



A multi-start Dynasearch algorithm for the time dependent single-machine total weighted tardiness scheduling problem

Eric Angel, Evripidis Bampis

► To cite this version:

Eric Angel, Evripidis Bampis. A multi-start Dynasearch algorithm for the time dependent single-machine total weighted tardiness scheduling problem. *European Journal of Operations Research*, 2005, 162 (1), pp.281–289. hal-00341340

HAL Id: hal-00341340

<https://hal.science/hal-00341340>

Submitted on 19 Jul 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A multi-start Dynasearch algorithm for the time dependent single-machine total weighted tardiness scheduling problem

E. Angel, E. Bampis
Université d'Évry Val-d'Essonne
LaMI
91025 Evry, France
email: {angel, bampis}@lami.univ-evry.fr

June 11, 2001

Abstract

We extend the dynasearch technique, recently proposed by Congram et al, in the context of time-dependent combinatorial optimization problems. As an application we consider a general time-dependent (idleness) version of the well known single-machine total weighted tardiness scheduling problem, in which the processing time of a job depends on its starting time of execution. We develop a multi-start local search algorithm and present experimental results on several types of instances showing the superiority of the dynasearch neighborhood over the traditional one.

1 Introduction

In this paper we evaluate the performance of dynasearch in the context of time-dependent scheduling. Dynasearch is a recently proposed neighborhood search technique [5] that allows a series of moves to be performed at each iteration of a local search algorithm, generating in that way an exponential size neighborhood. In order to efficiently explore such a neighborhood, dynasearch uses dynamic programming. Congram et al. applied dynasearch to the classical single machine total weighted tardiness problem ($1||\sum w_j T_j$) [5] and compared the quality of the obtained solutions with traditional multi-start and iterated descent algorithms. The obtained computational results were very encouraging in the case where dynasearch was applied inside an iterated local search algorithm when compared to classical iterated descent algorithms. However, the application of dynasearch in the case of a multi-start algorithm gave marginally better results than the classical multi-start methods. It is then natural to ask if dynasearch is not so appropriate for multi-start local search algorithms. In this work, we show that this is not true. More precisely, we study the time dependent version of the single machine total weighted tardiness problem, and we present computation results showing that multi-start dynasearch clearly dominates the classical multi-start local search algorithms.

In scheduling theory there have been an increasing interest, in the last few years, for scheduling problems with time-dependent processing times [1]. In this paper we consider a general time-dependent version of the well known single-machine total weighted tardiness scheduling problem. The problem can be stated as follows. We are given a set of n jobs, each job j has a due date d_j and a positive weight w_j . The processing time $f_j(t)$ of each job j depends on its starting time of execution t and is given by a function f_j . We shall note p_j^t for $f_j(t)$. So if a job j immediately starts after a job i , its duration is $p_j^{C_i}$ with C_i the

completion time of job i . We shall consider only the idleness version, and consider that all values are integer ones.

Since we make no assumptions on the functions f_j , this model captures a wide range of practical applications. A first example is when the availability of the resources (e.g. processing power) vary (e.g. in a monotone or cyclic way) over time; think for example at the load of a computer network. A second example is when any delay in the execution of a job may leads to an increase (resp. decrease) of the difficulty of the job and therefore to a modified duration; think for example to fire fighting (resp. destroying a target which is getting closer).

We denote by C_j the completion time and by $T_j = \max\{C_j - d_j, 0\}$ the tardiness of job j . The objective is to find a schedule which minimizes the total weighted tardiness $\sum_{j=1}^n w_j T_j$. Adopting the three-field standard notation of Graham et al. [7] we will denote this problem by $1 | p_j^t, idleness | \sum_j w_j T_j$.

This problem is strongly NP-hard since it is a generalization of the single-machine total weighted tardiness problem $1 || \sum_j w_j T_j$ [11]. Indeed, there exists a dynamic programming algorithm with a running time $\mathcal{O}(n^3 \sum_{j=1}^n p_j)$ for the problem $1 || \sum_j w_j T_j$, but only when weights are agreeable, that is $p_j \geq p_k \Rightarrow w_j \leq w_k$ for all jobs j and k [11]. There exists a branch and bound algorithm for the $1 || \sum_j w_j T_j$ problem [14], but as it is reported in [5] it cannot be used in practice on instances with more than 50 jobs. Moreover the design of approximation algorithms seems very hard, since the only known results concern only the far less general $1 || \sum_j T_j$ problem with a FPTAS due to Lawler [12] in $\mathcal{O}(n^7/\epsilon)$ time, and slightly improved by Kovalyov [10] in $\mathcal{O}(n^6 \log n + n^6/\epsilon)$ time. Since the problem we consider is a broad generalization of the $1 || \sum_j w_j T_j$ problem, these results stress the importance of the metaheuristic approach if one wants to practically deal with instances of this problem.

