Journal of Heuristics, 4: 141–160 (1998)


# Design and Evaluation of Tabu Search Algorithms for Multiprocessor Scheduling

ARNE THESEN
*Department of Industrial Engineering, University of Wisconsin—Madison, 1513 University Ave, Madison, WI, USA, 53706*

*Abstract*

Using a simple multiprocessor scheduling problem as a vehicle, we explore the behavior of tabu search algorithms using different tabu, local search and list management strategies. We found that random blocking of the tail of the tabu list always improved performance; but that the use of frequency-based penalties to discourage frequently selected moves did not. Hash coding without conflict resolution was an effective way to represent solutions on the tabu list. We also found that the most effective length of the tabu list depended on features of the algorithm being used, but not on the size and complexity of the problem being solved. The best combination of features included random blocking of the tabu list, tasks as tabus and a greedy local search. An algorithm using these features was found to outperform a recently published algorithm solving a similar problem.

**Key Words:** tabu search, scheduling, performance evaluation

## 1. Introduction

Over the past 12 years, the tabu search has established its position as an effective meta-algorithm guiding the design and implementation of algorithms for the solution of large scale combinatorial optimization problems in a number of different areas (Glover (1986), Glover and Laguna (1997)). A key reason for this success is the fact that the algorithm is sufficiently flexible to allow designers to exploit prior domain knowledge in the selection of parameters and sub-algorithms. Such prior knowledge is frequently obtained by repeatedly solving a few representative test cases using a spectrum of different techniques, attributes and/or parameters. Frequently referred to as *target analysis*, an application of this learning approach is described in Laguna and Glover (1993). In this paper we show that this approach can be used to select and fine-tune sub-algorithms for a simple but still effective tabu search algorithm.

The problem of interest in this research is the multiprocessor scheduling problem. Briefly, this is the problem of assigning $n$ independent tasks, each with a duration $t_i\,(i = 1, \ldots, n)$, to $m$ identical processors such that the completion time for the latest processor (i.e., the makespan) is minimized. The shortest schedule would be one where all processors completed their work at the same time. The makespan of this schedule will be referred to as the *ideal schedule duration* $(T^*)$. $T^*$ is readily computed as $T^* = \sum t_i / m$. The multiprocessor scheduling problem is a slight variation on the traditional bin packing problem where the objective would be to find the number of processors needed to complete the project by a given due date (Johnson et al. (1974), Kämpke (1988)). The state-of-the-art of the use of

meta-heuristics to solve this class of problems is rapidly evolving. For an overview of recent contributions see Osman and Kelly (1996), and Glover and Laguna (1997). Overviews of applications in the production scheduling area is given in Barness, Laguna, and Glover (1995) and Blazewicz et al. (1996).

The remainder of this paper is organized as follows: In Section 2 we introduce a number of different sub-algorithms for managing and searching tabu lists. Among the topics introduced here are the use of hash coded tabus and the use of randomly blocked tabu list tails. In Section 3 we introduce and fine-tune eight different algorithms implementing different combinations of the sub-algorithms introduced in Section 2. The performance of the resulting algorithms is evaluated, and the best performer is identified. We will see that, when task durations are expressed as three to four digit integers, a well-designed tabu search algorithm can easily find the optimal solution to problems with up to 10,000 tasks and 500 processors. In Section 4 we evaluate the performance of the winning algorithm when solving more difficult problems. Results of a comparison with previously published results for a related algorithm are also given. We will see that the best of the algorithms developed here used an average of about 60% fewer iterations to solve similar problems. Finally, a summary and conclusions are given in Section 5.

## 2. Elements of a tabu search

The tabu search algorithm combines a few simple ideas into a remarkably efficient framework for heuristic optimization. Among the main elements of this framework are:

- A (usually) *greedy local* search; the next solution is usually the best not-yet visited solution in the current neighborhood.
- A mechanism (*the tabu list*) discouraging returns to recently visited solutions.
- A mechanism that changes the solution path (perhaps by a random move) when no progress has been made for a long time.

In addition, many tabu search algorithms incorporate other features such as flexible memory structures and dynamic aspiration criteria (Glover (1995)). A rough overview of a conceptual tabu search algorithm is given figure 1. For difficult searches, techniques such as influential diversification (Hübscher and Glover (1994)) may be used to extend the duration and scope of the search.

