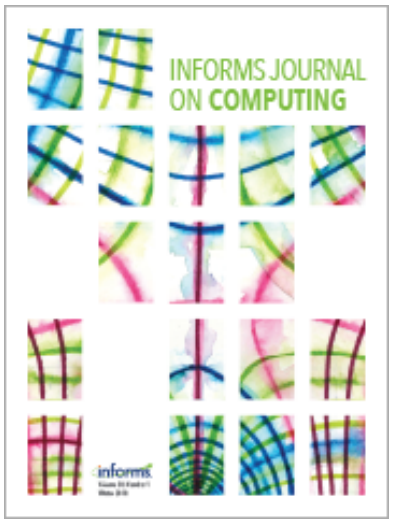

INFORMS Journal on Computing

Publication details, including instructions for authors and subscription information:
http://pubsonline.informs.org

An Iterated Dynasearch Algorithm for the Single-Machine Total Weighted Tardiness Scheduling Problem

Richard K. Congram, Chris N. Potts, Steef L. van de Velde,

To cite this article:
Richard K. Congram, Chris N. Potts, Steef L. van de Velde, (2002) An Iterated Dynasearch Algorithm for the Single-Machine Total Weighted Tardiness Scheduling Problem. INFORMS Journal on Computing 14(1):52-67. https://doi.org/10.1287/ijoc.14.1.52.7712

An Iterated Dynasearch Algorithm for the Single-Machine Total Weighted Tardiness Scheduling Problem

Richard K. Congram • Chris N. Potts • Steef L. van de Velde
Faculty of Mathematical Studies, University of Southampton, Southampton, SO17 1BJ, UK
Faculty of Mathematical Studies, University of Southampton, Southampton, SO17 1BJ, UK
Department of Decision and Information Sciences, Rotterdam School of Management, Erasmus University, P.O. Box 1738, 3000 DR Rotterdam, The Netherlands
Richard.Congram@paconsulting.com • C.N.Potts@maths.soton.ac.uk • S.Velde@fac.fbk.eur.nl

This paper introduces a new neighborhood search technique, called dynasearch, that uses dynamic programming to search an exponential size neighborhood in polynomial time. While traditional local search algorithms make a single move at each iteration, dynasearch allows a series of moves to be performed. The aim is for the lookahead capabilities of dynasearch to prevent the search from being attracted to poor local optima. We evaluate dynasearch by applying it to the problem of scheduling jobs on a single machine to minimize the total weighted tardiness of the jobs. Dynasearch is more effective than traditional first-improve or best-improve descent in our computational tests. Furthermore, this superiority is much greater for starting solutions close to previous local minima. Computational results also show that an iterated dynasearch algorithm in which descents are performed a few random moves away from previous local minima is superior to other known local search procedures for the total weighted tardiness scheduling problem.
(*Production Scheduling: Single Machine, Sequencing; Analysis of Algorithms; Dynamic Programming*)

1. Introduction

A descent or iterative improvement algorithm is a simple and practical type of local search method for obtaining near-optimal solutions for a wide variety of NP-hard combinatorial optimization problems. The main idea is to improve a feasible solution by performing a series of transformations or *moves*. The transformations are normally specified by a neighborhood structure, that defines which solutions can be generated by a single move. Given a feasible solution, the algorithm searches its neighborhood for a better solution. If a better solution exists, then one such solution is selected as the current solution and the search repeats; if there is no better solution, then the algorithm terminates. There are two main types of descent algorithms: *first-improve*, in which the current solution is replaced with the first better solution found in the neighborhood; and *best-improve*, in which the current solution is replaced with a best solution in the neighborhood. Subsequently, we refer to first-improve and best-improve as *traditional* descent algorithms. By definition, any type of descent algorithm terminates in a local optimum. Furthermore, the quality of the final solution as compared with the global optimum depends on the size and the structure of the neighborhood.

The main handicap of a descent algorithm is its myopic nature: it looks only one single move ahead,

INFORMS Journal on Computing © 2002 INFORMS
Vol. 14, No. 1, Winter 2002 pp. 52–67
0899-1499/02/1401/0052$5.00
1526-5528 electronic ISSN

and any move can lead to a "bad" solution where the search becomes trapped in a local optimum that may be substantially worse than a global optimum. This is the reason that traditional descent algorithms generally cannot compete with more sophisticated local search methods, such as simulated annealing, tabu search, and genetic algorithms, in terms of solution quality. On the positive side, descent algorithms are generally much faster and easier to implement.

A well-known approach for improving the solution quality of a descent algorithm is to adopt a *multi-start* approach in which several independent runs of descent, each using a different randomly generated starting solution, are performed, and then the best of the resulting solutions is selected. However, a far more effective approach is to allow dependent runs by generating the new starting solution from one of the previous local optima by a suitable perturbation method. Such an approach is known as *iterated* descent, and is a widely-recognized method of obtaining high solution quality at relatively low computational cost, without resorting to more intricate local search algorithms. To stress the power of this type of approach, we note that the iterated Lin-Kernighan algorithm, as implemented by Applegate et al. (1999), is a state-of-the-art local search algorithm for the traveling salesman problem—and this problem provides the single most competitive stage for testing new local search algorithms.

In this paper, we introduce *dynasearch* as a new approach for overcoming the myopic nature of traditional descent algorithms. In traditional descent, lexicographic search is used to find either a better or a best solution in a neighborhood that is usually of polynomial size. On the other hand, dynasearch uses dynamic programming in a local search algorithm, so that a neighborhood of *exponential size* is explored in *polynomial time*. Specifically, the generic key idea of dynasearch is to consider a neighborhood of the type used in a traditional descent algorithm, but to allow several moves of a certain type to be made in a single iteration. Dynamic programming is used to find the best combination of moves. Thus, each neighbor in a dynasearch algorithm corresponds to one or more moves in traditional descent.

Algorithms that explore exponential neighborhoods in polynomial time have been known for many years, tracing back to the work by Sarvanov and Doroshko (1981a, 1981b) for the traveling salesman problem (TSP). In the TSP, a salesman must perform a tour of shortest length, visiting each of n cities exactly once and then returning to the starting city. For an overview of research on such neighborhoods, we refer to the review of Deĭneko and Woeginger (2000) for the TSP, and to the general survey of Ahuja et al. (1999) which covers various problem areas. From these surveys, the main techniques that are identified for performing a search of an exponential number of solutions in polynomial time are matching (or linear assignment) algorithms, cyclic exchange techniques, and dynamic programming (or shortest path algorithms).

We briefly discuss the previous studies that use dynamic programming to explore a neighborhood of exponential size in polynomial time, since this relates most closely to our work on dynasearch. A pyramidal neighbor for the TSP passes through a subset of cities in the original order, and then the remaining cities in the reverse order. Sarvanov and Doroshko (1981a) and Carlier and Villon (1990) give a dynamic programming algorithm that explores in $O(n^2)$ time the pyramidal neighborhood, which has size $\Theta(2^n)$. Thus, this neighborhood differs from one comprising multiple moves as used in dynasearch. Balas (1999) and Balas and Simonetti (2001) consider a neighborhood for the TSP in which, for some given k, the sequence of cities can be rearranged subject to the constraint that if one city precedes another by more than k positions in the original sequence, then it must also precedes the other city in the rearranged sequence. A dynamic program finds the best of the $\Omega(((k-1)/e)^{n-1})$ neighbors in $O(k^2 2^k n)$ time. Again, this neighborhood is not built from multiple moves. A twisted sequence neighborhood for the TSP allows certain non-overlapping sections of the sequence representing the current tour to be reversed. Congram (2000) shows that the size of this neighborhood is $\Theta((3+2\sqrt{2})^n)$. Moreover, Deĭneko and Woeginger (2000) provide a dynamic programming algorithm that finds the best twisted sequence neighbor in

$O(n^7)$ time. The twisted sequence neighborhood contains all multiple 2-opt moves (a 2-opt move replaces two edges of the tour with two new edges) that would be produced by dynasearch. However, the high time complexity to search the twisted sequence neighborhood means that it is only of theoretical interest.

