Available online at www.sciencedirect.com
SCIENCE DIRECT®
Operations Research Letters 34 (2006) 591–598
Operations Research Letters
www.elsevier.com/locate/orl

Dynasearch for the earliness–tardiness scheduling problem with release dates and setup constraints

Francis Sourd*

Université Pierre et Marie Curie-CNRS-LIP6 4, place Jussieu-75252 Paris Cedex 05, France

Received 21 January 2004; accepted 30 June 2005
Available online 27 October 2005

Abstract

A large dynasearch neighborhood is introduced for the one-machine scheduling problem with sequence-dependent setup times and costs and earliness–tardiness penalties. Finding the best schedule in this neighborhood is NP-complete in the ordinary sense but can be done in pseudo-polynomial time. We also present experimental results.

Keywords: Earliness–tardiness scheduling; Very large neighborhood search

1. Introduction

Local search algorithms are widely used as a practical approach for solving combinatorial optimization problems. Starting with a feasible solution, these algorithms iteratively try to improve the *current* solution by searching a better solution in the *neighborhood* of the current solution until a *local minimum* is found. The efficiency of these algorithms critically depends on the definition of the neighborhood: with a larger neighborhood, the quality of the local minimum is generally better but the computation time required to explore the neighborhood is longer. So, in practice, large neighborhood are not useful unless it can efficiently be explored. Such an approach is referred to as a *very large-scale neighborhood search technique*, this term is popularized in the survey by Ahuja et al. [1].

In the scheduling literature, Congram et al. [2] present a large neighborhood—called *dynasearch*—for the single-machine total weighted tardiness scheduling problem. This neighborhood can be seen as the composition of an arbitrary number of independent swap operations. Its size is exponential but a dynamic programming algorithm is proposed to compute the optimal schedule in the neighborhood in polynomial time. Based on this work, Grosso et al. [3] propose an enhanced dynasearch neighborhood that combines the swap operator used in [2] with a second "extract and reinsert" operator.

In this paper, we extend this approach to a more general and a more practical problem that appears in particular in manufacturing scheduling problems, and that is called L2-STC-NCOS in MASCLIB, the library of manufacturing scheduling problems from

* Fax: +1 +33 1 44 27 70 00.
E-mail address: francis.sourd@lip6.fr (F. Sourd).
URL: http://www-poleia.lip6.fr/~sourd.

0167-6377/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.orl.2005.06.005

industry proposed by ILOG [4]. More precisely, our problem deals with setup times and setup costs, release dates, deadlines and a general end-time-dependent costs that can for example model the commonly used earliness–tardiness penalties. The reader is referred to [4,5] for a presentation of the relevance of this problem and for related references in the just-in-time and setup scheduling literature. Mathematical programming formulations of the problem, a branch-and-bound algorithm and an efficient heuristic procedure are also presented in [5].

Section 2 introduces the scheduling problem. The associated dynasearch problem, which consists in finding the optimal solution in the so-called dynasearch neighborhood, is shown to be NP-complete in the ordinary sense. Section 3 is devoted to the presentation of a pseudo-polynomial algorithm to search the neighborhood and experimental results are eventually presented in Section 4.

2. Problem definition and complexity

A single machine has to process n tasks $J_1, \ldots, J_n$ such that at most one task is performed at any time. Preemption of tasks is not allowed but idle time can be inserted between two tasks. Each task J_i has a processing time p_i and belongs to a *group* (or *family*) $g_i \in \{1, \ldots, q\}$ (with $q \leqslant n$). *Setup* or *changeover* times and costs, which are given as two $q \times q$ matrices, are associated to these groups. This means that in a schedule where J_j is processed immediately after J_i, there must be a setup time of at least $s(g_i, g_j)$ time units between the completion time of J_i, denoted by C_i, and the start time of J_j, which is $C_j - p_j$. During this setup period, no other task can be performed by the machine and we assume that the cost of the setup operation is $c(g_i, g_j) \geqslant 0$. We assume that there is no setup time and no setup cost between tasks belonging to the same group (that is $s(g, g) = 0$ and $c(g, g) = 0$) and the setup matrices satisfy the triangle inequalities (that is $s(g, g') \leqslant s(g, g'') + s(g'', g')$ and $c(g, g') \leqslant c(g, g'') + c(g'', g')$). A machine is said to be *in state* g as soon as the setup operation moving it into state g is finished and it remains in state g until another setup operation starts. Machine state is undefined during setup operations. Such setup times and costs are said to be *sequence-dependent* because they depend on both g and g'. When $s(g, g')$ and $c(g, g')$ $(g \neq g')$ can be expressed as a function in only the state g', setups are said tobe *sequence-independent*.

