56.3	159.1	8.3
100	5	10	8	189.6	11.5	193.4	4.6
100	4	10	8	221.8	31.5	218.5	69.0
100	3	10	8	242.6	57.4	234.4	88.1
100	2	10	8	236.5	107.0	234.1	30.6
150	5	10	2	92.1	49.5	92.1	52.1
150	4	10	2	110.0	62.2	109.3	31.5
150	3	10	2	105.3	129.5	106.3	45.6
150	2	10	2	145.1	53.6	141.7	53.2
150	5	10	5	228.0	164.3	228.0	60.2
150	4	10	5	197.2	123.7	207.2	16.4
150	3	10	5	213.6	11.7	211.0	2.2
150	2	10	5	258.2	14.6	247.6	18.4
150	5	10	8	270.1	8.2	270.1	80.3
150	4	10	8	321.6	4.6	315.6	47.2
150	3	10	8	366.2	198.0	365.4	73.8
150	2	10	8	377.0	3.2	365.2	13.9

$FAST$ and 81.9 seconds for $SOAR$. For instances with 100 tasks our algorithm achieved an average makespan of 147.6 with respect to 149.0 of $SOAR$, and an average CPU time of 31.2 seconds compared to the 66.7 seconds of $SAOR$. Finally, on instances with 150

tasks, $FAST$ produced an average makespan of 221.6 with an average CPU time of 41.2; $SOAR$, instead, achieved an average makespan of 223.7 and an average CPU time of 68.6 seconds on the same instance set.

3.4 The comparison between $FAST$ and TPA

In this subsection we compare the performance of $FAST$ (with dynamic parameters) and algorithm TPA proposed in Bianco et al. (1998). Preliminary, we conducted an analysis of the performance of different versions of TPA, in order to select the best one used for the comparison with $FAST$.

3.4.1 Experimental analysis of different versions of TPA

Algorithm TPA is a local search algorithm based upon a heuristic investigation of the space of task mode assignments, and upon heuristically solving the task scheduling problem for a given mode assignment. The algorithm, at each iteration, first assigns a mode to each task and then finds a feasible schedule for the tasks with the assigned modes by greedily scheduling the tasks according to a sequencing rule. Before executing a new iteration, a variation phase is executed for reducing the space of possible mode assignments. Different versions of TPA were proposed by the authors, according to the criterion selected in the variation phase, the specific heuristic for the assignment phase, and the sequencing rules adopted in the scheduling phase.

We tested algorithm TPA considering the *mode exchange* (ME), and the *critical path* (CP) criteria for the variation phase; moreover, we consider three sequencing rules for the sequencing phase, that is: *longest processing time* (LPT); *shortest processing time* (SPT); *maximum degree of competition* (MDC); finally, for the mode assignment phase, we consider the *assignment MR* procedure that greedily selects a mode for each task minimizing the usage of the most used resource, which seems to perform better with respect to the other ones provided in Bianco et al. (1998).

In Table 8, we report the performance of the two-phase algorithm implemented with different sequencing rules, and with the ME criterion for the variation phase, that experimentally performs better than the CP criterion.

For each instance class with $n = 50, 100, 150$, the table lists the average solution values, and the average CPU times in seconds to find the best solutions, for each sequencing rule

Table 8: Performances of TPA implemented with different sequencing rules and with the ME variation criterion.

instance class				LPT		SPT		MDC	
n	m	r	ρ	*makespan*	CPU (sec)	*makespan*	CPU (sec)	*makespan*	CPU (sec)
50	2	10	2	51.0	0.0	48.8	0.0	48.8	0.0
50	2	10	5	100.2	0.0	102.8	0.0	101.2	0.2
50	2	10	8	169.4	0.0	168.4	0.0	169.2	0.0
50	3	10	2	44.2	0.0	44.0	0.0	42.6	0.0
50	3	10	5	95.6	0.0	95.2	0.0	92.4	0.2
50	3	10	8	139.8	0.0	140.4	0.0	142.6	0.0
50	4	10	2	38.8	0.0	39.8	0.0	40.4	0.0
50	4	10	5	79.6	0.0	82.0	0.0	81.2	0.0
50	4	10	8	101.8	0.0	102.0	0.0	103.4	0.0
50	5	10	2	37.0	0.0	35.8	0.0	35.6	0.0
50	5	10	5	83.4	0.0	85.2	0.0	82.4	0.2
50	5	10	8	113.0	0.0	113.4	0.0	112.8	0.0
100	2	10	2	92.8	0.4	95.2	0.4	92.6	0.4
100	2	10	5	190.4	0.4	189.8	0.2	189.8	0.6
100	2	10	8	302.0	0.2	306.8	0.4	301.8	0.6
100	3	10	2	72.4	0.2	72.8	0.2	73.6	0.2
100	3	10	5	159.6	0.2	161.2	0.2	161.6	0.2
100	3	10	8	240.0	0.2	243.0	0.4	239.6	0.2
100	4	10	2	68.6	0.2	68.6	0.4	68.2	0.2
100	4	10	5	151.0	0.2	157.2	0.4	153.8	0.4
100	4	10	8	215.6	0.2	218.0	0.2	216.0	0.4
100	5	10	2	62.6	0.8	63.2	0.2	62.8	0.2
100	5	10	5	132.4	0.0	133.8	0.4	131.4	0.4
100	5	10	8	192.2	0.4	193.8	0.6	195.4	0.4
150	2	10	2	119.0	1.4	122.4	1.4	119.6	1.2
150	2	10	5	287.4	2.0	292.8	1.8	288.2	1.6
150	2	10	8	425.8	2.0	427.0	1.6	433.8	1.8
150	3	10	2	99.8	1.4	99.8	1.6	98.4	1.0
150	3	10	5	241.8	1.8	246.4	1.6	246.0	1.4
150	3	10	8	363.6	2.4	367.0	1.6	368.2	1.8
150	4	10	2	96.0	1.8	97.0	1.6	97.2	1.2
150	4	10	5	221.0	1.8	221.8	1.6	220.2	1.6
150	4	10	8	322.0	1.8	325.8	2.0	322.8	2.0
150	5	10	2	86.6	1.2	86.6	1.8	86.8	1.2
150	5	10	5	188.6	1.6	192.8	1.4	188.2	1.4
150	5	10	8	293.4	1.8	294.6	2.0	298.6	1.4