In parallel with our work on dynasearch, Hurink (1999) has developed a neighborhood that employs multiple moves for a problem of scheduling a single machine to minimize the total weighted completion time in which batching decisions have to be made. More precisely, each batch of jobs requires a setup time, and all jobs of the same batch have a common completion time, namely the time at which the batch completes its processing. The solution of this problem is represented by a sequence of jobs, since there is an efficient dynamic programming algorithm to form the batches that minimize the total weighted completion time when the sequence is known. Hurink's neighborhood uses multiple transposes of jobs in the sequence. His computational results indicate that this neighborhood is not particularly effective at exploring the solution space relative to one that removes a job from its current position and inserts it elsewhere. Forming a neighborhood from multiple insert moves appears difficult due to the complex relationship between the sequence and the batches produced by the dynamic programming algorithm.

Thus, existing studies in the literature that use a polynomial-time dynamic programming algorithm to search a neighborhood of exponential size are strictly limited. In this respect, our work offers a novel contribution.

This paper also studies the application of dynasearch to the single-machine total weighted tardiness scheduling problem. This work forms part of a research program to evaluate dynasearch as a technique for generating high-quality solutions to a variety of combinatorial optimization problems. The total weighted tardiness problem is an NP-hard archetypal machine scheduling problem for which many local search algorithms have been developed and tested; for an overview see Anderson et al. (1997). By showing that an iterated dynasearch algorithm outperforms the state-of-the-art local search algorithms, namely the multi-start tabu search algorithm of Crauwels et al. (1998), we are able to demonstrate the value of the dynasearch approach.

The organization of the paper is as follows. In Section 2, we give a formal description of the single-machine total weighted tardiness scheduling problem. Section 3 presents the dynasearch concept for the total weighted tardiness problem: it includes a discussion of the structure of the dynasearch neighborhood, the presentation of a polynomial-time dynamic programming algorithm to find the best solution in this neighborhood, the description of a simple dynasearch algorithm without re-starts, and finally some speed-ups to reduce the computation time. In Section 4, we present the iterated dynasearch algorithm. Here, we first discuss the general principles of multi-start and iterated local search, and then pay special attention to the concept of restarting the local search close to a local optimum. A detailed description of our iterated dynasearch algorithm follows. Section 5 reports on our computation experience. After discussing the design of our computational experiments, we compare multi-start and iterated versions of dynasearch with the corresponding multi-start and iterated versions of first-improve and best-improve descent. We observe that, although multi-start dynasearch is marginally superior to multi-start descent, the performance of iterated dynasearch clearly dominates that of both iterated descent algorithms. Then, we compare iterated dynasearch with the state-of-the-art multi-start tabu search algorithm of Crauwels et al. (1998), and conclude that iterated dynasearch is significantly better. Finally, some concluding remarks are contained in Section 6.

2. The Total Weighted Tardiness Problem

The single-machine total weighted tardiness scheduling problem can be stated as follows. Each of n jobs (numbered $1, \ldots, n$) is to be processed without interruption on a single machine that can handle no more than one job at a time. Job j ($j = 1, \ldots, n$) becomes available for processing at time zero, requires processing during an uninterrupted positive processing time

p_j, has a positive weight w_j, and has a due date d_j by which time it should ideally be completed. For a given processing order of the jobs, the earliest completion time C_j and the tardiness $T_j = \max\{C_j - d_j, 0\}$ of each job j can readily be computed. The problem is to find a processing order of the jobs with minimum total weighted tardiness $\sum_{j=1}^{n} w_j T_j$.

This total weighted tardiness scheduling problem is not only NP-hard in the strong sense (Lawler 1977, Lenstra et al. 1977), but is also very difficult from a practical point of view: the state-of-the-art branch and bound algorithm of Potts and Van Wassenhove (1985) runs into severe troubles when trying to solve instances with more than 50 jobs to optimality. Approximation also appears difficult, since there are no polynomial-time algorithms in the literature that are guaranteed to provide a solution with a value that is no more than a constant times the optimal solution value.

In a computational study, Crauwels et al. (1998) compare the performance of multi-start versions of simulated annealing, threshold accepting, tabu search and genetic algorithms. For each algorithm, one variant uses the "natural" representation of solutions as a permutation of the integers $1, \ldots, n$ to specify the processing order of jobs, and another variant uses a binary representation in which jobs are indicated as late or non-late, from which a decoding heuristic constructs a processing order of the jobs. Their results show that the best-quality solutions are provided by a multi-start tabu search algorithm with the permutation representation. Moreover, this is the champion local search algorithm, since it is superior to the simulated annealing algorithm of Matsuo et al. (1987), and to the descent and simulated annealing algorithms of Potts and Van Wassenhove (1991).

3. Dynasearch

Potts and Van Wassenhove (1991) and Crauwels et al. (1998) find that for the natural permutation representation, the swap neighborhood is preferred to other neighborhoods for the total weighted tardiness scheduling problem. In this section, we present the principles of our dynasearch algorithm. We compare the swap neighborhood with its dynasearch counterpart (Section 3.1), present a dynamic programming algorithm to search this neighborhood (Section 3.2), sketch the dynasearch algorithm without restarts (Section 3.3), and finally introduce some speed-ups that reduce the empirical running time of the dynasearch algorithm (Section 3.4). Note that the presentation of the iterated dynasearch algorithm is deferred to Section 4.

3.1. Swap and Dynasearch Swap Neighborhoods

For many combinatorial optimization problems whose solutions can be represented as sequences, partitions, or assignments, some type of k-exchange neighborhood structure ($k \geq 2$) is usually adopted, since it is both effective and easy to search. The k-exchange neighborhood contains all solutions that can obtained by exchanging k elements in the sequence, partition, or assignment. For a sequencing problem, *swap* is a 2-exchange neighborhood that interchanges any two elements, irrespective of whether they are adjacent. As an example, the permutation $(3, 2, 1, 4, 5, 6)$ is a swap neighbor of $(1, 2, 3, 4, 5, 6)$, that is obtained by swapping elements 1 and 3. Verifying local optimality for a k-exchange neighborhood requires $\Omega(n^k)$ time, where n is the total number of elements. For small values of k, a k-exchange neighborhood can be searched quickly but, when used in a traditional descent algorithm, the resulting solutions are only of average quality. As k increases, the computational effort required to search the neighborhood grows quickly, so that selecting larger values of k is often impractical. Therefore, a common choice is $k = 2$, which for sequencing problems corresponds to the swap neighborhood.

Let $\sigma = (\sigma(1), \ldots, \sigma(n))$ be a permutation or sequence that defines the current processing order of the jobs, where $\sigma(i)$ is the job in position i, for $i = 1, \ldots, n$. The swap neighborhood of a given permutation σ comprises all sequences that can be obtained by interchanging any two jobs $\sigma(i)$ and $\sigma(j)$, where $1 \leq i < j \leq n$. The size of the swap neighborhood is $n(n-1)/2$.