In addition to the setup constraints and costs, a cost $f_i(C_i)$ is due for each task. Such a cost depends on the completion time of J_i and is called *punctuality cost* in this paper. The idea is that the cost is low during the periods satisfying the manufacturer and the consumer and higher otherwise. More details about the possible use of such a cost function is given in [6]. Clearly, punctuality cost is a generalization of the *deviation* cost function $\mathrm{ET}_i(C_i) = \max(\alpha_i(d_i - C_i), \beta_i(C_i - d_i))$ used in earliness–tardiness scheduling. Note that release dates can easily be integrated in a function f_i.

In this paper, we are going to assume that all the functions f_i are piecewise linear, which is a very common and efficient way to represent and approximate general functions in computer science. For simplicity, we assume that the cost functions are continuous. The number of segments of f_i will be denoted by $\|f_i\|$.

The problem is to minimize the sum of the punctuality and setup costs. In this paper, we assume that all the numerical time-related values (setup times, processing times, abscissas of piecewise linear functions) are integer. The problem is NP-complete in the strong sense—observe that it generalizes the one-machine earliness–tardiness scheduling problem. We now formally define the *dynasearch neighborhood*.

Let us consider an *initial sequence*. The tasks can be renumbered such that the sequence is $(J_1, \ldots, J_n)$. Following [3], we define three operators which, given a pair of indices $i < j$, act on the sequence $\sigma = \alpha J_i \beta J_j \gamma$ as follows:

- $\mathrm{SWAP}_{ij}(\alpha J_i \beta J_j \gamma) = \alpha J_j \beta J_i \gamma$. When β is the empty sequence, the operator is also known as the *adjacent pairwise interchange*.
- $\mathrm{EBSR}_{ij}(\alpha J_i \beta J_j \gamma) = \alpha J_j J_i \beta \gamma$. EBSR stands for *extraction and backward-shifted reinsertion*.
- $\mathrm{EFSR}_{ij}(\alpha J_i \beta J_j \gamma) = \alpha \beta J_j J_i \gamma$. EFSR stands for *extraction and forward-shifted reinsertion*.

Two operators acting on position pairs $i < j$ and $k < l$ are called *independent* if $j < k$ or $l < i$. The *dynasearch sequence neighborhood* is the set of sequences that can be derived from the initial sequence through the application of a set of independent SWAP, EBSR and EFSR operators. Eventually, the *dynasearch neighborhood* is the set of schedules

whose task sequences are in the sequence neighborhood. As the objective criterion of the problem is not a regular one, there is not an obvious one-to-one mapping between a sequence and a schedule—in other words, for a given sequence, the earliest schedule may not be the optimal one.

The *dynasearch problem* is, given a schedule or a sequence σ, to find a schedule in the dynasearch neighborhood of σ with minimal cost. The decision variant of this minimization problem is to find a schedule with a cost less than some given value K.