2 Dynasearch neighborhood

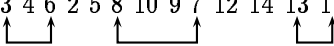
Local search algorithms, and their generalizations such as simulated annealing and tabu search (also called metaheuristics), are often used to obtain near optimal solutions for a wide range of NP-hard combinatorial optimization problems [15, 3, 4]. In these methods a *neighborhood* is defined, usually by giving a set of transformations that can be applied on the current solution. In the simplest local search method, at each iteration the algorithm searches the neighborhood of the current solution for a better one. If such a solution exists it becomes the new current solution and the process goes on, otherwise the algorithm has found a *local optimal* solution and stops.

In general, the more large is the neighborhood, the less there are local optima and the better in quality they are. Recently exponential size neighborhoods, which can be nevertheless searched in polynomial time, have been proposed for the traveling salesman problem [8] (see also the pioneer works [16] and [17]), the one machine batching problem [9], and the single machine total weighted tardiness scheduling problem [5].

The dynasearch neighborhood we use is based on the *swap* neighborhood which gives the best results compared to other ones for the $1 || \sum_j w_j T_j$ problem [6] [2], and probably for the generalized problem that we consider.

We shall represent a solution by a permutation $\sigma = (\sigma(1), \dots, \sigma(n))$ of the set $\{1, 2, \dots, n\}$, meaning that job $\sigma(j)$ is the j -th job to be scheduled.

Given a permutation $\sigma = (\sigma(1), \dots, \sigma(i), \dots, \sigma(j), \dots, \sigma(n))$ the swap neighborhood consists of all $n(n-1)/2$ permutations $\sigma' = (\sigma(1), \dots, \sigma(j), \dots, \sigma(i), \dots, \sigma(n))$, with $1 \leq i < j \leq n$, that can be obtained from σ by swapping two jobs. A move that swaps job $\sigma(i)$ with $\sigma(j)$, and a move that swaps job $\sigma(k)$ with $\sigma(l)$ are said to be *independent* if $\max\{i, j\} < \min\{k, l\}$ or $\max\{k, l\} < \min\{i, j\}$.

The *dynasearch swap* neighborhood, introduced in [5], consists of all solutions that can be obtained by a series of pairwise independent swap moves. For example, given the permutation $\sigma = (1 \ 3 \ 4 \ 6 \ 2 \ 5 \ 8 \ 10 \ 9 \ 7 \ 12 \ 14 \ 13 \ 11)$, we can apply the 3 independent swap moves figured  to obtain the neighboring permutation $\sigma' = (1 \ 6 \ 4 \ 3 \ 2 \ 5 \ 7 \ 10 \ 9 \ 8 \ 12 \ 14 \ 11 \ 13)$. It is not

difficult to see that this neighborhood has size $2^{n-1} - 1$.

To search this exponential neighborhood in an efficient way, i.e. to find the best neighboring permutation among the $2^{n-1} - 1$ candidate permutations (i.e. we use steepest descent local search), a dynamic programming algorithm is used. We use a backward enumeration scheme in which jobs are appended to the beginning of the current partial sequence and are possibly swapped with jobs already scheduled in the partial sequence.

We note $(x)^+ = \max\{x, 0\}$ for any integer x . Let $\sigma = (\sigma(1), \dots, \sigma(i), \dots, \sigma(j), \dots, \sigma(n))$, be a permutation. We note (σ_i, t) the best possible way to schedule jobs $\sigma(i), \sigma(i+1), \dots, \sigma(n)$ by applying a series of independent swaps on the sub-permutation $(\sigma(i), \sigma(i+1), \dots, \sigma(n))$, assuming that the first job scheduled in that sub-permutation is scheduled at time t . We take only into account the total weighted tardiness of jobs $\sigma(i), \sigma(i+1), \dots, \sigma(n)$ and forget jobs $\sigma(1), \sigma(2), \dots, \sigma(i-1)$ when dealing with (σ_i, t) . We note $F(\sigma_i, t)$ the corresponding total weighted tardiness of jobs $\sigma(i), \sigma(i+1), \dots, \sigma(n)$ in the state (σ_i, t) . We shall put $(\sigma_{n+1}, t) = \emptyset$ and $F(\sigma_{n+1}, t) = 0$ for any time t to simplify the description of the algorithm below.

