Bachelor Thesis Econometrie en Operationele Research

Analyzing Different Dynasearch Algorithms for the Single-Machine Total Weighted Tardiness Scheduling Problem

Emiel Feitsma • ID number: 411380

Supervisor: Thomas Breugem

Second assessor: Twan Dollevoet

In 2002, Congram, Potts and Van de Velde introduced dynasearch, a new competitive local search algorithm for the single-machine total weighted tardiness scheduling problem. At the time, an iterated dynasearch algorithm outperformed the state-of-the-art tabu search algorithm of Crauwels et al. (1998). In this thesis, we compare the basic dynasearch algorithm of Congram et al. (2002) with both existing and new extented versions of dynasearch, and with the breakout dynasearch algorithm of Ding et al. (2016).

July 2, 2017

Contents

1 Introduction

A descent algorithm is a simple and practical type of local search heuristic for solving computationally hard optimization problems. The main idea of a descent algorithm is to use a neighborhood structure to improve a given feasible solution, by performing a series of transformations or *moves*. A neighborhood of a current solution is defined by all the possible solutions that can be generated by a single move. The algorithm repeatedly selects a better solution in the neighborhood to be the new current solution, until no better solution exists.

We discuss two types of *traditional* descent algorithms; *first-improve* and *best-improve*. With first-improve descent, the current solution is replaced with the first better solution found in the neighborhood, while with best-improve descent the current solution is replaced with the best solution in the neighborhood.

With a single run of traditional descent, it is unlikely that the global optimum is found. We therefore employ two well-known approaches for improving the solution quality, namely a multi-start and an iterated approach. With a multi-start approach, several random starting solutions are generated, to perform different independent runs of descent, after which the best of the resulting solutions is selected. The second approach is known as iterated descent, which uses previous local optima to generate new starting solutions.

First-improve and best-improve descent algorithms can be used to search neighborhoods of polynomial size for a better or best solution. However, Congram et al. (2002) introduced *dynasearch*, an algorithm that searches a neighborhood of exponential size, reducing the chance of getting stuck in poor local optima. Dynamic programming in a local search algorithm is used, so that the exponential sized neighborhood can be searched in polynomial time. A basic dynasearch algorithm uses the same type of neighborhood as the traditional descent algorithms - a swap neighborhood - but allows several moves to be made in a single iteration.

Dynasearch and traditional descent can be used for all problem instances where a sequence has to be ordered, but we study the application of the algorithms to the single-machine total weighted tardiness scheduling problem (SMTWTSP). First we show that dynasearch is better than traditional descent on both computation time and solution quality. Subsequently, we compare the iterated dynasearch algorithm of Congram et al. (2002) with other algorithms based on dynasearch.

The organization of this thesis is as follows. The single-machine total weighted tardiness scheduling problem is briefly discussed in Section 2. In Section 3 we take a look at some literature about dynasearch and the tardiness scheduling problem. Section 4 presents the dynasearch concept for the total weighted tardiness problem, along with some speedups for the basic dynasearch algorithm. In Section 5, additional dynasearch neighborhoods are described. In Section 6 we present the iterated local search algorithms for traditional descent and dynasearch, and describe the breakout dynasearch algorithm of Ding et al. (2016). Section 7 reports on our computational experience; the results of the local search algorithm comparisons are given and discussed, along with the comparison of the performances of the different dynasearch-based algorithms. Finally, a conclusion on our research is given in Section 8.

2 The Tardiness Scheduling Problem

In this section, we discuss the single-machine total weighted tardiness scheduling problem, an NP-hard problem for which dynasearch initially was developed. The SMTWTSP can be stated as follows. A single machine is available to process each of n jobs, one job at a time. The processing of each job $\sigma(j)$ $(j = 1, ..., n)$ cannot be interrupted, and takes $p_{\sigma(j)}$ time periods. The relative importance of each job $\sigma(j)$ is expressed as the weight $w_{\sigma(j)}$. There are no restrictions on when the processing of job $\sigma(j)$ can start, but each job should ideally be completed before its due date $d_{\sigma(j)}$. When all jobs are scheduled, the weighted tardiness values can be computed as $w_{\sigma(j)}(P_{\sigma(j)} - d_{\sigma(j)})^+$, where $P_{\sigma(j)}$ is the total processing time of the first j scheduled jobs, i.e., $P_{\sigma(j)} = \sum_{k=1}^{j} p_{\sigma(k)}$, and $(x)^+ = \max\{x, 0\}$ for any real x. The problem is to minimize the total weighted tardiness TWT by finding an optimal processing order of all n jobs, where $TWT = \sum_{j=1}^{n} w_{\sigma(j)}(P_{\sigma(j)} - d_{\sigma(j)})^+$.

Table 1: Data for a 6-job Problem Instance

Job j	1	2	3	4	5	6
Processing time p_j	3	1	1	5	1	5
Weight w_j	3	5	1	1	4	4
Due date d_j	1	5	3	1	3	1

6-job instance example of Congram et al. (2002).

Example. Consider the 6-job instance that is specified in Table 1 (example of Congram et al. (2002)), and suppose the initial sequence is $S_0 = (1, 2, 3, 4, 5, 6)$. Figure 1 shows for every job of S_0 when it is being processed. The total weighted tardiness can be computed as follows. The weight of the first job is 3, and it is completed two time units too late, so the first job contributes $2 \times 3 = 6$ to the total weighted tardiness. The second job is completed before the due date, so it is not tardy. The third job is completed two time units after the due date, and the weight is 1, so the weighted tardiness of the third job is $2 \times 1 = 2$. Similarly, the fourth, fifth and sixth job contribute $9 \times 1 = 9$, $8 \times 4 = 32$ and $15 \times 4 = 60$ to the total weighted tardiness, respectively. Hence, the initial sequence S_0 corresponds to a total weighted tardiness value of $6 + 0 + 2 + 9 + 32 + 60 = 109$. Naturally, a different processing order of the jobs leads to a different total weighted tardiness value. For instance, sequence $S_1 = (5, 1, 2, 3, 6, 4)$, of which the processing details are also given in Figure 1, corresponds to a total weighted tardiness value of $0 + (3 \times 3) + 0 + (3 \times 1) + (10 \times 4) + (15 \times 1) = 67$. The example shows that with changing the processing order of the jobs, a significantly lower objective value can be reached.