We first illustrate the fact that the solution of the dynasearch problem cannot be found by iteratively applying the operators on which the neighborhood is based. For simplicity, we consider an example where only the SWAP operator is possible. The initial sequence is $\sigma = (J_1, J_2, J_3, J_4)$, there is no punctuality costs so that the setup costs are the only costs to be considered. We have $g_1 = g_4 = 1$, $g_2 = 2$ and $g_3 = 3$. The setup costs are symmetric with $c(1, 2) = c(1, 3) = 10$ and $c(2, 3) = 1$. The cost of the initial sequence is 21. The costs of $\mathrm{SWAP}_{12}(\sigma)$ and $\mathrm{SWAP}_{34}(\sigma)$ are equal to 30 and the cost of $\mathrm{SWAP}_{14}(\sigma)$ and $\mathrm{SWAP}_{23}(\sigma)$ are 21 so that σ is a local minimum with respect to the SWAP operator. However, if the independent SWAP_{12} and SWAP_{34} are simultaneously processed, the schedule $\sigma' = (J_2, J_1, J_4, J_3)$ with cost 20 is obtained, which is the optimum of the dynasearch problem. Then, $\mathrm{SWAP}_{13}(\sigma') = (J_4, J_1, J_2, J_3)$ gives a solution with cost 11, which is the optimum of our problem.

Theorem 1. *The dynasearch problem is NP-complete in the ordinary sense even if each punctuality cost function is the weighted tardiness indicator* $w_i T_i = w_i \max(0, C_i - d)$ *for some common due date* d.

Proof. The problem is clearly in NP. We transform PARTITION (which is NP-complete) to the dynasearch decision problem. Let an arbitrary instance of PARTITION given by the finite set $A = \{1, 2, \ldots, k\}$ and the size $s_a \in \mathbb{Z}^+$ for each item $a \in A$. Let $K = \sum_{a \in A} s_a/2$. We are going to build an instance of the dynasearch problem whose cost is less than or equal to K if and only if there exists a bipartition of A.

The scheduling problem has $n = 2k$ tasks $J_1, J_2, \ldots, J_n$ sequenced in this order. For any $a \in A$, the two jobs J_{2a-1} and J_{2a} are related to item a of PARTITION (J_i is related to $\lceil i/2 \rceil$). The main idea of the reduction is to partition A into B and $A \setminus B$ with $B = \{a | C_{2a-1} < C_{2a}\}$ where C_{2a-1} and C_{2a} are, respectively, the completion time of J_{2a-1} and J_{2a} in a schedule of the dynasearch neighborhood.

The processing time of any task J_i is 0. The punctuality cost function is $f_i(C_i) = (K+1)\max(0, C_i - K)$ for each task. There are n setup groups, let $g_i = i$ be the group of J_i. Let $c(2a-1, 2a) = s(2a, 2a-1) = s_a$ and $s(2a-1, 2a) = c(2a, 2a-1) = 0$. Otherwise, that is if $\lceil i/2 \rceil \neq \lceil j/2 \rceil$,

$$c(i, j) = s(i, j) = \begin{cases} 0 & \text{if } i < j, \\ K+1 & \text{otherwise.} \end{cases}$$

We first prove that any schedule with cost less than or equal to K is in the dynasearch neighborhood. Since the cost functions are nondecreasing, we can assume that the schedule has no idle time so that its makespan is equal to the length of all the setup times, which is an integer. As the cost is less than K, the makespan must be less than or equal to K. The setup time table shows that, if $a < a'$, any job related to a must be sequenced before any job related to a'. So, the sequence differs from the initial sequence by a number of swaps between adjacent jobs related to the same item. These swap operations are all independent so that this schedule is in the dynasearch neighborhood. The length of this schedule, which is less than K, is clearly $\sum_{a \notin B} s(2a, 2a-1) = \sum_{a \notin B} s_a$ and the total cost is equal to the setup costs which are $\sum_{a \in B} s(2a, 2a-1) = \sum_{a \in B} s_a$.

Therefore, there exists a schedule with cost at most K in the dynasearch neighborhood if and only if there is a bipartition of A. □

In the next section, we show that the optimal schedule of the dynasearch problem can be computed in pseudo-polynomial time.

3. Dynasearch algorithm

Based on the approach introduced by [6], let us define $\Sigma_k^g(t)$ as the cost of optimally scheduling the tasks $J_1, \ldots, J_k$ subject to the constraints:

- the sequence is in the dynasearch neighborhood of $(J_1, \ldots, J_k)$,

- the group of the last task (i.e. the task in position k) is g,
- the last task completes before time t.