Now the state (σ_i, t) must be obtained either by appending the job $\sigma(i)$ in front of the state $(\sigma_{i+1}, t + p_{\sigma(i)}^t)$ or by appending the sequence $(\sigma(j), \sigma(i+1), \dots, \sigma(j-1), \sigma(i))$, obtained by swapping jobs $\sigma(i)$ and $\sigma(j)$, in front of the state (σ_{j+1}, t') for some job $i+1 < j \leq n$ and time t' (to be determined later).

We have for the first case $F(\sigma_i, t) = w_{\sigma(i)}(t + p_{\sigma(i)}^t - d_{\sigma(i)})^+ + F(\sigma_{i+1}, t + p_{\sigma(i)}^t)$. For the second case, let t_k be the starting time of the k -th scheduled job for $i \leq k \leq j$ after having swapped $\sigma(i)$ and $\sigma(j)$. By definition of $F(\sigma_i, t)$, $t_i = t$. Then since jobs $\sigma(i)$ and $\sigma(j)$ have been swapped, $t_{i+1} = t_i + p_{\sigma(j)}^{t_i}$. Finally, $t_k = t_{k-1} + p_{\sigma(k-1)}^{t_{k-1}}$ for $i+1 < k \leq j$. Thus we have

$$F(\sigma_i, t) = w_{\sigma(j)}(t_i + p_{\sigma(j)}^{t_i} - d_{\sigma(j)})^+ + \sum_{i < k < j} w_{\sigma(k)}(t_k + p_{\sigma(k)}^{t_k} - d_{\sigma(k)})^+ + w_{\sigma(i)}(t_j + p_{\sigma(i)}^{t_j} - d_{\sigma(i)})^+ + F(\sigma_{j+1}, t_j + p_{\sigma(i)}^{t_j}).$$

If $j = i+1$, the sum $\sum_{i < k < j}$ is empty.

Finally the dynamic programming algorithm can be stated as:

$$F(\sigma_n, t) = w_{\sigma(n)}(t + p_{\sigma(n)}^t - d_{\sigma(n)})^+$$

and

$$F(\sigma_i, t) = \min\{w_{\sigma(i)}(t_i + p_{\sigma(i)}^{t_i} - d_{\sigma(i)})^+ + F(\sigma_{i+1}, t_i + p_{\sigma(i)}^{t_i}), \min_{j, i < j \leq n} w_{\sigma(j)}(t_i + p_{\sigma(j)}^{t_i} - d_{\sigma(j)})^+ + \sum_{i < k < j} w_{\sigma(k)}(t_k + p_{\sigma(k)}^{t_k} - d_{\sigma(k)})^+ + w_{\sigma(i)}(t_j + p_{\sigma(i)}^{t_j} - d_{\sigma(i)})^+ + F(\sigma_{j+1}, t_j + p_{\sigma(i)}^{t_j})\}, \text{ for } 1 \leq i < n$$

We want to calculate $F(\sigma_1, 0)$. To implement the dynamic programming algorithm we have used a basic memoize technique: an array stores the values of F already computed in order to reduce the number of recursive calls. The optimal set of independent swaps can be retrieved by examining an array which stores, for each job j and each time t for which a value $F(\sigma_j, t)$ was computed, the position of $\sigma(j)$ in the state (σ_j, t) .

We shall assume that the processing times are bounded up by a constant p_{max} , i.e. $0 \leq p_{\sigma(j)}^t \leq p_{max}$, $\forall j, t$. Under this assumption, the time t at which the last job can start is bounded up by $(n-1)p_{max}$. There are $n(n-1)p_{max}$ states (σ_i, t) , each one can be computed in $\mathcal{O}(n^2)$ time (assuming that the previous required states have been already computed), leading to a total $\mathcal{O}(n^4 p_{max})$ time complexity. The space complexity is $\mathcal{O}(n^2 p_{max})$.