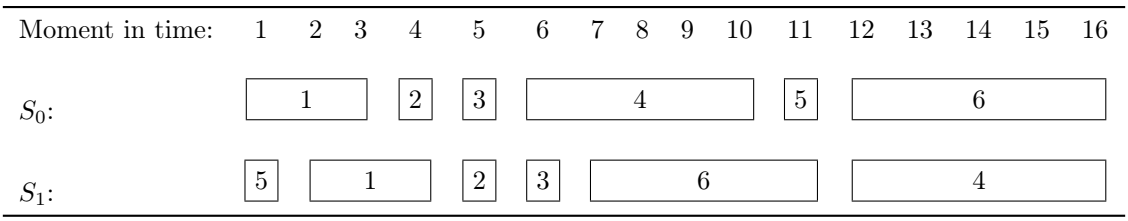


Figure 1: Processing of jobs of different sequences

3 Literature Review

The first technical report of *An iterated dynasearch algorithm for the single-machine total weighted tardiness scheduling problem* by Congram, Potts and Van de Velde dates back to december 1998. When the updated version of this paper, Congram et al. (2002), was published in the INFORMS Journal on Computing, it was still claimed that dynasearch outperformed the state-of-the-art local search algorithms at the time, namely the multi-start tabu search algorithm of Crauwels et al. (1998). However, Den Besten et al. (2000) claimed that their Ant Colony System algorithm for the SMTWTSP reached a performance similar to that of iterated dynasearch.

In the concluding remarks, Congram et al. (2002) suggest an extension for the dynasearch algorithm which they did not implement yet. Their suggestion is to extend dynasearch so that independent swap moves are used in combination with *insert moves*. This extension is implemented by Grosso et al. (2004), and used by, among others, Ergun and Orlin (2006) and Sourd (2006), all confirming that the performance of dynasearch improves when independent swap moves and insert moves are used in combination. Ergun and Orlin (2006) also introduce a *twist* neighborhood, next to the swap and insert neighborhoods. The twist neighborhood consists of all sequences that can be generated by taking a subset of the jobs in the current solution, and processing them in reverse order. In addition, the algorithm of Ergun and Orlin (2006) searches the dynasearch swap neighborhood in $O(n^2)$ time, as opposed to the $O(n^3)$ time bound for the dynasearch algorithm of Congram et al. (2002).

Through the years, several (improvements on) new algorithms to solve NP-hard problems, such as the SMTWTSP, have been introduced, either to solve them exact or with a heuristic approach (Gagné et al. (2002), Gupta and Smith (2006), Valente and Alves (2008)). Gupta and Smith (2006) compare their algorithms with the ones of Gagné et al. (2002) (who claim to have a competitive algorithm for the SMTWTSP with sequence dependent setup times), and obtain comparable computational results. However, Della Croce et al. (2011) point out that that the standard iterated dynasearch algorithm applied to the SMTWTSP still outperforms all the literature heuristics.

Furthermore, Tanaka et al. (2009) present an exact algorithm for the tardiness scheduling problem that outperforms previous exact algorithms for this problem. Anghinolfi and Paolucci (2009) present a competitive new population-based metaheuristic, called discrete particle swarm optimization. In 2010, Koulamas compares known exact approaches as well as heuristics for the SMTWTSP. The latter mentiones dynasearch briefly as the algorithm that searches an exponential sized neighborhood in polynomial time, but computational results of dynasearch are not discussed, or compared with the results of other algorithms. Ding et al. (2016) present a breakout dynasearch algorithm (BDS) that explores the search space by combining the dynasearch procedure with an adaptive perturbation strategy. They compare the computational results of BDS with the exact approach of Tanaka et al. (2009), the extended dynasearch approach (GPI-DS) of Grosso et al. (2004), and other metaheuristic algorithms. They conclude that BDS and GPI-DS outperform other known heuristics for the SMTWTSP. BDS performs slightly better than GPI-DS on large problem instances (up to 300 jobs), in terms of both solution quality and computational time.

Literature shows that the dynasearch algorithm presented by Congram et al. (2002) was an important breakthrough. Even though some extensions are made through the years, the state-of-the-art algorithms for solving the SMTWTSP are still based on the dynasearch algorithm.

4 Dynasearch

In this section we describe a basic dynasearch algorithm. In Section 4.1 we explain the dynasearch swap neighborhood and compare it with the traditional swap neighborhood. Section 4.2 describes how dynasearch combines a dynamic programming algorithm with backtracking to find local minima. Lastly, in Section 4.3 we describe some speedups for the dynasearch algorithm.

4.1 Swap Neighborhoods

Let the current processing order of the jobs be defined by a permutation σ, where $\sigma = (\sigma(1), ..., \sigma(n))$ and $\sigma(j)$ is the job on position j ($j = 1, ..., n$). Any sequence σ' is a swap neighbor of σ if with swapping two jobs on positions i and j σ' can be obtained, where $0 \leq i < j \leq n$. For example, the swap neighborhood of the permutation $(1, 2, 3)$ consists of the permutations $(2, 1, 3)$, $(1, 3, 2)$ and $(3, 2, 1)$, which are obtained by swapping 1 and 2, 2 and 3, and 1 and 3, respectively.

Any sequence σ' is a dynasearch swap neighbor of σ if σ' can be obtained by one or more swap moves, on the condition that all moves are independent. A move that swaps the jobs on positions i and j is independent with any move that swaps the jobs on positions k and l if $\max\{i, j\} < \min\{k, l\}$ or $\min\{i, j\} > \max\{k, l\}$. The strategy of dynasearch is similar to that of best-improve descent; the current solution is replaced with the best solution in the neighborhood.

The difference between swap and dynasearch swap moves is illustrated in Table 2. This table shows the moves that are made by best-improve descent and dynasearch on the initial sequence of the 6-job instance of the example in Section 2. With both approaches, the search becomes trapped in a local minimum after executing three moves. However, as the dynasearch swaps correspond to one or more traditional swaps, a lower objective value is reached.

Table 2: Swaps made by Best-Improve Descent and Dynasearch

	Best-Improve Descent		Dynasearch	
Iteration	Current Sequence	Total Weighted Tardiness	Current Sequence	Total Weighted Tardiness
	1 2 3 4 5 6	109	1 2 3 4 5 6	109
1	1 2 3 5 4 6	90	1 3 2 5 4 6	89
2	1 2 3 5 6 4	75	1 5 2 3 6 4	68
3	5 2 3 1 6 4	70	5 1 2 3 6 4	67

4.2 A Basic Dynasearch Algorithm

Dynasearch uses a dynamic programming algorithm to find the best dynasearch neighbor of the current sequence σ. First, the total weighted tardiness of the best dynasearch neighbor is determined with a recursion. Then, the corresponding sequence can be found by backtracking. The recursion works as follows. Among all partial sequences that can be obtained from the first k jobs of the current sequence $\sigma = (\sigma(1), ..., \sigma(n))$, let σ_k be the one with minimum total weighted tardiness. Now σ_0 is an empty sequence, and $\sigma_1 = (\sigma(1))$. For $k = 2, ..., n$, σ_k is obtained from a previous partial sequence σ_i, where $0 \leq i < k$. There are two possibilities for obtaining σ_k from σ_i.