Although a tabu search is conceptually simple, any implementation of an efficient tabu search algorithm is problem specific, and no generic tabu search software is available at this time. Among the issues faced by designers of tabu search algorithms are:

- The nature of the information included in the tabu list.
- The way the tabu list is organized.
- The lengths of the tabu lists.
- The types of moves used to create new solutions.
- The implementation of the local greedy search algorithm.
- Strategies for diversifying the search when no progress has been made for a while.

```
Initialize
  Identify initial Solution
  Create empty TabuList
  Set BestSolution = Solution
  Define TerminationConditions
  done = FALSE
Repeat
   if value of Solution > value of BestSolution then
     BestSolution = Solution
   if no TerminationConditions have been met then begin
     add Solution to TabuList
     if TabuList is full then
        delete oldest entry from TabuList
     find NewSolution by some transformation on Solution
     if no NewSolution was found or
        if no improved NewSolution was found for a long time then
           generate NewSolution at random
     if NewSolution not on TabuList then
        Solution = NewSolution
   end
   else
     done = TRUE
until done = TRUE
```

*Figure 1.* A rough overview of a conceptual tabu search algorithm.

In addition, many of the elements used in a tabu search are themselves heuristic algorithms, most of which rely on some carefully selected parameter value for optimal performance. Few generalizable guidelines are available for the choice of such parameters. These concerns are discussed later in the paper. But first, let us review the main elements of the tabu search algorithm.

### *2.1. Solutions, moves and neighborhoods*

Our strategy is to make successive changes to an initial solution such that a sequence of improved solutions is obtained. Applying a commonly used strategy for combinatorial problems that also was used by Barnes and Laguna (1993) and Hübscher and Glover (1994), we will use two types of *moves* to generate new solutions:

- *Transfer*: A task is assigned to another processor.
- *Interchanges*: Two tasks performed by different processors are swapped.

Algorithms using transfers are simpler, and easier to implement, than those using interchanges. However, when a move is to be made between two processors performing $n_1$ and $n_2$ tasks respectively, the number of transfers available is $n_1 + n_2$, while the number of interchanges available is $n_1 \times n_2$. Given this larger number of choices, it is more likely that the interchange approach provides a better candidate for the next move. On the other hand,

interchanges do not change the number of tasks assigned to different processors. Hence, a pure interchange-based algorithm is unlikely to find the optimum allocation of tasks. We will use a combined strategy that selects the best approach for each iteration. Barnes and Laguna (1993) noted that the use of two types of moves markedly improved the performance of the search procedure. Our evaluations confirm this finding.

The set of new solutions that can be developed by a move from a given solution is referred to as that solution's *neighborhood*. A neighborhood has $n \times (m-1)$ members when transfers are used. Hence, the local neighborhood can be quite large, and an exhaustive search of the entire neighborhood at every move is not feasible. Instead, we chose to restrict our search to moves between the busiest and the least busy processor. Since any move reducing the project duration must reduce amount of work assigned to the busiest processor, this restriction does not reduce our ability to find a better solution (if one exists).

### *2.2. The local search*

During the local search, the current neighborhood is explored, and a suitable move is selected. Ideally, all promising moves should be evaluated. But, for large neighborhoods, this may not be feasible. Instead, the search may be terminated as soon as a good move is found. The criteria used in the local search may differ from the global criteria. For example, when looking for a job to transfer between two processors, let us look for the job that minimizes the difference in their total task durations. The search can be deterministic or probabilistic. In the first case, the best move is always picked. In the second case, a set of probabilities is computed from the values of all moves, and a move is selected at random using these probabilities.

***Evaluation of moves.*** As a simplification, we consider only the effect that a move has on the completion times for the busiest and least busy processor. In this case, the best move is one where identical completion times are reached for the two processors. We measure the value of a move by how far away the resulting latest completion time is away from this ideal time. Note that this completion time may not be the resulting project completion time as another processor may take over the role as the busiest processor.

For an interchange, the value of a move is computed as the difference between the resulting latest completion time and the ideal completion time:

$$Z_{ij} = \text{Max}\{(d_b - t_{b,i} + t_{l,j}), (d_l + t_{b,i} - t_{l,j})\} - (d_b - d_l)/2, \tag{1}$$

where

$d_k$ = sum of durations of tasks assigned to processor $k$,
$b$ = index of busiest processor (i.e., $d_b = \max(d_k), k = 1, \ldots, m$),
$l$ = index of least busy processor (i.e., $d_l = \min(d_k), k = 1, \ldots, m$),
$t_{k,i}$ = duration of the $i$th task assigned to processor $k$,
$Z_{ij}$ = value of interchanging the $i$th task assigned to processor $b$ with the $j$th task assigned to processor $l$.