3 Experimental results

There is certainly a tradeoff between the benefice of using a large neighborhood in terms of the quality of local optima and the induced time increase, relative to a small neighborhood,

in order to search it. Our experiments were designed in order to determine if it was worth to spend more time exploring a larger neighborhood. To compare the performance of the standard and the dynasearch swap neighborhoods we have used multi-start local search (MLS).

In MLS with the dynasearch swap neighborhood, a fixed number of local search are performed, 10 in our case, and we retain the best local optima found out of these 10 local search. In MLS with the standard swap neighborhood, the number k of local search is not known *a priori* and it is a function of the time spent, say T , by the MLS with the dynasearch swap neighborhood, namely we perform local search until the total time spent is greater or equal to T . In this way we can fairly compare the two algorithms. In the sequel, MLS means multi-start local search with the swap neighborhood, and MDS means multi-start dynasearch, that is multi-start local search with the dynasearch swap neighborhood.

To speed-up the local search algorithm with the standard swap neighborhood, when a swap involving jobs $\sigma(i)$ and $\sigma(j)$, with $j > i$, is evaluated, only the portion in the right of the job $\sigma(j)$ is taken into account to calculate the cost of the new solution (we use additional arrays to record for each job its starting time, and for each position the partial cost induced by the current solution up to this position).

An instance of the problem $1 || p_j^t, idleness | \sum_j w_j T_j$ with n jobs is completely described by giving the $(1 + (n-1)p_{max})n$ processing times p_j^t with $1 \leq j \leq n$ and $0 \leq t \leq (n-1)p_{max}$, with the n weights and the n due dates for each job. We have chosen $p_{max} = 20$.

We have generated random instances of various types with sizes 30 and 40. For each job j an integer weight w_j is randomly uniformly generated in $[1, 10]$, whatever the type of the instance. In the first type of instances, each processing time p_j^t is independent of the others, and is chosen randomly in $[1, p_{max}]$. This method certainly leads to the most difficult instances, but from a practical point of view they are not realistic. In the second type of instances, each job j has an ideal starting time b_j and is penalized if its starts either too early or too late. Its processing time is given by $p_j^t = 1 + \min\{p_{max} - 1, \lfloor \frac{2 * |t - b_j|}{n-1} \rfloor\}$. In the third (resp. fourth) type of instances, the processing times are decreasing (resp. increasing) with the time, i.e. the latter a job starts, the shorter (resp. longer) its duration. More precisely, the processing time of a job j given that it starts at time t is $\max\{1, \lfloor p_{max} - r_j * t * \frac{p_{max}-1}{(n-1)p_{max}} \rfloor\}$ for the third type of instance, with r_j a real random number in $[0, 1]$, t in $\{0, 1, \dots, (n-1)p_{max}\}$, and $\min\{p_{max}, \lfloor 1 + r_j * t * \frac{p_{max}-1}{(n-1)p_{max}} \rfloor\}$ for the fourth type of instance.

The integer due dates were generated in a similar way than what it is usually done for the classical $1 || \sum_j w_j T_j$ problem: For each job j , an integer due date d_j is randomly generated in the interval $[P(1 - TF - RDD/2), P(1 - TF + RDD/2)]$ using a uniform distribution, with $P = \sum_{j=1}^n p_j$, $RDD \in \{0.2, 0.4, 0.6, 0.8, 1.0\}$ the relative range of due dates, and $TF \in \{0.2, 0.4, 0.6, 0.8, 1.0\}$ the tardiness factor. When the tardiness factor is close to 1 it means that the majority of jobs will be completed after their due dates and the problem is very constrained. The relative range of due dates indicates the variability of the due dates around their mean value. We adopted the same procedure, with $P = \sum_{j=1}^n \sum_{t=0}^{(n-1)p_{max}} p_j^t / ((n-1)p_{max} + 1)$ to take into account the time dependent processing times.

We have generated five instances for each of the 25 pairs of values of RDD and TF, which yields a total of 125 instances for each size and each type of instance. The experiments were performed on a Pentium III biprocessor-500Mhz. Average results (over 10 executions of MLS and MDS) are shown in Tables 1 to 8.