The first possibility is to simply append job $\sigma(k)$ to the partial sequence σ_{k-1} (so $i = k - 1$). In this case, job $\sigma(k)$ is not involved in any swap, so $\sigma_k = (\sigma_{k-1}, \sigma(k))$. The total weighted tardiness $F(\sigma_k)$ can than be computed as the sum of the objective value of partial sequence σ_{k-1} and the weighted tardiness of job $\sigma(k)$:

$$F(\sigma_k) = F(\sigma_{k-1}) + w_{\sigma(k)}(P_{\sigma(k)} - d_{\sigma(k)})^+. \tag{1}$$

The other possibility is to append jobs $\sigma(i + 1), ..., \sigma(k)$ to σ_i, and then interchange jobs $\sigma(i + 1)$ and $\sigma(k)$, where $0 \leq i < k - 1$. Since jobs $\sigma(k)$ and $\sigma(i + 1)$ are swapped, σ_k can be written as

$\sigma_k = (\sigma_i, \sigma(k), \sigma(i+2), ..., \sigma(k-1), \sigma(i+1))$, and the total weighted tardiness $F(\sigma_k)$ can be computed as

$$\begin{aligned} F(\sigma_k) = F(\sigma_i) &+ w_{\sigma(k)}(P_{\sigma(i)} + p_{\sigma(k)} - d_{\sigma(k)})^+ \\ &+ \sum_{j=i+2}^{k-1} w_{\sigma(j)}(P_{\sigma(j)} + p_{\sigma(k)} - p_{\sigma(i+1)} - d_{\sigma(j)})^+ \\ &+ w_{\sigma(i+1)}(P_{\sigma(k)} - d_{\sigma(i+1)})^+. \end{aligned} \tag{2}$$

For the first possibility, and for all $k-1$ options in the second possibility, a candidate value for $F(\sigma_k)$ is computed. The recursion in the dynamic programming algorithm computes the smallest of the candidate values for $F(\sigma_k)$ to find the best set of independent swaps. For the backtracking procedure, let b_k represent the best candidate value for $F(\sigma_k)$. If the first possibility provides the best candidate value, b_k is set to -1. Otherwise, b_k is set to the value of i that provided the best candidate value. The initialization of the dynamic programming algorithm can now be given by

$$\begin{aligned} F(\sigma_0) &= 0, \\ F(\sigma_1) &= w_{\sigma(1)}(p_{\sigma(1)} - d_{\sigma(1)})^+, \end{aligned} \tag{3}$$

and the recursion for $k = 2, ..., n$ by

$$F(\sigma_k) = \min \begin{cases} F(\sigma_{k-1}) + w_{\sigma(k)}(P_{\sigma(k)} - d_{\sigma(k)})^+, \\ \min\limits_{0 \le i \le k-2} \begin{cases} F(\sigma_i) + w_{\sigma(k)}(P_{\sigma(i)} + p_{\sigma(k)} - d_{\sigma(k)})^+ \\ + \sum\limits_{j=i+2}^{k-1} w_{\sigma(j)}(P_{\sigma(j)} + p_{\sigma(k)} - p_{\sigma(i+1)} - d_{\sigma(j)})^+ \\ + w_{\sigma(i+1)}(P_{\sigma(k)} - d_{\sigma(i+1)})^+. \end{cases} \end{cases} \tag{4}$$

After the optimal solution value $F(\sigma_n)$ is found, the backtracking procedure can be used to find the corresponding sequence. Starting with $k = n$, if $b_k \neq -1$, job $\sigma(k)$ is swapped with job $\sigma(b_k + 1)$ and k is set to b_k. If $b_k = -1$, no swap is performed, and k is set to $k-1$. This is repeated until $k = 0$, which means that all swaps are identified and performed.

For the first iteration, the current solution usually is obtained by some heuristic. After the first iteration, the best permutation in the previous neighborhood is used as current solution. The dynasearch algorithm continues as long as improving moves can be found. If no improving moves are found in an iteration, the algorithm terminates.

4.3 Speedups

In a basic dynasearch algorithm, for every potential swap move of jobs $\sigma(i+1)$ and $\sigma(k)$, for $k = 1, ..., n$ and $i = 0, ..., k-2$, the weighted tardiness of all $k-i$ jobs in positions $i+1, ..., k$ has to be computed. We describe below speedup procedures that may guarantee that the interchange of two jobs cannot reduce the total weighted tardiness, before the weighted tardiness values of the $k-i$ jobs in positions $i+1, ..., k$ are computed. Subsequently, we describe speedup procedures that avoid the values of k corresponding to positions at the start of the sequence. For both categories, we also describe the corresponding speedups for traditional descent.

In order to efficiently implement the speedups, some preprocessing is required. Next to the partial sums of processing times $P_{\sigma(k)}$, we compute the partial sums of weighted tardiness values $V_{\sigma(k)}$, and the partial sums of weights for late jobs $W_{\sigma(k)}$:

$$\begin{aligned} V_{\sigma(k)} &= \sum_{i=1}^{k} w_{\sigma(i)}(P_{\sigma(i)} - d_{\sigma(i)})^+, \\ W_{\sigma(k)} &= \sum_{i=1}^{k} w_{\sigma(i)} U_{\sigma(i)}, \end{aligned} \tag{5}$$

for $k = 1, ..., n$, where $U_{\sigma(i)}$ is equal to one if $P_{\sigma(i)} > d_{\sigma(i)}$, and zero otherwise.

Also, a value Δ is specified. In descent, Δ is set as the best improvement found thus far (for first-improve descent this means that Δ is always zero). Moves with an improvement smaller than Δ are rejected: non-improving moves, and moves that are worse than the best move found thus far. For dynasearch, all k terms in the first minimization of the right hand side of Equation 4 provide a candidate value for $F(\sigma_k)$. We set $\Delta = F(\sigma_i) + (V_{\sigma(k)} - V_{\sigma(i)}) - \hat{F}(\sigma_k)$, where $\hat{F}(\sigma_k)$ is the best candidate value found thus far. The candidate values that exceed $\hat{F}(\sigma_k)$ are rejected.

Let $cwt_{before}(i+1,k)$ and $cwt_{after}(i+1,k)$ be the combined weighted tardiness of jobs $\sigma(i+1)$ and $\sigma(k)$, before and after the swap respectively. Furthermore, let $ub = \min\{V_{\sigma(k-1)} - V_{\sigma(i+1)}, (p_{\sigma(i+1)} - p_{\sigma(k)})(W_{\sigma(k-1)} - W_{\sigma(i+1)})\}$ be the upper bound on the reduction in the total weighted tardiness of jobs $\sigma(i+2), ..., \sigma(k-1)$. The swap operator between jobs $\sigma(i+1)$ and $\sigma(k)$ does not need to be applied if at least one of the following conditions holds.