adopted in the TPA algorithm; on average, the LPT sequencing rule performs better than the other ones.

The computation time is small and not larger than 2 seconds on average. Indeed, the

algorithm gets stuck in a local optimum very soon (i.e., after 2 or 3 iterations). Nevertheless, for larger instances, that is when $n \geq 300$, the execution time of the algorithm grows very fast, e.g., the algorithm needs about 30 seconds per iteration when $n = 300$, and about 160 seconds when $n = 500$.

3.4.2 *FAST* vs. *TPA*

In the following, we report the results for the comparison between *FAST* and *TPA*. In particular, according to the analysis reported above, the version of *TPA* used for the comparison is the one with the *ME* variation criterion, the *LPT* sequencing rule, and the *MR* procedure for the mode assignment phase. The results are listed in Tables 9 and 10.

Table 9: Makespan comparison between *FAST* and *TPA* on instances with 50 and 100 tasks.

instance class				*TPA*		*FAST*		
n	m	r	ρ	*makespan*	CPU(sec)	*makespan*	*gap*	CPU(sec)
50	5	10	2	37.0	0.0	32.2	-13.0%	8.4
50	4	10	2	38.8	0.0	35.8	-7.7%	5.2
50	3	10	2	44.2	0.0	41.2	-6.8%	1.4
50	2	10	2	51.0	0.0	48.6	-4.7%	0.4
50	5	10	5	83.4	0.0	68.0	-18.5%	18.6
50	4	10	5	79.6	0.0	61.6	-22.6%	12.0
50	3	10	5	95.6	0.0	80.0	-16.3%	8.0
50	2	10	5	100.2	0.0	88.2	-12.0%	5.0
50	5	10	8	113.0	0.0	95.0	-15.9%	56.2
50	4	10	8	101.8	0.0	89.0	-12.6%	28.2
50	3	10	8	139.8	0.0	122.8	-12.2%	14.6
50	2	10	8	169.4	0.0	158.0	-6.7%	11.2
100	5	10	2	62.6	0.4	59.4	-5.1%	17.6
100	4	10	2	68.6	0.2	62.2	-9.3%	34.0
100	3	10	2	72.4	0.0	70.2	-3.0%	16.0
100	2	10	2	92.8	0.2	88.2	-5.0%	4.0
100	5	10	5	132.4	0.6	116.0	-12.4%	39.0
100	4	10	5	151.0	0.2	130.2	-13.8%	44.4
100	3	10	5	159.6	0.2	140.8	-11.8%	20.4
100	2	10	5	190.4	0.6	166.0	-12.8%	6.4
100	5	10	8	192.2	0.4	165.4	-13.9%	48.6
100	4	10	8	215.6	0.4	183.2	-15.0%	93.0
100	3	10	8	240.0	0.2	211.4	-11.9%	38.4
100	2	10	8	302.0	0.8	260.6	-13.7%	26.6

Columns *makespan* list the average values of the makespan. Columns *CPU* report

the average CPU times in seconds to find the best solutions. Finally, column *gap* reports the average relative difference $\left(\frac{makespan(FAST)-makespan(TPA)}{makespan(TPA)}\right)$, in percentage, between the makespan values provided by $FAST$ and TPA.