Note that we seek to find a move that minimizes the difference in durations between the

two processors. This was found to be more effective than related functions that would seek to bring either processor closer to the ideal schedule duration.

***Selection of moves.*** Under a *greedy search*, we are simply looking for the best available non-tabu move leading to a non-solution. To find this solution we scan through each available move, computing the resulting $Z$ value, and capturing each improved solution as it is encountered. By using a doubly linked data structure where tasks are linked in increasing order of duration for each processor, and where processors are linked in increasing order of assigned workload, this scanning is efficiently implemented, and the scan is terminated well before all moves are evaluated.

Under a *probabilistic search*, moves are assigned probabilities in inverse proportion to their relative gain. A move is then selected at random using these probabilities. The following procedure is used:

1. For each move $k$ compute relative weights:

$$x_k = \frac{1}{z - z_{\text{best}} + 1} \tag{2}$$

2. For each move $k$ compute probabilities:

$$p_k = \frac{x_k}{\sum x_i}$$

3. Let $u$ be a random number from a uniform distribution on (0, 1)
4. Select move $m$ such that

$$\sum_{i=0}^{m} p_i \leq u < \sum_{i=0}^{m+1} p_i$$

The quantity 1 was added in Eq. (2) to avoid division by 0 for the case where the gain is 0.

A probabilistic search cannot be implemented as efficiently as deterministic search. This is because (ideally) a complete list of all moves must be developed, and because two full and one partial pass must be made through this list for each iteration. Efficiencies may be gained by shortening the list, and by drawing $u$ before all $p_k$'s are computed. No such improvements were explored in the research reported here.

***Penalizing moves that may cause cycling.*** If a certain task has been moved a disproportionate number of times, it is unlikely that moving this task again will add significantly to the value of a solution. Rather, such moves are likely to be a symptom of cycling. To discourage such moves, and the resulting cycling, we may penalize moves that use this task. This can be done by adding a penalty to the value of a proposed solution proportional to the ratio between the number of times the tasks included in this move have been moved and

the average number of times all parts have been moved:

$$Z^*_{ij} = V + \frac{t_{b,i} \times n_{b,i}}{N} + \frac{t_{l,j} \times n_{lj}}{N} \tag{3}$$

where

$Z^*_{ij}$ = the modified value of the move,
$n_{k,i}$ = count of moves for the $i$th task assigned to processor $k$,
$N$ = count of moves for all tasks.

Using $Z^*_{ij}$, in place of $Z_{ij}$ will encourage the exploration of other, possibly more promising, areas of the solution space. One difficulty in using frequency-based penalties is that it is unclear what weight should be assigned to these penalties. On the other hand, Taillard (1994) reports on experiments where in every instance, it was possible to improve the efficiency of a search by using frequency-based penalties. Our evaluations confirm this observation. However, for the algorithms evaluated here, each iteration became more time consuming, and no improvements in solution times were observed.

### *2.3. Tabus and tabu lists*

A key premise of the tabu search is that usually it is better to visit new solutions than to return to recently visited ones. Toward this end, lists of recently visited solutions and/or recently used moves are maintained. Moves are disallowed if they, or the solution they lead to, are found on these lists. Items placed on such a *tabu list* remain there for a fixed number of iterations (equal to the length of the list). For the present problem, let us explore the use of both solutions and task moves as tabus. We will see that there is a significant difference in performance between the two.

***Tasks as tabus.*** The use of a task as a tabu has the advantage that a separate tabu list is not required. Instead, each task is marked by the time of its most recent move. This simplifies implementation. Reentry to a recently visited solution is now avoided by preventing tasks from being moved back to their previously assigned processor. However, the task is also prohibited from moving to other processors. This reduces the size of the neighborhood, and, perhaps, the efficiency of the search. Also, after a while, we may have moved all of the tasks assigned to a given processor to a different processor, potentially recreating a tabu solution using different processors. To avoid these situations, it is important not to restrict tasks from being moved for too long.

***Solutions as tabus.*** When solutions are tabus, sufficient information about previous solutions must be saved, such that entry into similar solutions can be recognized. A special challenge for the present problem is the fact that solutions are not unique. Since the processors are assumed to be identical, it does not matter which processor performs a given set of tasks. All such similar solutions should therefore also be detected.