(1) $p_{\sigma(k)} \geq p_{\sigma(i+1)}$ and $cwt_{after}(i+1,k) - cwt_{before}(i+1,k) + (\Delta)^+ > 0$,

(2) $p_{\sigma(k)} \leq p_{\sigma(i+1)}$ and $cwt_{after}(i+1,k) - cwt_{before}(i+1,k) \geq ub - \Delta$,

(3) $d_{\sigma(i+1)} \leq d_{\sigma(k)}, w_{\sigma(i+1)} \geq w_{\sigma(k)}$ and $p_{\sigma(i+1)} \leq p_{\sigma(k)}$

(4) $p_{\sigma(i+1)} < p_{\sigma(k)}, w_{\sigma(i+1)} > w_{\sigma(k)}$ and $\max\{P_{\sigma(i)} + p_{\sigma(k)}, d_{\sigma(k)}\} \geq d_{\sigma(i+1)}$,

Conditions (1) and (2) refer to speedups described by Congram et al. (2002). If condition (1) or (2) holds, we know that the required improvement will not be achieved. Condition (3) is a well-known condition for the SMTWTSP but, to our best knowledge, it was not related to dynasearch before. It is a condition based on a corollary of Rinnooy Kan (1976), who states that if this condition holds, then only those schedules in which job $\sigma(i+1)$ precedes job $\sigma(k)$ need to be considered. Thus if condition (3) holds, a sequence with $\sigma(k)$ before $\sigma(i+1)$ is never optimal. Condition (4) refers to a speedup described by Grosso et al. (2004). Consider any sequence $S = \sigma i \pi k \omega$. Rinnooy Kan et al. (1975) proved that condition (4) implies $F(\sigma k \pi i \omega) \geq F(\sigma i \pi k \omega)$, thus the corresponding swap is nonoptimal.

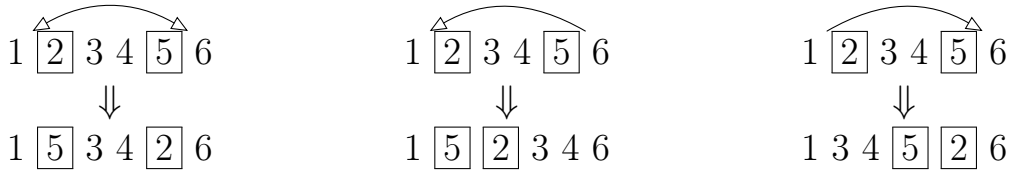
Finally, speedups can be used to avoid the values of k corresponding to positions at the start of the sequence. For dynasearch, if a dynasearch swap move does not involve the first h jobs of the sequence, the computation of $F(\sigma_k)$ for $k = 1, ..., h$ will be identical in the next iteration. Therefore, we need to perform the recursion in the next iteration only for $k = h+1, ..., n$. In the traditional descent algorithms, for every potential interchange of jobs $\sigma(i+1)$ and $\sigma(k)$, the weighted tardiness values of all jobs as a result of the swap had to be computed. However, since the jobs in the first i positions of the sequence stay the same after such an interchange, the value of $V_{\sigma(i)}$ can be used. The weighted tardiness values of the jobs in positions $i+1, ..., n$ have to be added to $V_{\sigma(i)}$ to compute the total weighted tardiness. If a potential swap move is executed, the values of $V_{\sigma(k)}$ are updated for $k = i+1, ..., n$.

5 Additional Dynasearch Neighborhoods

This section discusses two types of additional dynasearch neighborhoods. In Section 5.1 we describe the first, which is obtained by generalized pairwise interchange (GPI) operators. In Section 5.2, we present a new extension to the basic dynasearch neighborhood, obtained by allowing for dependent moves. In both sections, we discuss the changes and additions of the extension with respect to the original dynasearch (ODS) algorithm of Congram et al. (2002).

5.1 GPI-DS

In their concluding remarks, Congram et al. (2002) already suggest an extension for the dynasearch neighborhood which they did not implement yet: to use insert moves in combination with swap moves. Grosso et al. (2004) introduced this enhanced dynasearch neighborhood obtained by the GPI operators. Next to the existing swap operator, the operators EBSR (extraction and backward-shifted reinsertion) and EFSR (extraction and forward-shifted reinsertion) are defined. The different GPI operators are depicted in Figure 2.



(a) Swap (b) EBSR (c) EFSR

Figure 2: Moves of GPI operators on a sequence of six jobs

The GPI-DS recursion can now be stated, for the GPI operators $\theta = \text{swap}$, $\theta = \text{EBSR}$ and $\theta = \text{EFSR}$, as

$$\begin{aligned} &F(\sigma_0) = 0, \\ &F(\sigma_1) = w_{\sigma(1)}(p_{\sigma(1)} - d_{\sigma(1)})^+, \\ &F(\sigma_k) = \min \begin{cases} F(\sigma_{k-1}) + w_{\sigma(k)}(P_{\sigma(k)} - d_{\sigma(k)})^+, \\ \min\limits_{0 \le i \le k-2;\theta} \left\{ F(\sigma_i) + I^\theta(i+1,k), \right. \end{cases} \qquad k = 2, ..., n \end{aligned} \tag{6}$$

where $I^\theta(i+1,k)$ is the total weighted tardiness of the partial sequence $(\sigma(i+1), ..., \sigma(k))$ under application of the considered GPI operator θ. If only the swap operator is used, the recursion of Section 4.2 is obtained (Equation 3 and 4). $I^{EBSR}(i+1,k)$ and $I^{EFSR}(i+1,k)$ can be stated as

$$\begin{aligned} I^{EBSR}(i+1,k) &= w_{\sigma(k)}(P_{\sigma(i)} + p_{\sigma(k)} - d_{\sigma(k)})^+ + \sum_{j=i+1}^{k-1} w_{\sigma(j)}(P_{\sigma(j)} + p_{\sigma(k)} - d_{\sigma(j)})^+, \\ I^{EFSR}(i+1,k) &= w_{\sigma(i+1)}(P_{\sigma(k)} - d_{\sigma(i+1)})^+ + \sum_{j=i+2}^{k} w_{\sigma(j)}(P_{\sigma(j)} - p_{\sigma(i+1)} - d_{\sigma(j)})^+. \end{aligned} \tag{7}$$

Because of the larger neighborhood considered with GPI-DS, searching this neighborhood takes more computation time. In order to speed up the search, the following speedup procedures can be used, next to the basic dynasearch speedups described in Section 4.3. All the following GPI-DS speedups were suggested by Grosso et al. (2004).

The swap operator between job $\sigma(i)$ and $\sigma(k)$ does not need to be applied if at least one of the following conditions holds.

(1) $w_{\sigma(i)} \ge w_{\sigma(k)}, d_{\sigma(i)} \le d_{\sigma(k)}$ and $d_{\sigma(k)} + p_{\sigma(k)} \ge P_{\sigma(k)}$,

(2) $P_{\sigma(k)} \le d_{\sigma(k)}$.

Consider any sequence $S = \sigma i \pi k \omega$. Rinnooy Kan et al. (1975) proved that condition (1) and (2) separately imply $F(\sigma k \pi i \omega) \geq F(\sigma \pi i k \omega)$. Thus, sequence $\sigma \pi i k \omega$, obtained by applying EFSR between i and $k-1$, is not worse than $\sigma k \pi i \omega$. Hence, the considered swap is dominated.

Additionally, the EBSR operator between $\sigma(i)$ and $\sigma(k)$ does not need to be applied if at least one of the following conditions holds.