Clearly, Σ_k^g is a real function. The value $\Sigma_k^g(t)$ is set to ∞ when there is no feasible schedule ending before t (for instance, if $t < \sum_{i=1}^{k} p_i$). We also define $\Sigma_0^g(t)$ as the cost of setting up the machine to state g before t without scheduling any task.

3.1. Recurrence equation

For $k \in \{0, 1, \ldots, n\}$, let Σ_k be the vector of q functions $(\Sigma_k^1, \ldots, \Sigma_k^q)$. Intuitively, the gth coordinate $\Sigma_k^g(t)$ of the vector $\Sigma_k(t)$ represents the cost of scheduling the jobs $J_1, \ldots, J_k$ in a dynasearch order, such that they end before t and the machine is in state g at the end. If none of the jobs $J_1, \ldots, J_k$ is in group g, there is no sequence in the dynasearch neighborhood such that the last job is in group g and therefore, the cost $\Sigma_k^g(t)$ is necessarily infinite for any t.

Let us now consider the vector $\Sigma = \Sigma_{k-1}$ and we first consider the sequences in the dynasearch neighborhood ending with J_k. We denote by $\Sigma \cdot J_k$ the vector of q functions representing the optimal solution according to the two parameters g and t. Since J_k is by definition the last job, the gth (with $g \neq g_k$) coordinate of $\Sigma \cdot J_k$ is infinite for any t. The g_kth coordinate is computed by considering the possible groups and completions times of the job preceding J_k:

$$(\Sigma \cdot J_k)^{g_k}(t) = \min_{1 \leqslant g \leqslant q} \min_{t' < t} \left(\Sigma^g(t' - p_k - s(g, g_k)\right) + f_k(t') + c(g, g_k).$$

This expression is directly derived from the one of Sourd [6].

Fig. 1 shows that Σ_k can be recursively derived from $\Sigma_0, \Sigma_1, \ldots, \Sigma_{k-1}$. To understand the expression, we consider the sequence that correspond to the schedule yielding $\Sigma_k^g(t)$. This sequence has been obtained through a set of independent operators among SWAP, EBSR, EFSR. If J_k is the last operation of the sequence, then the cost of the schedule is clearly given by the first line in expression of Fig. 1. Otherwise, that is the position of J_k which has been modified by a SWAP, EBSR or EFSR operator, respectively, then

$$\Sigma_k = \min \left\{ \begin{array}{l} \Sigma_{k-1} \bullet J_k \\ \min_{1 \leq i \leq k} (\Sigma_{i-1} \bullet J_k \bullet J_{i+1} \bullet \cdots \bullet J_{k-1} \bullet J_i) \\ \min_{1 \leq i \leq k} (\Sigma_{i-1} \bullet J_k \bullet J_i \bullet \cdots \bullet J_{k-1}) \\ \min_{1 \leq i \leq k} (\Sigma_{i-1} \bullet J_{i+1} \bullet \cdots \bullet J_k \bullet J_i) \end{array} \right\}$$

Fig. 1. Computing Σ_k.

the cost of the schedule is given by the second, third or fourth line in Fig. 1, respectively.

The cost of the best schedule in the dynasearch neighborhood is finally given by $\min_{1 \leqslant g \leqslant q} \min_{t>0} \Sigma_n^g(t)$. Clearly, all the functions Σ_k^g are piecewise linear (see also [6]) and, since the processing times and the setup times are all integer, the abscissas of all the breakpoints of these functions are integer. This proves that the number of breakpoints is bounded by any constant T which is an upper bound on the makespan of the optimal schedule. For example, we can choose T as the sum of all the processing times and of $n \max_{gg'} s(g, g')$ and of the largest abscissa of all the cost functions f_i. This proves that the number of segments of the functions Σ_k^g is pseudo-polynomial in the input of the problem so that the dynasearch problem is also pseudo-polynomial. More precisely, computing $\Sigma \cdot J_k$ requires $O(qT)$ time [6] so that the total computation time is in $O(n^2 qT)$.