1. For each processor $k$ compute:

$$sumsqk = \sum_{i=1}^{n_k} t_{ki}^2$$

where

$t_{ki}$ = The $i$th task assigned to processor $k$
$n_k$ = Number of tasks assigned to processor $k$

2. Sort processors such that:

$$sumsq_i < sumsq_j \quad \text{if } i < j \quad \text{(i.e., processors are ordered by workload)}$$

3. Compute the hash code:

$$h = \sum_{k=1}^{m} k \times sumsq_k$$

where

$h$ = Hash code value assigned to this solution

4. Place $h$ in a sorted list of hash codes. Mark $h$ by the time of insertion.

*Figure 2*. Hash coding scheme for tabu solutions.

For a large problem involving perhaps 10,000 tasks, and 1,000 processors, it is prohibitively expensive to store complete information about past solutions. Instead, let us use hash coding (Knuth (1973), Glover and Laguna (1997)) to reduce the relevant aspects of previous solutions to a single number. Now, an ordered list of hash codes, each representing a different tabu solution, can be constructed, and a search can easily be conducted to determine if a proposed solution is on the list. Figure 2 presents the hash coding scheme used in this study. Briefly, the hash code is computed as the biased sum of the sum-of-squares of durations of tasks assigned to individual processors. Although step 2 calls for processors to be sorted in increasing order of assigned workloads, this step is not needed in practice. This is because processors already are ordered in this sequence to avoid having to deal with the large number of solutions that differs only in the identity of processors performing specific sets of tasks.

Note that no attempt is made to resolve conflicting hash codes (i.e., when different solutions are hashed to the same code). This is not a cause for concern when new entries are added to the list, as the new entry simply replaces the old one. However, it may cause a problem when a proposed solution is hashed to the same value as one obtained for a different solution currently on the list. In this case, the proposed new solution will erroneously be rejected. However, the likelihood that two recent solutions have the same hash code is quite small; and, in our tests, computer programs dealing correctly with hash code collisions were significantly slower than those that did not. We will see that algorithms using hash-coded solutions as tabus can be quite efficient, and that this efficiency is relatively insensitive to the choice of tabu list length.

***List length.*** The length of the tabu list ($L$), is a critical parameter in most tabu search algorithms. We will see that the wrong choice of $L$ may lead to a very inefficient algorithm.

Glover (1986) suggested that 7 would be a good value for $L$. Anderson et al. (1993) reported that list lengths between 7 and 15 worked well for a path assignment problem. Løkketangen (1995) cites a case where the most efficient algorithm included more than 200 items in the list. Morton and Pentico (1993) suggest that keeping all solutions on the list works best for scheduling problems. Taillard (1994) reports on the successful use of lists that randomly change in length at certain points of time. It appears that the most efficient length of the list depends on the problem being solved, and the algorithm being used. In Section 3 we show that long lists worked well for the test problems used here when solutions were tabus, while they were counter productive when tasks were tabus.

***List tenure and blocking.*** Items usually stay on the tabu list for a number of iterations equal to the length of the tabu list, and most algorithms consider an item to be a tabu throughout its tenure on the tabu list. However, in some cases it is fruitful to make exceptions. For example, Hübscher and Glover (1994) use a scheme where different portions of the tabu list are blocked out (i.e., made inaccessible) at different times to encourage intensification and diversification. Here we will investigate a significantly simpler blocking scheme. Specifically, we introduce the concept of an *accessible list* with a random length $(A_i)$ that is established separately for each iteration. The accessible list is a subset of the entire list, and the entire list is maintained and updated as before. An item is considered a tabu for iteration $i$ if it were given tabu status at iteration $(i - A_i)$ or later. At a later iteration, $A_i$ may have a smaller value, and an item previously considered to be a tabu, may not be a tabu at this time. This approach differs from the periodically changing random list lengths used by Taillard (1994) in that a list of fixed length is always maintained, and the length of the accessible subset of this list is reevaluated for each iteration.

The length of the accessible list for iteration $i(A_i)$ is a random quantity computed as follows:

$$A_i = A^{\text{low}} + u \times (A^{\text{high}} - A^{\text{low}}) \tag{4}$$

where

$A_i$ = length of the list accessible for the $i$th iteration,
$A^{\text{low}}$ = a constant giving the lower bound on $A_i$,
$A^{\text{high}}$ = a constant giving the upper bound on $A_i$,
$u$ = random variable uniformly distributed between zero and one.