In Tables 1 and 2, for each TF and RDD values we have three entries. The first entry is the cost of the best local optimum found using MLS (or MDS), the second entry is the execution time in seconds, and the last entry is the number of local search performed for MLS in order to have the same execution time than MDS. Table 3 indicates the improvement in percentage on the cost of the solution found by MDS versus MLS.

We can see than despite a much less number of local search (10 against several hundreds), MDS performs better than MLS. The improvement can be very significant, for example for instances with $RDD=0.2$ and $TF=0.6$, we obtain the optimal solution with MDS, whereas MLS gives solutions with a cost of 20.1.

rdd\tf	0.2	0.4	0.6	0.8	1.0
0.2	0.0; 2.3; 366	0.0; 2.4; 122	20.1; 8.7; 217	1966.1; 23.2; 422	12939.8; 24.4; 390
0.4	0.0; 2.3; 271	0.0; 3.1; 130	2.3; 10.8; 231	2237.4; 23.9; 405	14287.1; 24.9; 397
0.6	0.0; 2.4; 231	0.0; 4.9; 187	0.9; 13.5; 265	3297.3; 24.6; 404	11723.3; 25.4; 399
0.8	0.0; 2.7; 186	0.0; 7.2; 235	65.6; 16.5; 306	4356.7; 23.9; 378	17836.4; 25.4; 391
1	0.0; 3.6; 208	0.0; 10.3; 280	1035.2; 20.0; 345	6970.6; 25.0; 380	18160.4; 26.6; 398

Table 1: size: 40, type: 1, multi-start local search

rdd\tf	0.2	0.4	0.6	0.8	1.0
0.2	0.0; 2.3; 10	0.0; 2.4; 10	0.0; 8.7; 10	880.3; 23.1; 10	11769.6; 24.3; 10
0.4	0.0; 2.3; 10	0.0; 3.1; 10	0.0; 10.8; 10	1245.7; 23.9; 10	13064.3; 24.9; 10
0.6	0.0; 2.4; 10	0.0; 4.9; 10	0.5; 13.5; 10	2507.8; 24.6; 10	10820.6; 25.4; 10
0.8	0.0; 2.7; 10	0.0; 7.2; 10	49.8; 16.5; 10	3895.1; 23.9; 10	17010.8; 25.4; 10
1	0.0; 3.6; 10	0.0; 10.3; 10	994.0; 19.9; 10	6567.3; 24.9; 10	17361.0; 26.5; 10

Table 2: size: 40, type: 1, multi-start dynasearch

rdd\tf	0.2	0.4	0.6	0.8	1.0
0.2	0	0	-100.0	-55.2	-9.0
0.4	0	0	-100.0	-44.3	-8.6
0.6	0	0	-44.4	-23.9	-7.7
0.8	0	0	-24.1	-10.6	-4.6
1	0	0	-4.0	-5.8	-4.4

Table 3: size: 40, type: 1

Due to space limitation we present only the percentage of improvement for the other types of instances with size 30 and 40 in Tables 4 to 8. We can see that in the overwhelming of cases, multi-start dynasearch gives better results. There is an exception however for type 4 instances. For them we obtain exactly the same cost with multi-start local search and dynasearch. This probably indicates that these instances are solved to optimality.

In the paper of Congram et al. [5], MDS gave only little improvement over MLS for the $1||\sum w_j T_j$ problem. The improvement was however significant for a more elaborated method called iterated local search (in which a new local search is started from a solution “close” to the previously found local optimum, instead of a randomly generated solution as it is the case for MLS and MDS). The explanation we give for the efficiency of MDS over MLS for the time-dependent scheduling problem we consider is the following one. In our case, starting from a solution σ , the swap of two jobs $\sigma(i)$ and $\sigma(j)$ ($i < j$) can lead to a solution σ' in which the swap of two jobs $\sigma(k)$ and $\sigma(l)$ ($i < j < k < l$ or $k < l < i < j$) leads to a solution σ'' with a lower cost than solution σ , whereas performing first the swap of jobs $\sigma(k)$ and $\sigma(l)$ on solution σ is not profitable, i.e. it increases the cost of the solution obtained. This situation is not encountered in the static scheduling problem $1||\sum w_j T_j$. Therefore, we take the benefice of a lookahead capability which is absent from the standard local search algorithms which are traditionally myopic in nature.