The *dynasearch swap* neighborhood of σ allows a new permutation to be obtained by a series of swaps. To specify which swaps are allowed, we need a definition. The two moves that swap job $\sigma(i)$ with

job $\sigma(j)$, and job $\sigma(k)$ with job $\sigma(l)$, respectively, are said to be *independent* if $\max\{i, j\} < \min\{k, l\}$ or $\min\{i, j\} > \max\{k, l\}$. The dynasearch swap neighborhood consists of all solutions that can be obtained from σ by a series of pairwise independent swap moves. We claim that the dynasearch swap neighborhood has size $2^{n-1} - 1$. To justify our claim, we note that any dynasearch neighbor is defined by specifying which of the jobs in the first $n-1$ positions is involved in a swap move (since there are an even number of swapped job, it can be deduced whether the job in position n is to be swapped). Since there are 2^{n-1} ways of selecting the positions of the swapped jobs, and one of these corresponds to the situation where no jobs are swapped, we obtain the claimed neighborhood size.

Dynasearch uses a best-improve strategy: a dynasearch swap move is equivalent to a *best* series of independent swaps. Consequently, if a solution is a local optimum with respect to the dynasearch swap neighborhood, then it is also a local optimum with respect to the swap neighborhood, and vice versa.

We now present an example to illustrate the difference between the swap and dynasearch swap neighborhoods.

Example. Consider the 6-job instance that is specified in Table 1. Suppose the initial sequence is $(1, 2, 3, 4, 5, 6)$.

Table 2 shows the moves that are made by a best-improve descent algorithm with the swap neighborhood. With this traditional approach, we observe that the search becomes trapped in a local minimum with a total weighted tardiness of 70.

Table 3 shows the ability of dynasearch to make multiple independent swaps to move to the next solution. The example may suggest that, by definition, dynasearch necessarily yields a better result than best-improve (or first-improve) descent, for any instance. This is not the case: there may be instances for which traditional descent is better. Our claim, which is substantiated by the computational results in Section 5.2, is that dynasearch is better on average.

Table 1 Data for the 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

Table 2 Swaps Made by Best-Improve Descent

Iteration	Current Sequence	Total Weighted Tardiness
	1 2 3 4 5 6	109
1	1 2 3 5 4 6	90
2	1 2 3 5 6 4	75
3	5 2 3 1 6 4	70

3.2. Finding the Best Set of Independent Swaps

To find the best set of independent swaps that can be obtained from a permutation σ, we employ a dynamic programming algorithm. This algorithm uses a forward enumeration scheme in which jobs are appended to the end of the current partial sequence and are possibly swapped. We define a partial sequence to be in state (k, σ), for $k = 0, 1, \ldots, n$, if it can be obtained from the partial sequence $(\sigma(1), \ldots, \sigma(k))$ by applying a series of independent swaps. Of course, to find the best possible sequence in the dynasearch swap neighborhood of σ, which by definition must be in state (n, σ), we only need to consider a sequence that has minimum objective value among all sequences in this state.

Let σ_k be a partial sequence with minimum total weighted tardiness for jobs $\sigma(1), \ldots, \sigma(k)$ among partial sequences in state (k, σ). Further, let $F(\sigma_k)$ be the total weighted tardiness for jobs $\sigma(1), \ldots, \sigma(k)$ in σ_k. This partial sequence must be obtained from a partial sequence σ_i that has minimum objective value from all partial sequences in some previous state (i, σ), where $0 \le i < k$, by appending job $\sigma(k)$ if $i = k - 1$, or by first appending jobs $\sigma(i+1), \ldots, \sigma(k)$ and

Table 3 Dynasearch Swaps

Iteration	Current Sequence	Total Weighted Tardiness
	1 2 3 4 5 6	109
1	1 3 2 5 4 6	89
2	1 5 2 3 6 4	68
3	5 1 2 3 6 4	67

then interchanging jobs $\sigma(i+1)$ and $\sigma(k)$ if $0 \le i < k-1$. These two possibilities are considered in detail below.

• $i = k-1$. In this case, job $\sigma(k)$ is not involved in any swap, and $\sigma(k)$ is simply appended to a partial sequence σ_{k-1}; hence, $\sigma_k = (\sigma_{k-1}, \sigma(k))$. Accordingly,

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

where $(x)^+ = \max\{x, 0\}$ for any real x, and $P_{\sigma(k)} = \sum_{i=1}^{k} p_{\sigma(i)}$.

• $0 \le i < k-1$. Here, jobs $\sigma(k)$ and $\sigma(i+1)$ are swapped, so that σ_k can be written as $\sigma_k = (\sigma_i, \sigma(k), \sigma(i+2), \ldots, \sigma(k-1), \sigma(i+1))$, and the total weighted tardiness $F(\sigma_k)$ is readily computed as

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

Since all possibilities are considered, a recursion that computes the smaller of the two candidate values $F(\sigma_k)$ can be used in a dynamic programming algorithm to find the best set of independent swaps.

We are now ready to present our dynamic programming algorithm. The initialization is

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

and the recursion for $k = 2, \ldots, n$ is

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

The optimal solution value is then equal to $F(\sigma_n)$, and the corresponding sequence can be found by backtracking. For example, if the value of $F(\sigma_n)$ is given by the first term in the minimization, then job $\sigma(n)$ is not swapped, and we proceed to find how the value $F(\sigma_{n-1})$ is determined. Alternatively, if the value of $F(\sigma_n)$ is given by the second term for some index i, then jobs $\sigma(n)$ and $\sigma(i+1)$ are swapped, and we proceed to find how the value $F(\sigma_i)$ is determined. We continue in this way until $F(\sigma_0)$ or $F(\sigma_1)$ appears on the right-hand side of the recursion equation, in which case all independent swaps are identified and the backtracking procedure terminates.

This dynamic programming algorithm runs in $O(n^3)$ time and requires $O(n)$ space. Hence, dynasearch requires $O(n^3)$ time to verify local optimality, which is the same time requirement as for a traditional descent algorithm.

Finally, we note that a backward dynamic programming algorithm can be derived. However, it does not offer any computational advantages, since the backward algorithm has the same time and space requirements as the forward scheme.

3.3. A Basic Dynasearch Algorithm

In a straightforward implementation of the dynasearch algorithm with the swap neighborhood, we start with an initial permutation $\sigma^{(0)}$ as the current solution, where $\sigma^{(0)}$ is obtained by some constructive heuristic. At this stage, it is irrelevant as to how this initial solution is obtained. During iteration t, $\sigma^{(t-1)}$ is the current permutation, which we attempt to improve by making a move in the dynasearch swap neighborhood. Using the dynamic program of Section 3.2, we compute the values $F(\sigma_k^{(t-1)})$ for $k = 1, \ldots, n$, and then apply a backtracking procedure to find a corresponding permutation $\sigma^{(t)}$.

The solution defined by $\sigma^{(t)}$ is a local optimum in the dynasearch swap neighborhood if $F(\sigma_n^{(t-1)}) = F(\sigma_n^{(t-2)})$, where we set $F(\sigma_n^{(-1)})$ to be the objective value of the initial permutation $\sigma^{(0)}$. In this case, the algorithm terminates. On the other hand, if $F(\sigma_n^{(t-1)}) < F(\sigma_n^{(t-2)})$, then a further iteration with $\sigma^{(t)}$ as the current permutation is executed.

3.4. Speedups

There are several tricks that can be employed to reduce the computation time for traditional descent and dynasearch algorithms. We refer to a swap move that strictly reduces the total weighted tardiness as *improving*; otherwise, it is *non-improving*. Speedups that reduce the computation time for determining whether a move is non-improving, or is dominated

by some other move, are useful for both descent and dynasearch. (However, they would not be useful for a simulated annealing algorithm.) Another category of speedups applies to the dynamic programming algorithm that is used in dynasearch. These two categories of speedups are discussed in the following subsections.