Extensive experiments have shown that the best value for $A^{\text{low}}$ is one, and that the best value for $A^{\text{high}}$ is $L$ (the length of the list). We will see that the use of an accessible list of random length improves the performance of all tabu search algorithms studied in this study.

### *2.4. Diversification*

Occasionally, if the present solution path does not appear to be promising, it may be efficient to abandon the local search and to use a different mechanism to obtain the next solution.

*Table 1.* Common termination criteria for heuristic optimization algorithms.

| Criteria | Rule |
|---|---|
| Solution quality | Stop when solution quality exceeds a lower bound by no more than a given value |
| Gradient | Stop when there has been no improvement for $N$ iterations |
| Total iteration count | Stop after $N$ iterations |
| Execution time | Stop after $T$ minutes |

The next move could be selected at random, or, the algorithm could be restarted from a previously encountered good solution. In this case, measures must be taken to ensure that the path that was taken from this solution last time is not taken again. This can be done by altering the makeup of the old tabu list, or by making the next move at random. Hübscher and Glover (1994) discuss how influential diversification can be used to extend the scope of the search space for problems similar to the ones studied here.

### *2.5. Termination*

A common difficulty with all heuristic optimization algorithms is the fact that it is usually not possible to determine if the optimal solution has been reached. As a result, a number of different termination criteria are commonly used together (Table 1).

## 3. The learning phase

Here we first describe eight different algorithms implementing different permutations of the sub-algorithms described in the previous sections. We then present an empirical analysis leading to the identification of recommended list lengths for each algorithm. Finally, an evaluation of the resulting tuned algorithms leads to the identification of the most efficient implementation. This algorithm is then further analyzed in the following section.

### *3.1. Eight algorithms*

A broad outline of a tabu search algorithm for solving the multiprocessor scheduling problem is given in figure 3. The key options to be implemented within this framework were discussed in the previous section:

- Type of tabus (solutions or tasks),
- Selection of the next move (greedy or probabilistic search),
- Tabu list (use entire list or random subset).

```
Initialize
    Let i = 0
        L = maximum length of tabu list
        m = the number of processors
        n = the number of tasks
        M_i = −99999   for i = 1, ..., n (last time task i was moved)
        W = {W_1, W_2, ..., W_m}: the initial solution
        d_k = sum of duration of tasks assigned to processor k
        d_best = infinity
        done = FALSE
  Repeat
    Let i = i + 1
        m = index of processor finishing last. That is, d_m = max(d_k) for k = 1, ..., m
        d = d_m
    If d < d_best then
        d_best = d
        W_best = W
    If no termination condition is met then
        Add W to tabu list
        If length of list > L then
            delete oldest entry from list
        Let j = index of processor finishing first
        Find non-tabu task r in P_m., and non-tabu task s in P_j that, when
            interchanged minimizes abs(P_m., −P_j)(s may be zero)
        If no task (s) were found then
            let r be the index of a non-tabu task selected at random
        Move task r to processor j
        If s > 0 then
            Move task s to processor m
        Let M_r = i
            M_s = i
        Let W = the current solution
    else
        done = TRUE
  until done = TRUE
  terminate
```

*Figure 3.* Outline of a tabu search algorithm for multiprocessor scheduling.

Since these options are independent, and, since two alternatives were offered for each, a total of eight different algorithms can be constructed. These eight algorithms were implemented in Microsoft Visual C++ and run under Windows 95 (Table 2). All eight algorithms use a single parameter ($L$). Recommended values for this parameter are established in the next section. A traditional greedy LPT algorithm (Graham (1969)) was used to develop initial solutions. Briefly, the tasks were first sorted in decreasing order of their durations. Then, they were sequentially assigned to the least busy processor until all tasks were assigned. The resulting solutions were frequently within one percent of the ideal schedule length. This is consistent with the bounds presented in Graham (1976).

*Table 2.* Eight different implementations of the tabu search algorithm given in figure 2.

| Algorithm | Tabu | Search | Length of available tabu list |
|---|---|---|---|
| SGF | Solutions | Greedy | Fixed = $L$ |
| SGR | Solutions | Greedy | Random = U(l, $L$) |
| SPF | Solutions | Probabilistic | Fixed = $L$ |
| SPR | Solutions | Probabilistic | Random = U(l, $L$) |
| TGF | Tasks | Greedy | Fixed = $L$ |
| TGR | Tasks | Greedy | Random = U(l, $L$) |
| TPF | Tasks | Probabilistic | Fixed = $L$ |
| TPR | Tasks | Probabilistic | Random = U(l, $L$) |

### *3.2. Searching for the best length of the tabu list*

In order to explore the effect of the length of the tabu list ($L$) on performance, we measured the effort required by algorithms to solve thousands of random problems using different values of $L$. The resulting data was then analyzed for the purpose of finding the best ($L^*$) value of $L$ for each algorithm. Particular attention was paid to the fact that the value of $L^*$ might change with the size and complexity of the problem. Two types of analysis were performed:

- *Graphical analysis*, plots of solution times versus $L$ were inspected for trends.
- *Nonlinear approximations*, analytic functions were fitted and solved for $L^*$.