(1) $F(\sigma k i \pi \omega) < F(\sigma i k \pi \omega)$,

(2) $P_{\sigma(k)} \leq d_{\sigma(k)}$

Condition (1) implies that the EBSR between the jobs in positions $i+1$ and k is better than the one between the jobs in positions i and k. If condition (2) holds, $\sigma(k)$ is already early, so an EBSR will be non-improving.

5.2 Dependent Dynasearch

In this section, we present a new extension to the ODS algorithm of Congram et al. (2002): dependent dynasearch. First, we clarify our extension with an example. Then, we describe the specific changes of the dynamic programming algorithm with respect to the ODS algorithm.

The basic dynasearch algorithm allows only for independent swap moves to be executed. This means that, when building the optimal sequence σ_k, the two options are to just append job $\sigma(k)$ to σ_{k-1}, or append $\sigma(k)$ and execute a swap with a job on some position $i+1$, $i = 0, ..., k-2$. When such a swap is executed, the jobs $\sigma(i+2), ..., \sigma(k-1)$ stay in the same position (although their weighted tardiness values may change). Dependent dynasearch allows for dependent swap moves, by executing a single dynasearch swap move - a best series of swaps - on the jobs $\sigma(i+2), ..., \sigma(k-1)$ after $\sigma(i+1)$ and $\sigma(k)$ are swapped. The difference between an ODS swap and a DDS swap is depicted in Figure 3.

(a) Dynasearch (b) Dependent Dynasearch

Figure 3: Dynasearch swap vs. dependent dynasearch swap

Where best-improve executes the best single swap move, ODS executes the best series of independent swaps. So given an initial sequence σ, a single ODS move on σ always is as least as good as a single best-improve descent move on σ. In the same way, because DDS allows for 'swaps within swaps', a single DDS move on σ always is as least as good as a single ODS move on σ.

Let $\varsigma(a, b, t)$ be a function that returns the sequence obtained from a single dynasearch swap move on jobs $\sigma(a), ..., \sigma(b)$, given that the first job of this sequence starts at time t. The ODS recursion can now be stated as

$$
\begin{aligned}
&F(\sigma_0) = 0, \\
&F(\sigma_1) = w_{\sigma(1)}(p_{\sigma(1)} - d_{\sigma(1)})^+, \\
&F(\sigma_k) = \min \begin{cases} F(\sigma_{k-1}) + w_{\sigma(k)}(P_{\sigma(k)} - d_{\sigma(k)})^+, \\ \min\limits_{0 \leq i \leq k-2} \begin{cases} F(\sigma_i) + w_{\sigma(k)}(P_{\sigma(i)} + p_{\sigma(k)} - d_{\sigma(k)})^+ \\ +F(\varsigma(i+2, k-1, P_{\sigma(i)} + p_{\sigma(k)})) \\ +w_{\sigma(i+1)}(P_{\sigma(k)} - d_{\sigma(i+1)})^+, \end{cases} \end{cases} \qquad k = 2, ..., n,
\end{aligned} \tag{8}
$$

So where ODS just computes the weighted tardiness of jobs $\sigma(i+2), ..., \sigma(k-1)$, DDS computes what the weighted tardiness would be after a single dynasearch swap move on the corresponding jobs.

6 Iterated Algorithms

In this section, we describe two algorithms that use starting solutions obtained from previous local optima. In Section 6.1 we describe the version of iterated local search (ILS) presented by Congram et al. (2002), which we used in our computational experiments. Section 6.2 describes the breakout dynasearch (BDS) algorithm of Ding et al. (2016).

6.1 Iterated Local Search

In the ILS algorithm, a *kick* is applied to a current local optimum to generate the starting solution for the next iteration. A kick is some type of random move and the current local optimum either is the most recently found solution, or the best solution found thus far. The algorithm terminates if a predefined limit on computation time or number of iterations is reached, or if the optimal (or best known) solution is found. An overview of the ILS algorithm is given in Figure 4. In this figure, S_B corresponds to the best solution found thus far, and S_C corresponds to the current solution.

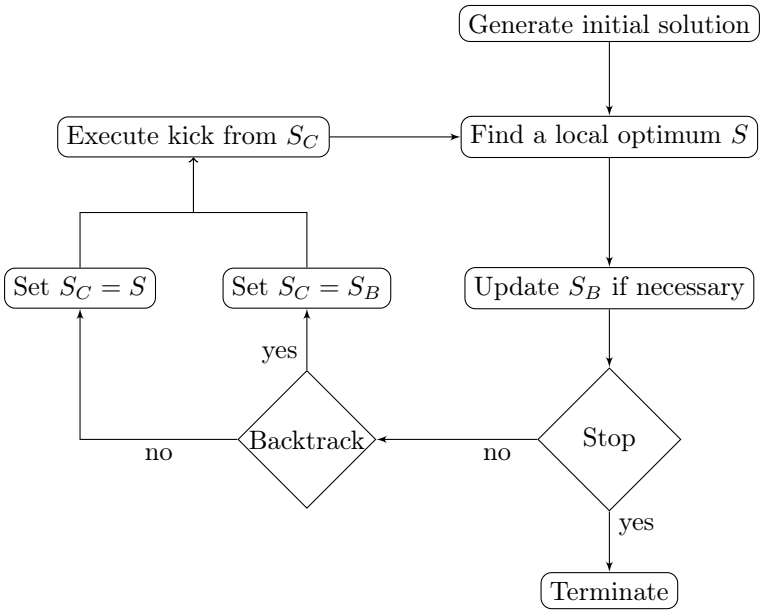


Figure 4: Overview of Iterated Local Search

In our implementation of ILS, the kick simply is a series of α random swap moves on the current solution, where α is a parameter. The value of α cannot be too big, because ILS then would resemble the multi-start approach. On the other hand, if the value of α is too small, it is unlikely for the algorithm to escape from the current local optimum. Setting the current local optimum to the best solution found thus far is called backtracking - not to be confused with the backtracking procedure in the dynasearch algorithm. An iteration refers to the procedure from finding a local optimum to executing the kick. Every β iterations, we backtrack to the best solution S_B. In the other iterations, the current solution is set to the most recently found solution. In our computational experiments, we set $\alpha = 9$ and $\beta = 5$.

For generating the initial solution, we use the Apparent Urgency (AU) heuristic. The AU heuristic starts with an empty sequence, and repeatedly fills the first unfilled position with the unscheduled job $\sigma(j)$ with the smallest AU value among all unscheduled jobs, until all jobs are scheduled. The AU value for job $\sigma(j)$ is given by

$$AU_{\sigma(j)} = \frac{w_{\sigma(j)}}{p_{\sigma(j)}} \exp\left(- \frac{(d_{\sigma(j)} - t - p_{\sigma(j)})^+}{k\bar{p}} \right), \tag{9}$$

where t is the total processing time of all scheduled jobs, $\bar{p}$ is the average processing time of all jobs, and k is a parameter which value is based on the tardiness factor TF. We use $k = 2.0$ for $TF > 0.4$, $k = 0.9$ for $TF = 0.4$ and $k = 0.5$ for $TF = 0.2$.