We observe that the dynamic program can be solved either

- by storing all the values $\Sigma_k^g(t)$ for $k \in \{1, \ldots, n\}$, $g \in \{1, \ldots, q\}$, $t \in \{1, \ldots, n\}$ in an array of size $O(nqT)$ or
- by storing the $O(nq)$ piecewise linear functions as the lists of their segments (see [6]).

In practice, the second option is far more efficient and we adopt this approach in the rest of the paper.

3.2. Backtracking

As the dynasearch procedure is to be iterated, retrieving the sequence that yields the optimal schedule in the dynasearch neighborhood is a very necessary step of the algorithm. Because the states of dynamic program are stored by the means of piecewise linear functions, the classical techniques used in dynamic programming must be adapted.

Clearly, the completion time of the last task in the optimal schedule can be set as the minimal t among

all the pairs (g, t) minimizing $\Sigma_n^g(t)$. Let $(g(n), t(n))$ denote the corresponding pair, $g(n)$ is the group to which this last task belongs, but which is precisely the task? To answer this question, we must store in the segments of the functions Σ_k^g some information that records how the segment was created. The expression in Fig. 1 shows that each function Σ_k^g is the minimum function of a finite set S of piecewise linear functions. So, each segment of Σ_k^g also belongs to at least one function of S.

So, let us specify how the segments are marked when computing Σ_k. Let us first consider the segments created by $\Sigma_{k-1} \cdot J_k$, which requires the computation of at most q piecewise linear functions. For each $g \in \{1, \ldots, q\}$, all the segments of the function $t \mapsto \min_{t'<t} \left(\Sigma_{k-1}^g(t' - p_k - s(g, g_k)) + f_k(t')\right) + c(g, g_k)$ are marked with tag (NOP, g). The first element NOP in the pair indicates that the last element (job J_k) is not concerned with any operator, so that the computation of Σ_k is derived from Σ_{k-1}, the second element g specify that this function is related to the gth coordinate of Σ_{k-1}, that is the machine was previously in state g.

Similarly, the segments of the g_ith coordinate of $\Sigma_{i-1} \cdot J_k \cdot J_{i+1} \cdot \cdots \cdot J_{k-1} \cdot J_i$, are marked with tag ($\text{SWAP}_{ik}, g$). SWAP_{ik} indicates that the sequence between J_i and J_k is modified by the SWAP operator so that the function is derived from Σ_{i-1} and more precisely its gth coordinate. Similar tags are used for the EBSR and EFSR operators.

When backtracking the dynamic program, we have to fix the completion time of all the jobs. We have already seen how to fix the completion time $t(n)$ of the last task. The name of the last task is revealed by the tag of the segment yielding $\Sigma_n^{g(n)}(t(n))$.

If the tag is (NOP, g), then the last job is unchanged so it is J_n. Since J_n completes at $t(n)$, the previous job must complete before $\bar{t}(n-1) = t(n) - p_n - s(g, g(n))$ and belongs to the group $g(n-1) = g$. So we can go on the backtracking procedure by determining the completion time $t(n-1)$ of the task in position $n-1$, which is the earliest time $t \leqslant \bar{t}(n-1)$ that minimizes $\Sigma_{n-1}^{g(n-1)}$ on $[0, \bar{t}(n-1)]$.