Table 10: Makespan comparison between $FAST$ and TPA on instances with 200, 300 and 500 tasks.

instance class				TPA		$FAST$		
n	m	r	ρ	*makespan*	CPU(sec)	*makespan*	*gap*	CPU(sec)
200	5	10	2	115.6	8.6	116.8	1.0%	106.8
200	4	10	2	123.2	4.0	120.4	-2.3%	71.0
200	3	10	2	134.2	5.0	131.6	-1.9%	81.0
200	2	10	2	155.8	5.4	152.8	-1.9%	53.0
200	5	10	5	249.2	5.2	233.2	-6.4%	152.8
200	4	10	5	280.4	5.8	261.6	-6.7%	112.2
200	3	10	5	314.2	4.2	290.8	-7.4%	131.2
200	2	10	5	361.2	6.0	325.8	-9.8%	126.6
200	5	10	8	369.6	5.8	336.4	-9.0%	158.6
200	4	10	8	420.2	6.6	388.4	-7.6%	164.4
200	3	10	8	469.0	7.2	427.4	-8.9%	129.0
200	2	10	8	544.2	6.8	504.0	-7.4%	159.8
300	5	10	2	161.0	26.4	155.0	-3.7%	92.6
300	4	10	2	180.8	20.8	182.0	0.7%	115.4
300	3	10	2	195.0.	27.4	197.2	-1.1%	80.4
300	2	10	2	231.2	27.2	240.2	3.9%	93.4
300	5	10	5	350.2	29.2	348.6	-0.5%	106.8
300	4	10	5	388.4	22.6	388.0	-0.1%	141.8
300	3	10	5	450.4	26.6	440.6	-2.2%	148.6
300	2	10	5	512.0	29.4	469.0	-8.4%	142.6
300	5	10	8	556.4	29.6	551.0	-1.0%	104.2
300	4	10	8	621.0	31.2	608.8	-2.0%	131.8
300	3	10	8	684.8	32	659.8	-3.7%	130.2
300	2	10	8	828.2	31.6	779.0	-5.9%	154.8
500	5	10	2	256.4	110.6	250.0	-2.5%	113.8
500	4	10	2	286.6	197.8	279.8	-2.4%	132.2
500	3	10	2	306.8	195.8	305.4	-0.5%	112.0
500	2	10	2	368.2	172.6	362.5	-1.6%	119.8
500	5	10	5	580.2	141.2	592.6	2.3%	163.2
500	4	10	5	621.6	124.6	626.6	0.8%	90.4
500	3	10	5	708.8	136.2	722.8	2.0%	127.0
500	2	10	5	840.4	184	851.6	1.3%	120.6
500	5	10	8	877.6	134.6	862.4	-1.7%	168.2
500	4	10	8	993.0	134.8	1005.8	1.3%	84.0
500	3	10	8	1141.8	162	1134.4	-0.6%	90.2
500	2	10	8	1329.0	205	1310.8	-1.4%	82.6

Results in Tables 9-10 show that the makespan values of $FAST$ are on average 12.4% less than those of TPA on instances with 50 tasks, and 10.6% on instances with 100 tasks. For larger instances this gap reduces to the following values: 5.7% when $n = 200$, 2.0% when $n = 300$, and 0.3% when $n = 500$. We note that this gap reduction has not to be interpreted as a negative outcome of $FAST$: indeed, while TPA halts when it gets stuck in a local optimum, our algorithm has a stopping criterion of 200 seconds regardless of what is the number tasks; therefore, a larger time limit could lead to even better solution values for instances with 300 and 500 tasks.

4 Conclusions

In this paper we proposed a novel two-phase approach metaheuristic for multi-mode task scheduling. The algorithm exploits concepts like strategic oscillation and variable neighborhood search. Indeed, it adopts an intensification phase in which first uses a wide neighborhood for a certain number of iterations, and then, if no improvement is met, it passes to a narrow neighborhood to refine the search. A multi-start mechanism is implemented to diversify solutions. The performance of the proposed solution approach has been compared to that of two known multi-mode scheduling heuristics, showing how it is able to produce often better results in a very limited computing time.

References

[1] L. Bianco, P. Dell'Olmo, M.G. Speranza, Nonpreemptive Scheduling of Independent Tasks with Prespecified Processor Allocations, Nav. Res. Log. 41 (1994) 959–971.

[2] L. Bianco, P. Dell'Olmo, M.G. Speranza, Scheduling Independent Tasks with Multiple Modes, Disc. Appl. Math. 62 (1995) 35–50.

[3] L. Bianco, P. Dell'Olmo, M.G. Speranza, Heuristics for Multi-Mode Scheduling Problems with Dedicated Resources, Eur. J. of Oper. Res. 107 (1998) 260–271.

[4] L. Bianco, P. Dell'Olmo, S. Giordani, M.G. Speranza, Minimizing Makespan in a Multimode Multiprocessor Shop Scheduling Problem, Nav. Res. Log. 46 (1999) 893–911.

[5] M. Caramia, S. Giordani, A New Approach for Scheduling Independent Tasks with Multiple Modes, J. of Heuristics DOI 10.1007/s10732-007-9062, in press, 2007.

[6] F. Glover, Multi-Start and Strategic Oscillation Methods - Principles to Exploit Adaptive Memory, in: M. Laguna, J.L. Gozales Velarde, eds., Computing Tools for Modeling, Optimization and Simulation: Interfaces in Computer Science and Operations Research, Kluwer Academic Publishers, Amsterdam, 2000, pp. 1–24.