4 Conclusion and extensions

Our work is in the continuity of Congram et al. [5] which have introduced the dynasearch swap neighborhood for the $1||\sum_j w_j T_j$ problem. By introducing the time parameter inside the dynamic programming algorithm we obtain a pseudopolynomial algorithm in time and space, whereas their algorithm needed $\mathcal{O}(n^3)$ time and $\mathcal{O}(n)$ space, but we enlarge considerably the class of problems which can now be treated. We need not anymore to consider

rdd\tf	0.2	0.4	0.6	0.8	1.0
0.2	0	-3.2	-0.1	0.1	-0.0
0.4	0	-17.1	0.3	0.2	-0.1
0.6	0	9.5	0.4	-0.0	-0.0
0.8	0	0	0.9	0.0	0.0
1	0	3.5	0.5	-0.0	-0.0

Table 4: size: 40, type: 2

rdd\tf	0.2	0.4	0.6	0.8	1.0
0.2	-3.7	-0.4	0.0	0.0	0.0
0.4	-17.3	-1.4	0.0	0.0	0.0
0.6	0	-1.4	0.1	0.0	0.0
0.8	0	-1.5	0.1	-0.0	0.0
1	0	-2.0	0.0	0.0	-0.0

Table 5: size: 40, type: 3

rdd\tf	0.2	0.4	0.6	0.8	1.0
0.2	0	0	-100.0	-39.1	-2.5
0.4	0	0	0	-34.6	-3.8
0.6	0	0	-97.6	-6.2	-3.3
0.8	0	0	-15.1	-3.4	-1.3
1	0	0	3.1	-1.0	-2.6

Table 6: size: 30, type: 1

rdd\tf	0.2	0.4	0.6	0.8	1.0
0.2	0	-1.3	0.1	0.2	0.0
0.4	0	-15.4	1.0	-0.0	-0.0
0.6	0	-0.7	0.2	0.1	0.0
0.8	0	5.1	0.2	0.1	-0.0
1	0	1.3	0.3	0.0	-0.0

Table 7: size: 30, type: 2

rdd\tf	0.2	0.4	0.6	0.8	1.0
0.2	-3.7	-0.0	0.0	0.0	0.0
0.4	-13.5	-0.6	0.1	0.0	0.0
0.6	-7.0	-0.5	0.0	0.0	0.0
0.8	-100.0	-1.1	0.1	0.0	0.0
1	0	-0.1	0.0	0.0	-0.0

Table 8: size: 30, type: 3

problems in which the cost change between neighboring solutions depends only on the jobs between $\sigma(i)$ and $\sigma(j)$ which are swapped. Namely, in the $1 \parallel \sum_j w_j T_j$ problem when jobs $\sigma(i)$ and $\sigma(j)$ ($i < j$) are swapped, the jobs in the left of $\sigma(j)$ and in the right of $\sigma(i)$ have the same starting time than before the swap. For instance multiprocessor scheduling problems, in which a swap may involve jobs executed not only in the same processor, but also in different processors, could be treated in the same way, by introducing one time variable for each processor.

The experimental results have demonstrated the superiority of the dynasearch approach over the standard swap neighborhood for the $1 \parallel p_j^t, idleness \parallel \sum_j w_j T_j$ problem. More elaborate methods such as iterated local search [5] or metaheuristics such as simulated annealing and tabu search [15] would perhaps give better results, but we think that they should be in agreement with ours, namely it is worth to spend more time exploring the bigger dynasearch neighborhood. A reason for the strong improvement we have noticed for the multi-start local search may be the lookahead capability of the dynasearch neighborhood when it is applied to time-dependent scheduling problems. Enlarging the neighborhood would increase this lookahead capability, and therefore it is natural to wonder how big can we make the neighborhood until the benefit of having a large neighborhood is neutralized by the long time spent at each iteration to explore it. Insertion moves (consisting in taking a job and inserting it between two consecutive jobs) could be added to swap moves, resulting in a bigger neighborhood still searchable in pseudopolynomial time using the same dynasearch technique.