3.4.1. Speedups for Total Weighted Tardiness Comparisons. Before providing details of our speedups, we first present some preprocessing that allows these speedups to be implemented efficiently. As indicated in Section 3.2, we compute for the current sequence σ the partial sums of processing times

$$P_{\sigma(k)} = \sum_{i=1}^{k} p_{\sigma(i)},$$

for $k = 1, \ldots, n$. Additionally, we compute the partial sums of weighted tardiness values

$$V_{\sigma(k)} = \sum_{i=1}^{k} w_{\sigma(i)}(P_{\sigma(i)} - d_{\sigma(i)})^+,$$

and the partial sums of weights for late jobs

$$W_{\sigma(k)} = \sum_{i=1}^{k} w_{\sigma(i)} U_{\sigma(i)},$$

for $k = 1, \ldots, n$, where

$$U_{\sigma(i)} = \begin{cases} 1 & \text{if } P_{\sigma(i)} > d_{\sigma(i)}, \\ 0 & \text{otherwise.} \end{cases}$$

We now describe some tests which, if successful, indicate that interchanging jobs $\sigma(i+1)$ and $\sigma(k)$, where $0 \le i \le k-2$, of the current permutation σ, cannot reduce the total weighted tardiness by more than some specified value Δ. In descent, such a test is used to reject a potential move that swaps jobs $\sigma(i+1)$ and $\sigma(k)$. Specifically, for first-improve descent, we set $\Delta = 0$, so that non-improving moves are rejected. For best-improve descent, we set $\Delta = 0$ at the start of the neighborhood search for σ, and then update Δ as appropriate so that Δ is the best improvement found thus far. In this case, the test rejects non-improving moves, and moves that cannot achieve an improvement that exceeds Δ. For dynasearch, the computation of $F(\sigma_k)$ by dynamic programming uses minimization to compare candidate values that are obtained for different values of i. From the recursion, an initial candidate value is $\widehat{F}(\sigma_k) = F(\sigma_{k-1}) + w_{\sigma(k)}(P_{\sigma(k)} - d_{\sigma(k)})^+$. The candidate value for i, where $0 \le i \le k-2$, is equal to $F(\sigma_i)$ plus the total weighted tardiness for jobs $\sigma(i+1), \ldots, \sigma(k)$ that results from swapping jobs $\sigma(i+1)$ and $\sigma(k)$. We set $\Delta = F(\sigma_i) + (V_{\sigma(k)} - V_{\sigma(i)}) - \widehat{F}(\sigma_k)$, where $\widehat{F}(\sigma_k)$ is the best candidate value found thus far. Thus, the test rejects the candidate value corresponding to i when it cannot provide a better candidate value than $\widehat{F}(\sigma_k)$.

Under the most straightforward implementation, to evaluate the interchange of the jobs in positions $i+1$ and k requires the total weighted tardiness of $k-i$ jobs in positions $i+1, \ldots, k$ to be computed. We describe below a pre-testing procedure that, although avoiding the computation of $k-i$ weighted tardiness values, may indicate that the swap move cannot improve the total weighted tardiness by more than Δ. For cases in which the result of this pre-testing is inconclusive, we present a method of reducing the computational effort in evaluating the total weighted tardiness of the relevant $k-i$ jobs.

We now present our pre-testing procedure which, if successful, guarantees that interchanging jobs $\sigma(i+1)$ and $\sigma(k)$ cannot reduce the total weighted tardiness by more than Δ. First, suppose that $p_{\sigma(k)} \ge p_{\sigma(i+1)}$. If the combined weighted tardiness of jobs $\sigma(i+1)$ and $\sigma(k)$ increases, or decreases by an amount that does not exceed Δ as a result of the swap, then the required improvement is not achieved. Alternatively, suppose that $p_{\sigma(k)} \le p_{\sigma(i+1)}$. An upper bound on the reduction in the total weighted tardiness of jobs $\sigma(i+2), \ldots, \sigma(k-1)$ is given by $\min\{V_{\sigma(k-1)} - V_{\sigma(i+1)}, (p_{\sigma(i+1)} - p_{\sigma(k)})(W_{\sigma(k-1)} - W_{\sigma(i+1)})\}$. Therefore, if the combined weighted tardiness of jobs $\sigma(i+1)$ and $\sigma(k)$ increases by an amount that exceeds or is equal to this upper bound minus Δ as a result of the swap, then the required improvement is not achieved.

We now describe our method of computing the total weighted tardiness of jobs $\sigma(i+1), \ldots, \sigma(k)$. At the start of each iteration, we partition the current sequence σ into runs of non-late and late jobs, where a run of non-late (late) jobs is a maximal set of adjacent non-late (late) jobs. Our motivation is that if $p_{\sigma(k)} \le p_{\sigma(i+1)}$, then all the jobs in positions $i+2, \ldots, k-1$ are completed at the same time or earlier

after swapping jobs $\sigma(i+1)$ and $\sigma(k)$. Hence, there is no delay to any run of non-late jobs within positions $i+2, \ldots, k-1$, so that their contribution to the total weighted tardiness remains zero. Similarly, if $p_{\sigma(k)} \geq p_{\sigma(i+1)}$, then the total weighted tardiness of any run of late jobs within positions $i+2, \ldots, k-1$ increases by $(p_{\sigma(k)} - p_{\sigma(i+1)})W$ as a result of the swap, where W is the total weight of jobs in the run.

To use the above speedups for runs of non-late and late jobs, we associate a string with the current sequence of jobs, where the string is constructed as follows. If the first run of jobs is late, then the string starts with a 1; otherwise, it starts with a 0. Whenever a new run starts, the position of the first job in the run is added to the string. The final number in the string is $n+1$. For example, for an instance with 40 jobs, the string 0 2 7 40 41 means the job in position 1 is non-late, the jobs in positions 2 to 6 are late, the jobs in positions 7 to 39 are non-late, and the job in the last position is late.

3.4.2. Speedups for the Dynamic Program in Dynasearch. We now describe a speedup that avoids computing $F(\sigma_k^{(t-1)})$ for values of k corresponding to positions at the start of the sequence. Specifically, if the jobs in the first few positions of the sequences $\sigma^{(t-1)} = \sigma^{(t-2)}$ are identical, then the computation of $F(\sigma_k^{(t-1)})$ is identical to that of $F(\sigma_k^{(t-2)})$ for the values of k corresponding to these initial positions.

Presenting this argument more formally, consider the computation of $F(\sigma_k^{(t-1)})$ for $k = 1, \ldots, n$. Let h_t denote the largest index such that

$$\sigma^{(t-1)}(k) = \sigma^{(t-2)}(k) \qquad \text{for } k = 1, \ldots, h_t.$$

Then we must have that

$$F(\sigma_k^{(t-1)}) = F(\sigma_k^{(t-2)}) \qquad \text{for } k = 1, \ldots, h_t.$$

Accordingly, for iteration t of the dynasearch algorithm, we need to perform the recursion only for $k = h_t + 1, \ldots, n$.

4. Iterated Dynasearch

This section addresses the issue of how to design an iterated version of the dynasearch algorithm presented in the previous section. In Section 4.1, we discuss the general design of iterated local search algorithms that use restarts near local minima. In Section 4.2, we present the implementation details of the iterated dynasearch algorithm that we use in our computational experiments.

4.1. Iterated Local Search

A simple but effective procedure to explore multiple local minima, which can be implemented in any type of local search algorithm, is to perform multiple runs with the algorithm, each using a different starting solution. In the simplest form of this approach, these starting solutions are generated randomly, and do not rely on the results of previous runs of the local search algorithm. Typically, the starting solutions are chosen randomly, or by applying some constructive heuristic but with varying values of its parameters. We refer to this approach as *multi-start* local search.