6.2 Breakout Dynasearch

Like with ILS, the next starting solution in BDS is obtained from a current local optimum. The difference between the two algorithms lies in the perturbation phase, where BDS uses an adaptive and multi-type perturbation mechanism as opposed to the simple kick in ILS. Also, the method of generating an initial solution differs between the algorithms.

To generate an initial solution we start with an empty sequence, and repeatedly insert the unscheduled job with the smallest value of $d_{\sigma(i)}/w_{\sigma(i)}$ among all unscheduled jobs, until all jobs are scheduled. With probability λ, the selected job is inserted in the position that gives the least increased objective value. Otherwise, it is inserted into a randomly selected position. Where Ding et al. (2016) employ the fast neighborhood search algorithm of Ergun and Orlin (2006), we use the dynasearch algorithm described in Section 4 instead (both yield the exact same solutions, but the first only requires $O(n^2)$ time to search the swap neighborhood, instead of the $O(n^3)$ time bound for the latter).

After an initial solution S^* is generated and the initial jump magnitude is set, BDS repeats the following procedure until a stopping criterion is reached (in our computational experiments, BDS employs the same stopping criteria as ILS). First, the dynasearch algorithm searches for a new local optimum, with S^* as starting solution. On the basis of this new local optimum and other history information, the jump magnitude L and perturbation type T for the next perturbation are determined. The jump magnitude is the number of perturbation moves, and the perturbation type is one of the following three options: directed swap perturbation, random forward insert perturbation and random swap perturbation. In the directed swap perturbation, all moves that reconstruct the arcs that were broken during the last ϕ iterations are forbidden. Subsequently, the best solution found thus far S^* is updated, if the new solution is better. Lastly, the perturbation is executed on S^*.

The number of perturbation moves L always is one lower, one higher or exactly the same as in the previous perturbation, and is determined as follows. If a cycle is encountered, L is increased if the upper bound L_{max} is not exceeded. If no cycle has been encountered for a specific number of iterations, and L is bigger than the lower bound L_{min}, L is decreased. Otherwise, L does not change.

The perturbation type is determined as follows. The random forward insert perturbation is selected if the best found solution has not been improved for more than T_0 iterations, where T_0 is a parameter. Because the dynasearch procedure employs only the swap operator, a forward insert perturbation is much stronger than a perturbation with swap moves. Otherwise, either the directed swap perturbation or the random swap perturbation is employed, based on the following statistic:

$$p = \begin{cases} 1 - \exp(-\frac{nc}{max\_inc}) & \text{if no cycle was encountered,} \\ \gamma + \exp(-\frac{iter\_cb}{num\_opt}) & \text{otherwise,} \end{cases} \tag{10}$$

where nc is the current number of consecutive iterations where no cycle was encountered, max_inc is the maximum number of consecutive iterations where no cycle was encountered, $iter\_cb$ is the last iteration number of when the current solution was visited before, num_opt is the total number of unique local optima that were visited, and γ is a coefficient between 0 and 1. The directed swap perturbation is now selected with probability p (and thus the random swap perturbation is selected with probability $1 - p$). For pseudo-code of the algorithm, and for specific values of the parameters used in BDS, we refer to the appendix.

7 Results

In this section, we present our computational results of the comparison of the various local search algorithms. In Section 7.1 we discuss the design of the computational experiments. In Section 7.2 we compare the performance of both multi-start and iterated versions of traditional descent and dynasearch. Section 7.3 compares the different dynasearch algorithms discussed in this thesis.

7.1 Experimental Design

For $n = 40$, $n = 50$ and $n = 100$ jobs, we use the 125 benchmark instances that are available online at the OR-Library (http://people.brunel.ac.uk/~mastjjb/jeb/orlib/wtinfo.html), along with the specifics of how the instances were generated. We compare our solution values with the best known solution values available at the OR-Library. All of our algorithms were coded in Java and run on a Toshiba laptop with an Intel Core i3 CPU. Following past researchers, we compare the performance of the various local search algorithms on basis of the following statistics:

NI The average number of iterations per instance, where an iteration refers to descending to a local minimum.

PD The percentage deviation of the solution value f_s found by the local search algorithm from the optimal (or best known) solution value OPT, i.e., PD = $100(f_s$ - $OPT)/OPT$. (When $OPT = 0$, PD = f_s is alternatively used);

APD The average PD value for a sample of 125 instances;

MPD The maximum PD value out of a sample of 125 instances;

NO The number of optimal (or best known) solution values found out of 125;

ACT:i3 The average computation time per instance in seconds on our Toshiba with Intel Core i3.

Because of the way we computed the PD values, they sometimes turn out to be very high. For example, if $OPT = 8$ and $f_s = 58$, the difference between OPT and f_s is 50, but the corresponding PD value is 625. However, if $OPT = 16,008$ and $f_s = 16,058$, the difference between OPT and f_s is also 50, but the corresponding PD value is only 0.31. The APD and MPD values are good performance measures to compare the different algorithms, but they do not give very useful information on single algorithms if the OPT and f_s values of each separate instance are not considered.

7.2 Multi-Start and Iterated Local Search

In this section we compare the performance of both multi-start and iterated versions of traditional descent and dynasearch. All multi-start algorithms are run for 0.2, 0.4 and 2.0 seconds per instance, for $n =$ 40, 50 and 100, respectively. Because the next starting solution in the iterated algorithms is relatively close to a previous local optimum, less moves are required to reach a next local optimum. Therefore, the iterated algorithms are run for 0.1, 0.2 and 1.0 seconds per instance, for $n = 40$, 50 and 100, respectively. In Table 3, the computational results are given for multi-start and iterated first-improve descent, best-improve descent and dynasearch. All results in this table are averages of the results of ten independent runs.

Table 3: Computational Results for Multi-Start and Iterated Local Search Algorithms

		First-Improve Descent				Best-Improve Descent				Dynasearch			
	n	NI	APD	MPD	NO	NI	APD	MPD	NO	NI	APD	MPD	NO
Multi-Start	40	68.4	0.601	15.891	64.0	201.2	0.279	11.120	109.4	868.8	0.061	5.284	121.2
	50	53.4	0.637	11.744	58.5	200.6	0.546	20.718	89.2	908.9	0.157	6.908	110.2
	100	14.1	1.340	26.660	30.7	120.1	2.130	100.76	43.2	649.9	1.707	97.321	63.6
Iterated	40	133.5	0.357	14.387	80.1	375.0	0.163	7.519	116.3	1289.7	0.018	2.011	124.6
	50	132.1	0.478	20.002	68.5	447.0	0.386	15.600	90.1	1613.2	0.013	1.454	123.0
	100	80.4	0.903	33.907	32.5	415.3	4.744	167.69	39.8	1683.6	0.069	5.275	119.0

From the results of Table 3 it is clearly visible that dynasearch outperforms the traditional descent algorithms on all performance measures. Despite the identical running times for all algorithms, dynasearch reaches much higher NI values, indicating the speed of the algorithm. The APD and MPD values are much lower for dynasearch, so for the instances where the optimal (or best known) solution was not found, dynasearch finds values much closer to this optimal (or best known) value than traditional descent. Finally, the NO values of dynasearch significantly improve over those of traditional descent.