If the tag is (SWAP_{in}, g), we similarly know that the backtracking procedure must go on with the function $\Sigma_{i-1}^{g(i-1)}$ with $g(i-1) = g$ but we must also calculate the completion time $t(i-1)$ of the task in position $i-1$ in the new sequence. A *local* dynamic program is necessary for that. It first recomputes $\Sigma'_i = \Sigma_{i-1} \cdot J_n$, then $\Sigma'_{i+1} = \Sigma_{i-1} \cdot J_n \cdot J_{i+1}$ and so on until $\Sigma'_{n-1} = \Sigma_{i-1} \cdot J_n \cdot J_{i+1} \cdot \cdots \cdot J_{n-1}$. As the last operation in the new optimal sequence is J_i, we have that the penultimate task—which is J_{n-1} if $i < n-1$ and J_n if $i = n-1$—must end before $\bar{t}(n-1) = t(n) - p_i - s(g_{n-1}, g_i)$. So, the completion time $t(n-1)$ of the penultimate task is the earliest time before $\bar{t}(n-1)$ such that Σ'_{n-1} is minimum. Similarly, $t(n-2), \ldots, t(i)$ are recursively computed. Then, we know that the job in position $i-1$ must end before $\bar{t}(i-1) = t(i) - p_n - c(g, g_n)$ and thus $t(i-1)$ is derived from the analysis of Σ_{i-1}^g on the interval $[0, \bar{t}(i-1)]$.

The cases where the tag is (EBSR_{in}, g) or (EFSR_{in}, g) are solved similarly. The backtracking procedure is iterated until all the completion times are computed.

4. Experimental results

In order to evaluate the behavior of the dynasearch neighborhood, we compared it to a simple descent algorithm based on the *union* of the three neighborhoods SWAP, EBSR and EFSR—by contrast, the dynasearch neighborhood is seen as the *composition* of these neighborhoods. This algorithm is described in [5] and is used to find an initial feasible solution in the branch-and-bound procedure. Even if a single descent only finds a local optimum, the iteration of several descent procedure from randomly sequenced tasks is shown to be an efficient search algorithm, at least for small instances whose optimality can be computed. Indeed, when the algorithm is run n times the (global) optimum is found at least once for 95.2% of the 2160 instances with $n \in \{10, 12, 15\}$ tasks, the mean deviation is 0.08% and the maximal deviation is less than 10%.

With regard to the dynasearch algorithm, preliminary tests have shown that finding the best schedule in the dynasearch neighborhood is quite time-consuming. So, a descent algorithm that would run the dynasearch procedure at each step is not efficient at all. Therefore, in the tests presented below, the dynasearch descent procedure is started with the local optimum found by the simple descent procedure.

We used the test generator presented in [5] to create instances with $n \in \{30, 60, 100\}$ tasks and $q \in \{3, 10\}$

setup groups. In order to limit the complexity of the result tables, we fixed the other parameters which, according to preliminary tests, are not very significant for the computation times and solution quality:

- The *average tardiness* τ is equal to 0.8 and the *range factor* is set to 0.2.
- The *release date factor* (which is null in [5]) is set to 0.5. It means that the release dates are generated from the uniform distribution $[0, 0.5 \sum p_i]$.
- The setup times and costs are generated from the uniform distribution [50, 100].

For each pair (n, q), five instances were generated so that the benchmark set contains 30 different instances. Both algorithms are implemented in C++ and were run on a 1 GHz personal computer under the Windows 2000 operating system. When possible the dominance rules presented in [2,3] have been generalized for our problem in order to speed up the computation. For each instance, the simple descent algorithm was run 20 times starting with a random sequence and the dynasearch descent algorithm was run starting with each one of the 20 local optima found by the simple descent.

Table 1 compares the simple descent algorithm and the dynasearch-based algorithm. Each line of the table reports the behavior of the two algorithms for an instance, whose size (n, q) is indicated in the first two columns. The third column represents the best known solution for the instance, which is the solution found by the dynasearch procedure. The three columns related to the simple descent algorithm respectively reports the deviation between the best solution found by the algorithm (after 20 descents) and the best known solution, the mean deviation (for the 20 runs) and the mean running time. For the dynasearch algorithm, there is no "best" column because the best solution is by definition the solution of this algorithm.

These preliminary tests have shown that the dynasearch procedure generally improves the solution found by the simple descent algorithm. However, the computation times are long so that this neighborhood must be used "moderately". It cannot be shown by the table, but we observed that the best solutions found by the dynasearch algorithm generally correspond to a good solution of the descent procedure.