A promising but relatively unexplored idea is to restart near a local minimum, rather than from a randomly generated solution. Under this approach, the next starting solution is obtained from the current local optimum (where the current local optimum is usually either the best local optimum found thus far, or the most recently generated local optimum) by applying a prespecified type of random move to it. We refer to such a move as *kick*, and to the approach as *iterated* local search. In this way, not all the good characteristics from previously found solutions are lost. Although some of the basic ideas of iterated local search appear in a study by Baxter (1981) for a depot location problem, the use of a randomized kick traces back to Baum (1986a, 1986b), who presents iterated 2-opt and 3-opt algorithms for the traveling salesman problem, in which a single random 2-opt move is used as a kick to move away from the current local optimum to the next starting solution. Although his computational results are discouraging, more recent research (for example, see Johnson 1990, Johnson and McGeoch 1997, Johnson et al. 2001, Martin et al. 1991, 1992, Martin and Otto 1996, Brucker et al. 1996, 1997, Lourenço 1995, and Lourenço and Zwijnenburg 1996) shows that iterated local search can be extremely competitive.

An iterated local search algorithm essentially performs a local search on the local optima. Following

the generation of an initial current solution, a traditional descent or dynasearch algorithm is applied to find a solution S which is a local optimum. If S is the first local optimum that is generated, then we define $S_C = S$ to be the current solution. When appropriate, the current best solution found thus far, which we denote by S_B, is updated.

The general form of the algorithm allows backtracking to occur. For backtracking, the current solution is set to be the best solution found thus far. The justification of backtracking is to ensure that much of the search is performed in "interesting" regions of the solution space. Thus, backtracking can help overcome decisions that direct the search towards inferior local optima. In cases where backtracking is not used, a decision is made as to whether to adopt the new local optimum S as the current solution, or to retain S_C.

Having decided on a current solution, a kick is applied. If the solution resulting from the kick has some unsatisfactory features, then it can be rejected before applying traditional descent or dynasearch to find a local optimum. In this case, another kick is executed, and the process is repeated until the kick is accepted. Then, a local optimum is found, and the entire procedure is repeated until a stopping criterion (usually based on computation time or the total number of iterations) is satisfied. Figure 1 displays the main features of iterated local search.

To date, backtracking does not appear to be used in iterated local search. Various criteria for accepting S are possible: accept S if it has a better objective value than S_C, as suggested by Johnson (1990); or adopt a simulated annealing acceptance criterion, as suggested by Martin et al. (1991, 1992).

Regarding the choice of kick, there is ample computational evidence that iterated local search is competitive only if the kick is sufficiently large to move to a solution that is not too close to the local optimum. If it is not, then the effect of the kick might be reversed in a single or small number of iterations, and the kick would literally lead nowhere—this is the most likely explanation of Baum's disappointing results. On the other hand, the kick should not be too large, or else the good characteristics of the previous local optimum are lost, and the procedure is then effectively multi-start rather than iterated

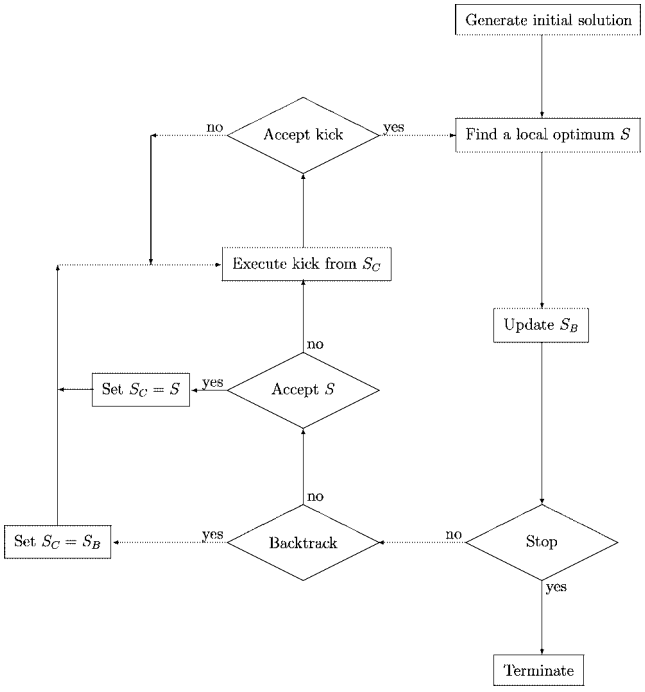

Figure 1 Overview of Iterated Local Search

local search. For a k-exchange neighborhood, an effective kick would then be a single (or several) $(k+1)$-exchange, or several k-exchanges. As an example, for the traveling salesman problem, both Johnson (1990) in his iterated 3-opt and iterated Lin-Kernighan algorithms, and Martin et al. (1991, 1992) in their iterated simulated annealing algorithm with the 3-opt neighborhood structure, use a specific 4-opt move as a kick.

4.2. Implementation of Iterated Dynasearch

In this section, we present our iterated dynasearch algorithm for the total weighted tardiness scheduling problem. To fully evaluate the effectiveness of the dynasearch concept, we compare the empirical performance of iterated dynasearch not only with the performance of the state-of-the-art multi-start tabu search algorithm of Crauwels et al. (1998), but also with iterated first-improve and iterated best-improve descent.

In our implementation of these three iterated algorithms, a kick is a series of random swap moves (that in general are not independent) from a previously generated local minimum. Even though the same underlying neighborhood is used for the kicks,

for a sufficiently long series of moves, the random swaps invariably produce a solution that is outside the region of attraction of the local minimum. We use the parameter α to denote the number of random swap moves in a kick. Based on the results of initial experiments, we use the parameter value $\alpha = 6$.

Our algorithms backtrack to the best solution found thus far every β iterations, where an iteration refers to finding a local optimum following a kick, and β is a parameter. For iterations in which backtracking does not occur, we always accept the new solution S to be the current solution. Our initial experiments indicate that $\beta = 5$ is a suitable value.

The performance of the algorithms is not particularly sensitive to small changes in the values of parameters α and β, and in some of the other design features. For instance, by varying α, we observe that using a kick with six, rather than five or seven random swap moves, produces only slightly better results. Moreover, for kicks based on random 3-exchanges, a single random 3-exchange is not very effective, but a kick comprising two random 3-exchanges is a reasonable alternative to six random swaps.

We have not yet exploited a known property of the total weighted tardiness problem: there exists an optimal solution in which non-late jobs are sequenced in non-decreasing order of due dates, or EDD order. In an attempt to reduce the computation time for dynasearch descents, we transpose adjacent non-late jobs if they are not in EDD order, during a number of iterations at the beginning of the algorithm. Then, for the remaining iterations, we diversify the search by transposing non-late jobs so that they no longer appear in EDD order. Specifically, for the first 100 iterations, we search the sequence from the beginning, transposing adjacent non-late jobs if they are not in EDD order. After the first 100 iterations, a similar procedure transposes adjacent non-late jobs, either if they are not currently in EDD order, or with probability 1/3 if they are currently in EDD order and their interchange would not cause either one to become late. All transposes are performed immediately prior to the kick.

Finally, we discuss the heuristic that is used to produce a starting solution for the iterated local search algorithms. For non-iterated versions of descent or dynasearch in which a single local minimum is to be generated, a better starting solution often leads to a better local minimum. However, when multiple local minima are generated, as in iterated local search algorithms, the quality of the starting solution has less effect on the quality of the final solution.