The dynasearch swap neighborhood is much larger than the traditional swap neighborhood, increasing the chance of finding new local optima. However, this neighborhood is searched even faster than the traditional swap neighborhood in traditional descent. Lastly, it is shown that with the use of new starting solutions close to previous local optima, the performance of iterated local search improves over the multi-start local search performance, in almost every case.

7.3 Different Dynasearch Algorithms

In this section we compare the performance of the different dynasearch-based algorithms that are discussed in this thesis; breakout dynasearch (BDS), and iterated versions of dependent dynasearch (DDS), dynasearch (ODS) and GPI-based dynasearch (GPI-DS). First, for an initial evaluation of the contribution of DDS, we compare the DDS neighborhood search with the ODS neighborhood search. Then, we compare all four algorithms on solution quality and computation time.

Because a DDS swap move always is as least as good as an ODS swap move (if performed on the same sequence), DDS is expected to find local optima in less iterations than ODS. Figure 5 depicts a single iteration of ODS and DDS, performed on the same sequence of 40 jobs.

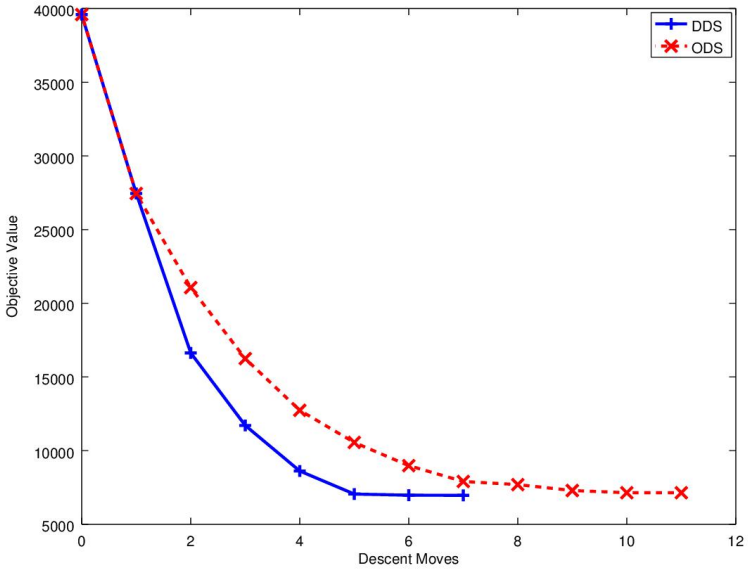


Figure 5: One iteration of ODS and DDS

Figure 5 shows that DDS finds a local optimum in only seven descent moves, while ODS needs eleven descent moves to find a local optimum. This difference is reached because DDS decreases the objective value faster than ODS, especially with the first few descent moves. Also, DDS finds a slightly better objective value than ODS. Figure 5 only shows the comparison of DDS and ODS on a single sequence of 40 jobs. However, a similar pattern is observed for almost all instances of 40, 50 and 100 jobs. On average, DDS requires less descent moves per iteration to find a local optimum.

A further comparison of DDS and ODS is given in Table 4, where also computational results of BDS

and GPI-DS are given. The results in this table are averages of the results of ten independent runs.

Table 4: Computational Results for Different Dynasearch Algorithms

n	Algorithm	NI	APD	MPD	NO	ACT:i3
40	BDS	160.44	0.0236	2.9381	124.0	0.0452
	DDS	12.87	0.0	0.0	125.0	0.0206
	ODS	28.33	0.0	0.0	125.0	0.0036
	GPI-DS	4.25	0.0	0.0	125.0	0.0028
50	BDS	764.24	0.0407	2.4126	118.5	0.3584
	DDS	49.43	0.0	0.0	125.0	0.1033
	ODS	135.79	0.0007	0.0808	124.5	0.0267
	GPI-DS	8.15	0.0	0.0	125.0	0.0082
100	BDS	3618.44	0.5834	25.8909	86.1	13.4024
	DDS	123.11	0.0	0.0	125.0	4.6670
	ODS	255.36	0.0053	0.6643	124.4	0.2301
	GPI-DS	14.86	0.0	0.0	125.0	0.1197

All algorithms are run until the optimal (or best known) solution value is found, or the maximum number of 10,000 iterations is reached. We call a job instance 'difficult' to solve for an algorithm if on average more than 10,000 iterations are needed to find the optimal (or best known) solution. The computational results show that BDS is outperformed by the other algorithms on both solution quality and computation time. For the 40-job instances, there seems to be only one 'difficult' instance for BDS (instance 85). Without the upper bound of 10,000 iterations, the NI value increases from 160.44 to 379.56, and the ACT:i3 value increases from 0.0452 to 0.1033 to solve all 40-job instances. For the 50-job and 100-job instances the number of 'difficult' instances is much higher, so not all 125 instances can be solved within a reasonable amount of time.

For ODS, one 50-job instance and one 100-job instance are 'difficult' (instances 109 and 81, respectively). However, all 125 instances can be solved with an average computation time of 0.0858 seconds for the 50-job instances, and 0.3758 seconds for the 100-job instances. Thus, ODS still outperforms DDS on computation time if no maximum number of iterations is set. The average number of iterations over 125 instances increases to 445.14 and 447.38 for the 50-job and 100-job instances, respectively.

The neighborhood searched with GPI-DS is much larger than the one searched with ODS, so executing a single move takes more time with GPI-DS. Also, one iteration (descending to a local minimum) takes more time with GPI-DS. However, as GPI-DS requires significantly less iterations, the average time to find all optimal (or best known) solutions is lower.

8 Conclusion

In our research, we first implemented the iterated and multi-start versions of the dynasearch algorithm as described by Congram et al. (2002), along with the traditional descent algorithms. Iterated (multi-start) dynasearch is shown to outperform the iterated (multi-start) traditional descent algorithms. Next, we implemented BDS, DDS and GPI-DS to run all algorithms on the same CPU, for a fair comparison of the results.

In the BDS algorithm, we employ the dynasearch algorithm of Congram et al. (2002), instead of the fast neighborhood dynasearch algorithm of Ergun and Orlin (2006). But where our results show that BDS performs significantly worse than GPI-DS, Ding et al. (2016) obtained results with BDS slightly better than those of GPI-DS. Because the description of Ding et al. (2016) is not very clear on every part of the algorithm, and their results differ so much from ours, it is very plausible that our implementation differs on more points than just the dynasearch phase from the original one of Ding et al. (2016). For example, from the description of the $DetermineJumpMagnitude$ (given in the appendix), it looks like updating the variables wc and num_nc in the **if** statement on lines 10-14 is useless, because they always get assigned to the same value at the start of $DetermineJumpMagnitude$. After the values are updated in the if statement, they are not used.