A promising way of improving dynasearch would be to use some kind of branch and bound inside the dynamic programming algorithm. We need a lower bounding function $L(\sigma_i, t)$ which gives a lower bound on the total weighted tardiness of the best sub-solution involving jobs $\sigma(i), \sigma(i+1), \dots, \sigma(n)$ given that the first job is scheduled at time t , and we need a third parameter c in $F(\sigma_i, t, c)$, which indicates the total weighted tardiness of the partial solution constructed so far (this has no incidence on the time complexity). If the lower bound indicates that the best solution we can obtain by performing independent swaps on jobs $\sigma(i), \sigma(i+1), \dots, \sigma(n)$ starting from time t is worst than a solution we have already obtained, then we need not to compute the state $F(\sigma_i, t)$.

For the time-dependent problem we have considered in this paper, a lower bound based on the relaxation of the $1 \parallel \sum_j w_j T_j$ to a transportation problem (see for example [13]) would have a time complexity of $\mathcal{O}(n^3)$, and therefore would perhaps be not competitive. We are now exploring this approach for other time dependent scheduling problems whose lower bounds can be obtained in a faster way.

References

- [1] B. Alidaee and N.K. Womer. Scheduling with time dependent processing times: Review and extensions. *Journal of the Operational Research Society*, 50(7):711–720, 1999.
- [2] E.J. Anderson, C.A. Glass, and C.N. Potts. Machine scheduling. In E.H.L. Aarts and J.K. Lenstra, editors, *Local search in combinatorial optimization*, pages 361–414. Wiley, 1997.
- [3] E. Angel and V. Zissimopoulos. On the quality of local search for the quadratic assignment problem. *Discrete Applied Mathematics*, 82:15–25, 1998.
- [4] E. Angel and V. Zissimopoulos. On the classification of NP-complete problems in terms of their correlation coefficient. *Discrete Applied Mathematics*, 99:261–277, 2000.
- [5] R.K. Congram, C.N. Potts, and S.L. van de Velde. An iterated dynasearch algorithm for the single-machine total weighted tardiness scheduling problem. Technical report, Faculty of Mathematical Studies, University of Southampton, U.K., december 1998.
- [6] H.A.J. Crauwels, C.N. Potts, and L.N. Van Wassenhove. Local search heuristics for the single machine total weighted tardiness scheduling problem. *INFORMS Journal on Computing*, 10:341–350, 1998.

- [7] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.
- [8] G. Gutin. Exponential neighbourhood local search for the traveling salesman problem. *Computers and Operations Research*, 26(4), 1999.
- [9] J. Hurink. Efficient calculation of a best neighbor for a one machine batching problem. Technical report, Osnabrücker Schriften zur Mathematik, Reihe P, No. 180, 1996.
- [10] M.Y. Kovalyov. On one machine scheduling to minimize the number of late items and the total tardiness. Technical report, Institute of Engineering Cybernetics, Academy of Sciences of Byelorussian SSR, Minsk, Byelorussia, 1991. Preprint N4.
- [11] E.L. Lawler. A “pseudopolynomial” algorithm for sequencing jobs to minimize total tardiness. *Annals of Discrete Mathematics*, 1:331–342, 1977.
- [12] E.L. Lawler. A fully polynomial approximation scheme for the total tardiness problem. *Operations Research Letters*, 1:207–208, 1982.
- [13] M. Pinedo. *Scheduling: Theory, Algorithms and Systems*. Prentice Hall, 1995.
- [14] C.N. Potts and L.N. Van Wassenhove. A branch and bound algorithm for the total weighted tardiness problem. *Operations Research*, 33:363–377, 1985.
- [15] C.R. Reeves, editor. *Modern heuristic techniques for combinatorial problems*. Blackwell Scientific Publications, 1993.
- [16] N.N. Doroshko V.I. Sarvanov. The approximate solution of the travelling salesman problem by a local search algorithm that searches neighborhoods of exponential cardinality in quadratic time (in russian). *Software: Algorithms and Programs*, 31:8–11, 1981. Mathematical Institute of the Belorussian Academy of Sciences, Minsk.
- [17] N.N. Doroshko V.I. Sarvanov. The approximate solution of the travelling salesman problem by a local search algorithm with scanning neighborhoods of factorial cardinality in cubic time (in russian). *Software: Algorithms and Programs*, 31:11–13, 1981. Mathematical Institute of the Belorussian Academy of Sciences, Minsk.