In our implementation, all searches start from the heuristic solution that is generated by the so-called Apparent Urgency (AU) rule. The AU rule is a constructive heuristic that is presented by Morton et al. (1984). It is selected for two reasons: first, it is used by Crauwels et al. (1998) so that using the AU rule makes comparisons easier; second, Potts and Van Wassenhove (1991) and Morton and Pentico (1993) find that the AU rule compares favorably with other constructive heuristics for the total weighted tardiness problem. The AU rule is a dynamic list scheduling heuristic that selects an unscheduled job j with the smallest AU_j value to occupy the first unfilled position of the sequence, where AU_j is defined by

$$AU_j = \frac{w_j}{p_j} \exp\left(-\frac{\max\{0, d_j - t - p_j\}}{k\bar{p}}\right).$$

In this expression, t is the sum of the processing times of the scheduled jobs, $\bar{p}$ is the average processing time of the jobs, and k is the so-called lookahead parameter, which is preset according to the tightness of the due dates. In our implementation, we follow Potts and Van Wassenhove (1991) by using $k = 0.5$ for TF = 0.2, $k = 0.9$ for TF = 0.4, and $k = 2.0$ for TF > 0.4, where $\mathrm{TF} = 1 - \sum_{j=1}^{n} d_j/(n\sum_{j=1}^{n} p_j)$ is a parameter of the problem instance that is known as the tardiness factor (see Section 5.1 for the values of TF that are used in our test instances).

5. Computational Experience

This section reports on our computational experience with randomly generated instances to evaluate and compare the empirical performance of the various local search algorithms. In Section 5.1, we discuss the design of the computational experiments, including the random instance generator. Section 5.2 compares the performance of multi-start first-improve

descent, multi-start best-improve descent, and multi-start dynasearch, as well as iterated versions of these algorithms. Finally, in Section 5.3, we compare iterated dynasearch with the multi-start tabu search algorithm of Crauwels et al. (1998).

5.1. Experimental Design

We use exactly the same set of randomly generated problem instances as Crauwels et al. (1998), who adopted the generation scheme proposed by Potts and Van Wassenhove (1985). Instances with $n = 40$, $n = 50$ and $n = 100$ were randomly generated, and for each job j ($j = 1, \ldots, n$), an integer processing time p_j was generated from the uniform distribution $[1, 100]$ and an integer processing weight w_j was generated from the uniform distribution $[1, 10]$. Different instance classes of different "hardness" were generated by using different uniform distributions for generating the due dates. For a given *relative range of due dates* RDD (RDD $= 0.2, 0.4, 0.6, 0.8, 1.0$) and a given *average tardiness factor* TF (TF $= 0.2, 0.4, 0.6, 0.8, 1.0$), an integer due date d_j was randomly generated from the uniform distribution $[P(1 - TF - RDD/2), P(1 - TF + RDD/2)]$, where $P = \sum_{j=1}^{n} p_j$. Five instances were generated for each of the 25 pairs of values of RDD and TF, yielding 125 instances for each value of n. These instances are available electronically at the OR library that is run by Beasley (1990).

In their study, Crauwels et al. (1998) attempt to solve the instances with $n = 40$ and $n = 50$ by applying the branch and bound algorithm of Potts and Van Wassenhove (1985). The algorithm successfully solves 124 and 103 problems out of 125 for the 40- and 50-job instances, respectively, on an HP 9000-G50 computer within a time limit of two minutes for each instance. There were no attempts to solve the 100-job instances since prohibitively large computation times were anticipated. To evaluate our results, we compare the solution values generated by the local search algorithms with the optimal solution values. However, for instances where the optimal solution value is unknown (which is the case for all of the 100-job instances), the comparison is with best known solution value, which is used as though it was the optimal value.

All of our descent and dynasearch algorithms were coded in C and run on a SPARC 5/110 server station. However, Crauwels et al. (1998) ran their algorithms, which were coded in C, on a HP 9000-G50 computer. The information collected by the Standard Performance Evaluation Corporation (1992) indicates that the HP 9000-G50 is a factor of 1.276 faster than the SPARC 5/110, and accordingly we use this conversion factor to make a direct comparison of computation times. As with any comparison between algorithms that are run on different machines, conversion factors for computation times only give a rough guide. Consequently, any conclusions that use the computation times obtained with our conversion factor of 1.276 are only valid if performance differences are sufficiently large.

We compare the performance of the various local search algorithms on basis of the following statistics:

ARPD = the average relative percentage deviation of the solution value found by the local search algorithm from the optimal (or best known) solution value;

MRPD = the maximum relative percentage deviation of the solution value found by the local search algorithm from the optimal (or best known) solution value;

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

ACT : SPARC = the average computation time in seconds on a SPARC 5/110 server station;

ACT : HP = the average computation time in seconds on a HP 9000-G50 computer;

NI = the number of iterations performed in multi-start and iterated descent and dynasearch, where an iteration refers to descending to a local minimum, and the number of moves performed for each start in the multi-start tabu search algorithm of Crauwels et al. (1998).

Table 4 Computational Results for Multi-Start Local Search Algorithms

		First-improve Descent				Best-improve Descent				Dynasearch			
n	ACT: SPARC	NI	ARPD	MRPD	NO	NI	ARPD	MRPD	NO	NI	ARPD	MRPD	NO
40	2	117	0.245	7.756	77.8	33	0.165	6.617	99.6	70	0.087	5.289	116.9
50	4	133	0.503	11.197	65.8	32	0.270	6.750	83.1	74	0.162	5.394	95.1
100	20	87	0.573	14.640	33.4	13	0.437	20.070	44.6	39	0.360	20.070	47.3

During our experimental work, we obtained better solutions for eight of the 100-job instances than those generated by Crauwels et al. (1998). Nevertheless, to facilitate a direct comparison of results, the best known solution values of Crauwels et al. (1998) are used when computing the above statistics. When we obtain a solution value that is better than the previous best known value, it is then treated as if the optimum or best known solution value of Crauwels et al. (1998) has been found, except that its relative percentage deviation is not zero but negative—so, such a solution actually reduces the average relative percentage deviation statistics for the local search algorithm. A further consequence of obtaining new best known solution values is that the numbers of optimal or best known solution values, as listed by Crauwels et al. (1998), are sometimes higher than would be the case if they were recalculated with respect to the new values.

5.2. Multi-Start and Iterated Dynasearch vs. First-Improve and Best-Improve Descent

This subsection compares the empirical performance of multi-start versions and iterated versions of first-improve descent, best improve descent and dynasearch. All results are obtained with the parameter values $\alpha = 5$ and $\beta = 5$ (see Section 4.2 for the definition of α and β), and without the transpose of adjacent jobs according the EDD criteria, as described in Section 4.2. Also, identical speedups (see Section 3.4) are applied in each algorithm, but they appear to be most effective in reducing the computation time of multi-start first-improve descent. All multi-start algorithms are run for 2, 4 and 20 seconds for the 40-, 50- and 100-job instances, respectively, and for the iterated algorithms these times are halved. (More computation time is allocated to the multi-start algorithms, since each descent requires many more moves to reach a local optimum than is the case for the iterated algorithms.) In each case, the results are obtained from the average of 10 independent runs.

Table 4 gives our computational results for multi-start versions of first-improve descent, best-improve descent, and dynasearch. Although multi-start dynasearch provides the best quality solutions, its superiority over multi-start best-improve descent is not substantial. Each descent in first-improve executes on average only about 6, 7 and 10 moves for the 40-, 50- and 100-job instances, respectively. By contrast, corresponding numbers of best-improve descent moves are 32, 40 and 86, and corresponding numbers of dynasearch moves are 15, 18 and 31, respectively. Note that a dynasearch move may correspond to several descent moves, so the numbers of moves for dynasearch are not directly comparable to those for descent. The small number of moves per descent for first-improve indicates that the search often becomes trapped in a local minimum fairly quickly, which explains the relatively poor performance of this algorithm. However, for best-improve descent and dynasearch, the numbers of moves tends to be sufficiently large to allow the search to find better quality local minima.