In this thesis we introduced dependent dynasearch, an extension to the original dynasearch algorithm, which allows for dependent moves. Although DDS requires significantly less iterations than ODS to find optimal (or best known) solutions, more computation time is required because a single iteration takes much more time. A topic of further research could be to speed up the DDS neighborhood search. If this can be done efficiently, DDS could eventually outperform ODS. Also, the principle of dependent moves could be investigated for the GPI-DS algorithm, which is shown to outperform BDS, DDS and ODS.

References

[1] D. Anghinolfi, M. Paolucci. A new discrete particle swarm optimization approach for the single-machine total weighted tardiness scheduling problem with sequence dependent setup times. European Journal of Operational Research 193 (2009) 73-85.

[2] J. E. Beasley. OR library: distributing test problems by electronic mail. Journal of the Operational Research Society 41 (1990) 1069–1072.

[3] Ü. Bilge, M. Kurtulan, F. Kıraç. A tabu search algorithm for the single machine total weighted tardiness problem. European Journal of Operational Research 176(3) (2007) 1423–1435.

[4] M. den Besten, T. Stützle, M. Dorigo. Ant Colony Optimization for the Total Weighted Tardiness Problem. In: Schoenauer M. et al. (eds) Parallel Problem Solving from Nature PPSN VI. PPSN 2000. Lecture Notes in Computer Science, vol 1917. Springer, Berlin, Heidelberg

[5] 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 Journal on Computing 14 (2002) 52–67.

[6] H. A. J. Crauwels, C. N. Potts, L. N. Van Wassenhove. Local search heuristics for the single machine total weighted tardiness scheduling problem. INFORMS Journal on Computing 10 (1998) 341–350.

[7] F. Della Croce, E. Desmier, T. Garaix. A note on "Beam search heuristics for the single machine early/tardy scheduling problem with no machine idle time". Computers & Industrial Engineering 60 (2011) 183-186.

[8] J. Ding, Z. Lü, T. C. E. Cheng, L. Xu. Breakout dynasearch for the single-machine total weighted tardiness problem. Computers & Industrial Engineering 98 (2016) 1-10.

[9] Ö. Ergun, J. B. Orlin. Fast neighborhood search for the single machine total weighted tardiness problem. Operations Research Letters 34 (2006) 41-45.

[10] C. Gagné, W. Price, M. Gravel, Comparing an aco algorithm with other heuristics for the single machine scheduling problem with sequence-dependent setup times, Journal of the Operational Research Society 53 (2002) 895–906.

[11] S. R. Gupta, J. S. Smith. Algorithms for single machine total tardiness scheduling with sequence dependent setups. European Journal of Operational Research 175 (2006) 722-739.

[12] A. Grosso, F. Della Croce, R. Tadei. An enhanced dynasearch neighborhood for the single-machine total weighted tardiness scheduling problem. Operations Research Letters 32 (2004) 68–72.

[13] C. Koulamas. The single-machine total tardiness scheduling problem: Review and extensions. European Journal of Operational Research 202 (2010) 1-7.

[14] A. H. G. Rinnooy Kan. Machine Scheduling Problems Classification, Complexity and Computations. Martinus Nijhoff, The Hague (1976).

[15] A.H.G. Rinnooy Kan, B.J. Lageweg, J.K. Lenstra. Minimizing total costs in one-machine scheduling. Oper. Res. 23 (1975) 908–927.

[16] F. Sourd. Dynasearch for the earliness-tardiness scheduling problem with release dates and setup constraints. Operations Research Letters 34 (2006) 591-598.

[17] S. Tanaka, S. Fujikuma, M. Araki. An exact algorithm for single-machine scheduling without machine idle time. J Sched (2009) 12: 575-593. doi:10.1007/s10951-008-0093-5.

[18] J. M. S. Valente, R. A. F. S. Alves. Beam search algorithms for the single machine total weighted tardiness scheduling problem with sequence-dependent setups. Computers & Operations Research 35 (2008) 2388-2405.

Appendix

BDS

This section presents the pseudo-code and parameter values of BDS. The pseudo-code of BDS is given in Algorithm 1.

Algorithm 1 Pseudo-code of BDS for the SMTWTSP

Input: Processing time, weight and due time of each job in an unscheduled job sequence
Output: The best scheduled sequence S^* found so far
$S^* \leftarrow GenerateInitialSolution()$
$L \leftarrow L_0$
while stopping condition not reached **do**
    $S' \leftarrow Dynasearch(S^*)$
    $L \leftarrow DetermineJumpMagnitude(L, S', history)$
    $T \leftarrow DetermineTypePerturbation(S', history)$
    if $F(S') < F(S^*)$ **then**
        $S^* = S'$
    end if
    $S^* \leftarrow Perturbation(L, T, S^*, history)$
end while

For determining the jump magnitude and perturbation type, the following statistics are used.

HT	A hash table with all previous encounterd local optima, along with the iteration number of when they were visited.
lc	The number of the iteration of when the last cycle was encountered.
$iter_{cur}$	The current iteration number.
w	The number of consecutive iterations at which no improvement on the best solution was found.
$prev\_visit$	The iteration number of when the current local optimum was previously visited ($prev\_visit = -1$ if the current local optimum was not visited before).

The pseudo-code of $DetermineJumpMagnitude$ is given in Algorithm 2. The algorithm shows that L is decreased if the no cycle has been encountered for more than $(wc/num\_nc)\mu$ iterations, where (wc/num_nc) is supposed to be the average number of iterations between two consecutive cycles, and μ is a coefficient. The parameter settings for BDS are given in Table 5. The values are based on initial experiments of Ding et al. (2016).

Algorithm 2 Pseudo-code of *DetermineJumpMagnitude*

Input: Local optimum S returned by dynasearch, current jump magnitude L and history information including HT, lc, $iter_{cur}$ and w
Output: Jump magnitude L for the next perturbation phase
$wc \leftarrow 10, num\_nc \leftarrow 1$
$prev\_visit \leftarrow PreviousEncounter(HT, F(S))$
if $F(S) < F(S^*) || w > T_0$ **then**
    $w \leftarrow 0$
else
    $w \leftarrow w + 1$
end if
if $prev\_visit \neq -1$ **then**
    $wc \leftarrow wc + iter_{cur} - lc$
    $num\_nc \leftarrow num\_nc + 1$
    $lc \leftarrow iter_{cur}$
    $L \leftarrow L + 1$
else if $(iter_{cur} - lc) > (wc/num\_nc) \cdot \mu$ **then**
    $L \leftarrow L - 1$
end if
if $L > L_{max}$ **then**
    $L \leftarrow L_{max}$
else if $L < L_{min}$ **then**
    $L \leftarrow L_{min}$
end if

Table 5: Parameter settings for BDS for different problem sizes

Parameter	Description	Value
λ	Parameter used for the initial solution	0.8
μ	Coefficient for determining jump magnitude	2
γ	Coefficient for determining perturbation type	0.3
ϕ	Tabu tenure in directed perturbation	$n/4$
T_0	Coefficient for strong perturbation	$500 + n$
L_{min}	Minimum number of jump magnitude	4
L_{max}	Maximum number of jump magnitude	$10 + n/100$