A random problem generator was used in all evaluations. This generator first produced the requested number (between 500 and 10,000) of random task durations drawn from an exponential distribution with a mean of 1,000. These durations were then rounded up to the next integer value. Finally, all durations were proportionally adjusted such that the average task duration was equal to the expected task duration. To ensure that a known optimal solution existed, the number of processors for each problem was set such that at least 20 tasks were expected for each processor. Under these conditions, the ideal schedule length was always reached. (More difficult problems including problems with uniformly distributed, noninteger durations are evaluated in Section 4.)

Since optimal solutions were reached for all problems used in these evaluations, we were able to adopt as performance measures both the time (in seconds), and the number of iterations required to reach the optimal solution. In the remainder of this section the recommended list lengths for each algorithm are established. In the following section, we evaluate the performance of the resulting algorithms.

***Graphical analysis.*** We first collected data on the time required by each algorithm to solve 1,950 random problems, each with 5,000 tasks and 250 processors. All algorithms solved

*Table 3.* Effect of the length of the tabu list ($L$) on the time in seconds to solve problems with 5,000 tasks and 250 processors. Random, integer processing times drawn from an exponential distribution were used. The optimal target solution was always found.

| $L$ | TGR | SGR | SGF | TGF | TPR | TPF | SPR | SPF |
|---|---|---|---|---|---|---|---|---|
| 1 | 14.4 | 17.5 | 17.4 | 15.0 | 14.7 | 14.4 | 15.5 | 14.6 |
| 3 | 3.4 | 17.6 | 18.7 | 15.2 | 15.2 | 14.0 | 16.1 | 16.2 |
| 5 | 2.8 | 8.2 | 16.8 | 12.8 | 14.3 | 14.6 | 15.9 | 15.8 |
| 7 | 2.9 | 4.0 | 11.7 | 13.0 | 13.3 | 13.7 | 15.2 | 16.0 |
| 9 | 3.0 | 4.0 | 7.4 | 9.5 | 14.0 | 13.8 | 17.0 | 16.4 |
| 11 | 2.9 | 4.3 | 5.4 | 8.1 | 14.8 | 14.0 | 17.3 | 16.0 |
| 15 | 2.9 | 4.1 | 4.6 | 5.8 | 15.5 | 14.4 | 14.9 | 15.9 |
| 19 | 3.4 | 4.2 | 4.7 | 7.9 | 19.1 | 14.9 | 16.3 | 16.0 |
| 23 | 3.5 | 4.0 | 3.8 | 6.8 | 31.1 | 15.6 | 16.6 | 16.2 |
| 27 | 3.6 | 4.2 | 4.2 | 6.5 | 36.1 | 18.7 | 17.2 | 15.9 |
| 31 | 4.0 | 4.4 | 4.2 | 9.0 | 35.5 | 25.2 | 16.7 | 16.0 |
| 39 | 5.0 | 4.7 | 5.0 | 10.6 | 34.9 | 33.2 | 15.5 | 16.3 |

all problems. List lengths ranging from 1 to 39 were used. The results are summarized in Table 3. We note that solution times for all greedy algorithms improved significantly as $L$ increased from one, while the solution times of the probabilistic algorithms improved only slightly if at all. In fact, the performance of SPR and SPF was found to be almost independent of $L$, and the performance of TPF and TPR deteriorated significantly as $L$ increased beyond 15. Table 3 also reveals that it is always possible to choose a value of $L$ for which a greedy algorithm performs significantly faster than its corresponding probabilistic algorithm. It is interesting to note that TGR performed better than TGF for all values of $L$. This was also true when runtimes were used as a performance measure. More research is required to explain this phenomenon. However, one possible explanation is that the random application of a shorter tabu list is sufficient to avoid cycling while at the same time the (occasional) exploration of promising, but otherwise tabu, neighborhoods is permitted.