The second preliminary tests are about the usefulness of the operators EBSR and EFSR. For the weighted tardiness problems, Grosso et al. [3] show that the usage of EBSR and EFSR significantly improves the performance of the dynasearch procedure. We tested whether the same conclusion can be drawn for the earliness–tardiness problem with setups by running "limited" versions of our algorithms in which only the SWAP operator is active. Results for the five instances with $n = 60$ and $q = 10$ are presented in Table 2. Even if computation times are slightly decreased, performance is greatly unimproved: the optimum is never found and the deviation between the best solution found by either the simple descent procedure or the dynasearch algorithm is significantly larger. This result shows that combining SWAP neighborhoods with EBSR and EFSR neighborhoods is critically important to produce an efficient search algorithm.

Based upon these preliminary tests, a final algorithm was built. This algorithm iteratively executes the descent procedure (grounded on the three operators SWAP, EBSR and EFSR). If the solution found by this procedure is not worse than the best solution found so far plus a threshold of $\rho = 3\%$ then the dynasearch is run. Otherwise, the algorithm simply goes on with another descent procedure. The algorithm stops when the time limit is reached and returns the best solution found. Table 3 compares the performance of this algorithm with the iterated descent procedure. The time limit of both algorithms is fixed to 4 min, which means that the iterated descent procedure runs more descents than its dynasearch variant. For each 100-job instance, the deviation (from the best solution found in all our experiments) reported in the table are the minimum, maximum and average deviations for 10 runs. For both algorithms, the mean number of calls to the descent procedure is indicated in the column "desc". For the dynasearch-based algorithm, the number of times that a dynasearch neighborhood is explored and the total time spent in the exploration are, respectively, reported in the columns "ndyn" and "tdyn".

The average deviation is better for the dynasearch-based algorithm, but the difference with the descent algorithm is finally quite small. More interestingly, since the minimum deviation of the dynasearch algorithm is significantly better, we can conclude that the dynasearch algorithm is more likely to find a very good solution, especially when the allowed CPU time becomes larger. It was observed in Table 1 that, when

Table 1
Experimental results for SWAP, EBSR and EFSR-based algorithms

n	q	Best	Local search			Dynasearch	
		solution	Best	Avg	Time (s)	Avg	Time (s)
30	3	33866	0.00	6.07	0.06	4.88	0.15
		27742	0.00	2.93	0.06	2.20	0.16
		33586	0.00	5.47	0.06	3.44	0.18
		39235	0.00	7.29	0.06	7.07	0.14
		40372	0.00	6.57	0.06	6.53	0.14
	10	46742	0.00	3.97	0.05	3.87	0.19
		48510	1.30	9.73	0.05	8.44	0.20
		58859	0.00	7.43	0.06	6.97	0.22
		53429	0.00	2.72	0.05	1.77	0.28
		65337	1.15	7.63	0.06	6.20	0.23
60	3	133702	0.00	5.85	0.80	5.81	1.50
		102482	0.00	4.26	0.71	4.23	1.44
		152685	0.00	2.98	0.70	2.50	1.35
		139446	0.00	3.56	0.61	3.38	1.62
		173338	0.39	5.60	0.60	5.47	1.24
	10	178464	0.00	5.94	0.58	4.99	1.88
		196661	0.00	7.05	0.57	5.71	2.18
		185427	1.56	6.62	0.52	5.27	2.92
		165989	0.00	6.98	0.56	6.46	1.80
		157513	0.57	8.20	0.57	5.70	2.93
100	3	356027	0.24	4.08	4.01	3.85	9.48
		340291	0.00	4.91	4.82	4.59	10.46
		316728	0.00	5.57	4.65	5.05	8.24
		322797	0.00	6.73	4.35	6.56	8.98
		330573	0.00	6.77	4.61	5.94	13.55
	10	504943	1.83	5.42	3.28	4.27	17.72
		436167	0.55	4.68	3.47	3.50	18.01
		509439	0.00	4.28	2.78	2.74	16.04
		468147	3.36	8.25	3.55	6.57	17.02
		415678	2.62	6.25	3.38	5.07	16.40