For iterated versions of the algorithms in which the restarts are not from random starting sequences but are obtained from kicks, the results are shown in Table 5. These results exhibit different characteristics from those in Table 4. Specifically, iterated dynasearch can be seen to outperform iterated first-improve and iterated best-improve descent on all performance measures: it achieves significantly lower ARPD and MRPD values, and it finds considerably more optimal (or best) solutions than the two iterated descent algorithms.

For all iterated algorithms and all instance sizes, the average numbers of moves per descent are between

Table 5 Computational Results for Iterated Local Search Algorithms

		First-improve Descent				Best-improve Descent				Dynasearch			
n	ACT: SPARC	NI	ARPD	MRPD	NO	NI	ARPD	MRPD	NO	NI	ARPD	MRPD	NO
40	1	109	0.080	2.350	95.0	115	0.090	6.018	108.3	119	0.049	5.338	121.2
50	2	111	0.386	10.686	74.2	140	0.181	6.800	88.9	143	0.044	2.776	111.5
100	10	78	0.447	15.851	36.0	123	0.291	18.391	47.1	122	0.041	1.834	82.5

5 and 8, which is slightly more than the number of random moves in the kick ($\alpha = 5$). The average numbers of moves per descent are less for best-improve than for dynasearch, which suggests that dynasearch allows more searching before becoming trapped in a local minimum, and is consistent with the better quality solutions found by dynasearch. Although the average number of moves per descent for first-improve is more than for best-improve and dynasearch for the 100-job instances, the higher quantity of moves cannot compensate for their lower quality.

One reason for the impressive performance of iterated dynasearch is its apparent ability to move between local minima. The dynasearch swap neighborhood is much larger than the traditional swap neighborhood, containing different basins of attraction around the same set of local minima. For the instances with 100 jobs, iterated dynasearch detects about six times more local minima with distinct objective values than iterated first-improve or iterated best-improve descent.

5.3. Iterated Dynasearch vs. Tabu Search

In this subsection, we report on our computational results that compare the empirical performance of iterated dynasearch with that of the state-of-the-art local search algorithm for the total weighted tardiness scheduling problem, namely the multi-start tabu search method of Crauwels et al. (1998). All information regarding the performance of the tabu search algorithm with five starts is taken directly from Crauwels et al. (1998). Table 6 provides results for the two algorithms. The iterated dynasearch algorithm is run with our recommended parameter values $\alpha = 6$ and $\beta = 5$ (note the change from the value $\alpha = 5$ that is used in the previous subsection to obtain our initial results), and results are obtained from the average of 10 independent runs. To allow a fair comparison with

Table 6 Computational Results for Iterated Dynasearch and Multi-Start Tabu Search

	Iterated Dynasearch					Multi-Start Tabu Search				
n	NI	ARPD	MRPD	NO	ACT:HP	NI	ARPD	MRPD	NO	ACT:HP
40	50	0.0150	1.8594	123.7	0.20					
	100	0.0001	0.0166	124.8	0.40					
	150	0.0000	0.0000	125.0	0.62					
						$2n^2/5$	0.00	0.33	118	1.32
						$4n^2/5$	0.00	0.17	123	2.64
50	100	0.0021	0.1738	122.3	0.64					
	200	0.0006	0.0754	124.1	1.33					
	450	0.0002	0.0193	124.8	2.86	$2n^2/5$	0.01	0.28	113	2.95
	900	0.0000	0.0000	125.0	5.76	$4n^2/5$	0.00	0.16	118	5.92
100	100	0.0077	0.4117	107.2	2.80					
	200	0.0031	0.2357	116.6	5.72					
	500	0.0012	0.1410	120.9	13.95					
	1300	0.0001	0.0534	123.2	36.40	$2n^2/5$	0.04	4.39	103	37.6

tabu search using a similar computation time, iterated dynasearch is run for different numbers of iterations. The rows of the table are aligned according to the average computation time values. Note that our computation times are converted to equivalent times on the HP 9000-G50 computer by multiplying by a factor 1.276 to account for our slower computer.

Table 6 clearly shows that iterated dynasearch generates better solutions than the tabu search algorithm, in considerably shorter computation times. The speedups of Section 3.4 contribute significantly to the reduced computation times. From our computational results, we conclude that iterated dynasearch is superior to all other known local search algorithms for the total weighted tardiness scheduling problem.

Even without the speedups, we claim that dynasearch is preferred. Typically, the speedups reduce the run times of dynasearch for the instances with 40, 50 and 100 jobs by factors of 2.6, 3,4 and 5.5, respectively. However, a tabu search algorithm with the appropriate speedups must yield smaller reduction factors, since some of saving in computation time can only be achieved with a search for improving moves. Even allowing for reduced computation times for tabu search, the quality of solutions produced by tabu search remains lower than that for dynasearch under similar run times.

6. Concluding Remarks

The contribution of this paper is twofold. First, a general contribution is the introduction of a new local search technique known as dynasearch. Through its use of dynamic programming, dynasearch explores a neighborhood of exponential size in polynomial time. Since each dynasearch move corresponds to a series of moves in a traditional local search algorithm, dynasearch has a lookahead capability that is not present in these previous methods.

Dynasearch is applicable to a variety of problems for which solutions are naturally represented as sequences. Congram (2000) reports on dynasearch algorithms for the traveling salesman problem and the linear ordering problem that use multiple 2-opt and 3-opt moves, and multiple insert moves, respectively. For each problem, the overall improvement to the objective function that results from the independent moves is equal to the sum of the improvements from the individual moves. When defining independent moves for a particular problem, the definition of independence should be chosen so that individual improvements sum in this way. When we cannot find a suitable definition of independence that produces individual improvements which can be summed, dynasearch cannot be used. A common feature of the total weighted tardiness scheduling problem, the traveling salesman problem and the linear ordering problem is that each can be solved to optimality by a dynamic programming algorithm, although the algorithms have an exponential number of states. Another way of viewing dynasearch is that these dynamic programs are modified according to the current sequence so that some state transitions are forbidden and some states are merged so that a polynomial number of states remains.

For a given problem type, a variety of neighborhoods can be used a dynasearch. For example, in the total weighted tardiness scheduling problem, independent insert moves (an insert move removes a job from its current position and inserts it elsewhere) can be used instead of independent swap moves. More significantly, the flexibility of the approach allows independent swap and insert moves to be used in combination.

The second contribution relates specifically to the application of dynasearch to the single-machine total weighted tardiness scheduling problem. An iterated dynasearch algorithm for this problem performs significantly better than all other known local search algorithms, including the state-of-the-art tabu search algorithm of Crauwels et al. (1998), with respect to both solution quality and computation time. The explanation for iterated dynasearch finding better solutions is that, after performing a sufficiently large number of random moves from a local minimum, dynasearch stands a much better chance of descending into a different local minimum than traditional descent algorithms. In this way, iterated dynasearch is able to explore more distinct local minima. This performance is entirely attributable to the much larger size of the dynasearch neighborhood: the dynasearch neighborhood contains an exponential number of

solutions, while the traditional descent neighborhood contains only a polynomial number of solutions. One explanation for iterated dynasearch being faster is the use various speedups that sometimes reject a move without a complete evaluation of the total weighted tardiness.