We then extended the analysis to problems with different numbers of tasks, but with the same number of tasks per processors. Again, a large number of problems were generated and solved using different algorithms and list lengths, and the resulting data sets analyzed in the same manner as in the previous section. A typical result for algorithm SGF with from 3,000 to 9,000 processors is shown in figure 4. We note that observed solution times exhibit more variability as the size of the problem increases.

The effort required to solve a problem with a fixed number of tasks may depend on the number of processors available to perform these tasks. We ran a number of tests to determine if different values of $L^*$ should be used to solve problems with different numbers of processors. Typical results for algorithm SGR is shown in figure 5. Each point in this figure shows the average of 20 solution times for problems with 9,000 tasks and 150, 300 or 450 processors. We see that the optimal value of $L$ remains unchanged at about 9, but

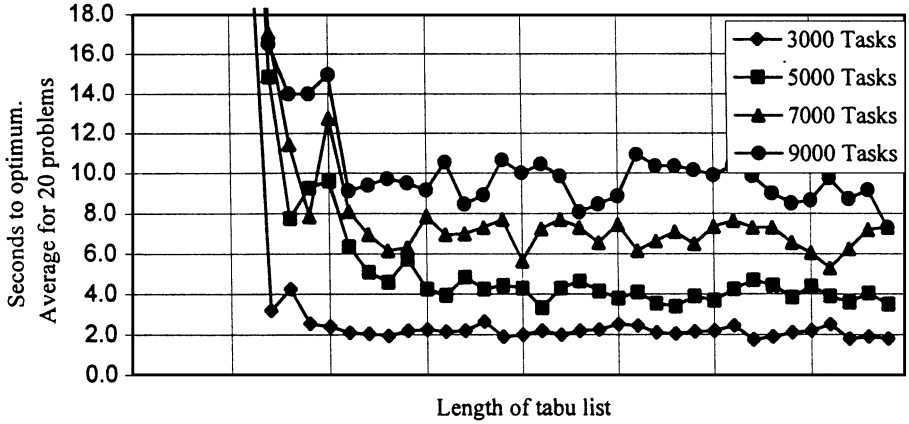


*Figure 4.* Effect of length of the tabu list on the average time required by algorithm SGF to solve problems with 20 tasks per processor. Random, integer processing times drawn from an exponential distribution were used. The optimal target solution was always found.

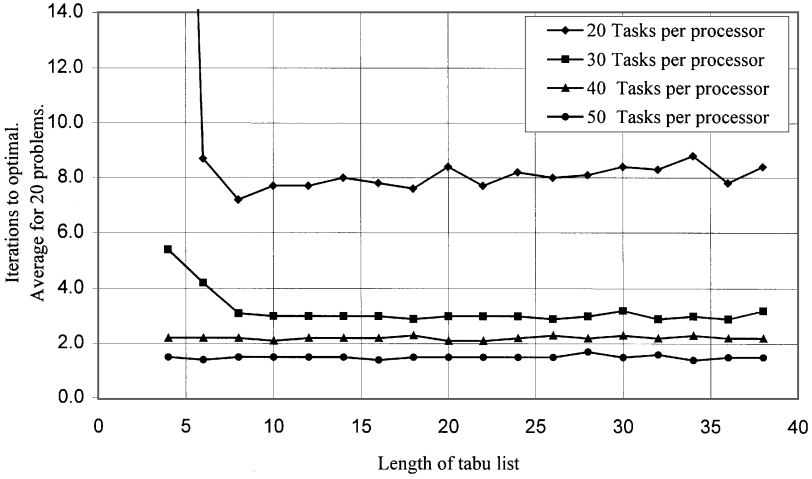


*Figure 5.* Effect of length of the tabu list on the average time required by algorithm SGR to solve problems with 9,000 tasks and 20 to 50 tasks per processor.

that the range of good values around the optimal value appears to decrease very slightly as the difficulty of the problem increases.

Graphs such as those shown in figures 4 and 5 were developed for all algorithms. Based on an analysis of these graphs, it was concluded that the value of $L^*$ for a given algorithm did not depend on the size and complexity of the problem being solved. However, the resulting values of $L^*$ differed significantly between different algorithms. Specific values are listed later in this section.

***Least squares approximation of expected solution times.*** Table 3 suggests the existence of a well-defined relationship between expected solution times ($t$), and list length ($L$) for all algorithms. If such a relationship can be captured in analytic form, then it should be possible to solve both for the optimal list length $L^*$ and the corresponding optimal expected solution time $t^*$. In addition, we should be able to find range ($R_p$) of values for $L(L_{\text{lo},p}$–$L_{\text{hi},p})$ for which $t$ is not more than $p$ percent greater than $t^*$.