Table 2
Experimental results for SWAP-based algorithms

n	q	Best	Simple descent			Dynasearch		
			Best	Avg	Time (s)	Best	Avg	Time (s)
60	10	178464	10.12	20.86	0.22	6.40	14.92	1.60
		196661	16.13	22.62	0.19	10.68	17.33	1.53
		185427	8.93	18.27	0.21	8.35	14.95	1.45
		165989	15.41	22.35	0.22	12.81	19.97	1.19
		157513	11.71	24.34	0.21	8.75	21.64	1.22

$n = 100$ and $q = 10$, the local optimum of the descent procedure is more likely to be improved by the dynasearch neighborhood. In our final algorithm, when the dynasearch procedure finds a better solution, the procedure is iterated. It is the reason why the number of dynasearch explorations (and the time spent in the

Table 3
Average performance after 4 CPU minutes

n	q	Best	Simple descent				Dynasearch					
			Min	Max	Avg	desc	Min	Max	Avg	desc	ndyn	tdyn (s)
100	3	351849	0.27	1.30	0.75	51.8	0.26	1.51	0.78	39.4	28.2	68.5
		340125	0.05	0.46	0.28	42.6	0.00	0.80	0.18	35.2	18.8	41.2
		311007	0.00	3.48	1.59	50.8	0.02	2.96	1.59	39.4	22.4	49.0
		319782	0.93	3.01	2.06	51.1	0.00	3.02	1.39	40.0	20.8	45.0
		330573	0.00	1.15	0.12	45.8	0.00	1.02	0.16	42.2	8.2	18.0
		Average	0.25	1.88	0.96	48.4	0.06	1.86	0.82	39.2	19.7	44.3
100	10	491673	1.26	4.37	2.95	62.0	0.00	4.76	2.13	37.0	28.6	91.4
		414991	1.68	6.06	3.15	59.4	1.15	5.90	2.98	40.6	25.8	78.0
		493229	0.36	2.10	1.28	63.6	0.28	3.37	1.71	46.0	29.6	70.0
		481898	0.67	4.37	2.77	58.1	0.00	4.17	2.31	41.2	21.4	72.2
		411390	0.46	3.17	1.39	61.8	0.00	3.74	1.52	35.8	35.8	106.8
		Average	0.88	4.02	2.31	61.0	0.29	4.39	2.13	40.1	28.2	83.7

dynasearch procedure) are greater in the case $n = 100$ and $q = 10$.

Therefore, the real key factor of the good performance of our algorithms is the combination—union or composition—of the three neighborhoods SWAP, EBSR and EFSR. But we have also shown that the dynasearch procedure is able to improve the solution found by the simple descent algorithm so that it can be regarded in practice as a good way to escape from a local optimum.

Acknowledgements

The author is grateful to an anonymous referee for helpful comments that improved the presentation of the theoretical and experimental results.

References

[1] R.K. Ahuja, Ö. Ergun, J.B. Orlin, A.P. Punnen, A survey of very large-scale neighborhood search techniques, Discrete Appl. Math. 123 (2002) 75–103.
[2] R.K. Congram, C.N. Potts, S.L. van de Velde, An iterated dynasearch algorithm for the single-machine total weighted tardiness scheduling problem, INFORMS J. Comput. 14 (2002) 52–67.
[3] A. Grosso, F. Della Croce, R. Tadei, An enhanced dynasearch neighborhood for the single-machine total weighted tardiness scheduling problem, Oper. Res. Lett. 32 (2004) 68–72.
[4] W. Nuijten, T. Bousonville, F. Focacci, D. Godard, C. Le Pape, Towards a real-life manufacturing scheduling problem and test bed, Proceedings of PMS'04, 2004, pp. 162–165. ⟨http://www2.ilog.com/masclib⟩.
[5] F. Sourd, Earliness–tardiness scheduling with setup considerations, Comput. Oper. Res. 32 (2005) 1849–1865.
[6] F. Sourd, Optimal timing of a sequence of tasks with general completion costs, European J. Oper. Res. 165 (2005) 82–96.