In a recent study that was undertaken in parallel with our work, Stützle et al. (1999) apply an ant colony approach to the total weighted tardiness scheduling problem. Essentially, the ant colony generates sequences of jobs through the use of a priority rule with changing parameters. These sequences form the input into a descent algorithm that uses the swap and insert neighborhoods in combination. Although their solutions are of slightly better quality than the ones obtained with our dynasearch algorithm, this is at the expense of extra computation time. As observed above, it is straightforward to extend dynasearch so that independent swap moves and insert moves are used in combination. A fairer comparison of the ant colony approach would be with such an extended dynasearch algorithm.

This paper provides convincing evidence that iterated dynasearch is a powerful new local search technique for a variety of combinatorial optimization problems. One topic for future research is to characterize the types of problems to which dynasearch can be applied. Another topic is to obtain further empirical evidence on the ability of iterated dynasearch to find high-quality solutions of other combinatorial optimization problems, and on its ability to compete with, or outperform, other local search algorithms for these problems.

Acknowledgment

The authors are grateful to two anonymous referees whose comments have helped in the presentation of our work.

References

Ahuja, R. K., O. Ergun, J. B. Orlin, A. Punnen. 1999. A survey of very large-scale neighborhood search techniques. Working Paper, Operations Research Center, Massechusetts Institute of Technology, Cambridge, MA.

Anderson, E. J., C. A. Glass, C. N. Potts. 1997. Machine scheduling. E. H. L. Aarts, J. K. Lenstra, eds. *Local Search in Combinatorial Optimization*. Wiley, Chichester, UK, 361–414.

Applegate, D., R. Bixby, V. Chvatal, W. Cook. 1999. Finding tours in the TSP. Report No. 99885, Forschungsinstitut für Diskrete Mathematik, Universität Bonn, Bonn, Germany.

Balas, E. 1999. New classes of efficiently solvable generalized traveling salesman problems. *Annals of Operations Research* **86** 529–558.

Balas, E., N. Simonetti. 2001. Linear time dynamic-programming algorithms for new classes of restricted TSPs: a computational study. *INFORMS Journal on Computing* **13** 56–75.

Baum, E. B. 1986a. Iterated descent: a better algorithm for local search in combinatorial optimization problems. Technical Report, Caltech, Pasadena, CA.

Baum, E. B. 1986b. Towards practical 'neural' computation for combinatorial optimization problems. J. S. Denker, ed. *Neural Networks for Computing*. Proceedings AIP Conference 151, American Institute of Physics, New York, 53–58.

Baxter, J. 1981. Local optima avoidance in depot location. *Journal of the Operational Research Society* **32** 815–819.

Beasley, J. E. 1990. OR library: distributing test problems by electronic mail. *Journal of the Operational Research Society* **41** 1069–1072.

Brucker, P., J. Hurink, F. Werner. 1996. Improving local search heuristics for some scheduling problems—I. *Discrete Applied Mathematics* **65** 97–122.

Brucker, P., J. Hurink, F. Werner. 1997. Improving local search heuristics for some scheduling problems. Part 2. *Discrete Applied Mathematics* **72** 47–69.

Carlier, J., P. Villon. 1990. A new heuristic for the traveling salesman problem. *RAIRO Recherche Opérationelle* **24** 245–253.

Congram, R. K. 2000. Polynomially searchable exponential neighbourhoods for sequencing problems in combinatorial optimisation. Ph.D. thesis, University of Southampton, UK.

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

Deĭneko, V., G. J. Woeginger. 2000. A study of exponential neighborhoods for the traveling salesman problem and for the quadratic assignment problem. *Mathematical Programming* **87** 519–542.

Hurink, J. 1999. An exponential neighborhood for a one-machine batching problem. *OR Spektrum* **21** 461–476.

Johnson, D. S. 1990. Local optimization and the traveling salesman problem. M. S. Paterson, ed. *Automata, Languages and Programming*. Lecture Notes in Computer Science 443, Springer, Berlin, Germany, 446–461.

Johnson, D. S., L. A. McGeoch. 1997. The traveling salesman problem: a case study. E. H. L. Aarts, J. K. Lenstra, eds. *Local Search in Combinatorial Optimization*. Wiley, Chichester, UK, 215–310

Johnson, D. S., J. L. Bentley, L. A. McGeoch, E. E. Rothbergh. 2001. Near-optimal solutions to very large traveling salesman problems. Monograph in preparation, AT&T Labs, Florham Park, NJ.

Lawler, E. L. 1977. A "pseudopolynomial" algorithm for sequencing jobs to minimize total tardiness. *Annals of Discrete Mathematics* **1** 331–342.

Lenstra, J. K., A. H. G. Rinnooy Kan, P. Brucker. 1977. Complexity of machine scheduling problems. *Annals of Discrete Mathematics* **1** 343–362.

Lourenço, H. R. 1995. Job-shop scheduling: computational study of local search and large-step optimization methods. *European Journal of Operational Research* **83** 347–364.

Lourenço, H. R., M. Zwijnenburg. 1996. Combining the large-step optimization with tabu-search: application to the job-shop scheduling problem. I. H. Osman, J. P. Kelly, eds. *Meta-Heuristics: Theory and Applications*, Kluwer, Norwell, MA, 219–236.

Martin, O., S. W. Otto. 1996. Combining simulated annealing with local search heuristics. *Annals of Operations Research* **63** 57–75

Martin, O., S. W. Otto, E. W. Felten. 1991. Large-step Markov chains for the traveling salesman problem. *Complex Systems* **5** 299–326.

Martin, O., S. W. Otto, E. W. Felten. 1992. Large-step Markov chains for the TSP incorporating local search heuristics. *Operations Research Letters* **11** 219–224.

Matsuo, H., C. J. Suh, R. S. Sullivan. 1987. A controlled search simulated annealing method for the single machine weighted tardiness problem. Working Paper 87-12-2, Department of Management, University of Texas, Austin, TX.

Morton, T. E., D. W. Pentico. 1993. *Heuristic Scheduling Systems with Applications to Production Systems and Project Management*. Wiley, Chichester, UK.

Morton, T. E., R. M. Rachamadugu, A. Vepsalainen. 1984. Accurate myopic heuristics for tardiness scheduling. GSIA Working Paper No. 36–83–84, Carnegie-Mellon University, Pittsburgh, PA.

Potts, C. N., L. N. Van Wassenhove. 1985. A branch and bound algorithm for the total weighted tardiness problem. *Operations Research* **33** 363–377.

Potts, C. N., L. N. Van Wassenhove. 1991. Single machine tardiness sequencing heuristics. *IIE Transactions* **23** 346–354.

Sarvanov, V. I., N. N. Doroshko. 1981a. The approximate solution of the travelling salesman problem by a local algorithm that searches neighborhoods of exponential cardinality in quadratic time (in Russian). *Software: Algorithms and Programs* **31** Mathematical Institute of the Belorussian Academy of Sciences, Minsk, Belarus, 8–11.

Sarvanov, V. I., N. N. Doroshko. 1981b. The approximate solution of the travelling salesman problem by a local algorithm with scanning neighborhoods of factorial cardinality in cubic time (in Russian). *Software: Algorithms and Programs* **31** Mathematical Institute of the Belorussian Academy of Sciences, Minsk, Belarus, 11–13.

Standard Performance Evaluation Corporation. 1992. Performance results. 10754 Ambassador Drive, Suite 201, Manassas, VA 20109. http://performance.netlib.org/performance/html/new.spec.cint92.col0.html

Stützle, T., M. den Besten, M. Dorigo. 1999. Ant colony optimization for the weighted tardiness problem. Technical Report IRIDIA/99-16, Université Libre de Bruxelles, Brussels, Belgium.

Accepted by Jan Karel Lenstra; received January 2000; revised March 2001, July 2001; accepted August 2001.