To estimate the relationship between $t$ and $L$, we analyzed data sets from a large number of experiments similar to the ones illustrated in figures 4 and 5. The data sets were first sanitized by removing the iteration counts that were recorded when runs were truncated before the optimal solution was found. This typically occurred when an algorithm was run with extremely short list lengths. A least-squares fit was then made of the remaining data to the following function:

$$\tau = f(L, A, a_A, b_A, c, d, e, n, w) = \frac{a_A}{(L - b_A)^* \ln(c \times L)} + d \times L + e \tag{5}$$

Where

$\tau$ = estimate of expected solution time,
$L$ = list length,
$A$ = algorithm being used,
$a_A$ = constant to be estimated, assumed to depend on $A$ only,
$b_A$ = constant to be estimated, assumed to depend on $A$ only,
$c$ = constant to be estimated,
$d$ = constant to be estimated,
$e$ = constant to be estimated,
$n$ = number of tasks for this problem,
$w$ = number of processors for this problem.

Using Microsoft Excel's Solver routine, we obtained the coefficient values that minimized the sums of the squares of errors between observed and estimated solution times. Promising fits were found for several data sets. However, only for algorithm TGR did the data have sufficiently low variability to yield useful results. A typical result is shown in figure 6.

Excel was also used to find $L^*$ and $R_p$ for problems of different sizes (Table 4). We note that $L^*$ does not appear to be affected by problem size. Table 4 also gives the range of values of $L$ that leads to an expected solution time within 5% of the optimal. We note

*Table 4.* Estimated parameters and optimal lists lengths for problems with 20 tasks per processor.

| Algorithm | Tasks | Processors | $a$ | $b$ | $c$ | $d$ | $e$ | msq error | $L_{\text{lo}}$ | $L^*$ | $L_{\text{hi}}$ | $t^*$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| TGR | 3,000 | 250 | 18.4 | 1.87 | 221661.0 | 0.04 | 0.63 | 0.03 | 5.4 | 7.5 | 11.1 | 1.1 |
| TGR | 5,000 | 250 | 18.4 | 1.87 | 3798.6 | 0.05 | 1.40 | 0.04 | 5.5 | 7.9 | 12.4 | 2.1 |
| TGR | 7,000 | 250 | 18.4 | 1.87 | 74.9 | 0.10 | 3.29 | 0.52 | 4.5 | 7.6 | 12.5 | 4.5 |
| TGR | 9,000 | 250 | 18.4 | 1.87 | 26.6 | 0.17 | 4.77 | 0.22 | 4.5 | 6.7 | 10.8 | 6.6 |

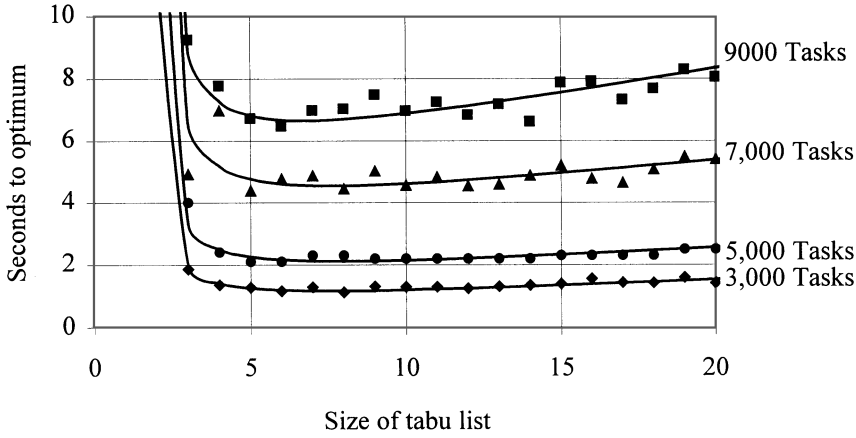


*Figure 6.* Effect of the length of the tabu list on the average time algorithm TGR needs to solve problems with 20 tasks per processor. Each point is the average for 50 different problems. Curves are least-squares estimates using Eq. (5). All runs were terminated after 20,000 nonimproving moves.

that this range fluctuates with problem size, but no trend is observed. Similar results were obtained for the other algorithms.

### *3.3. Recommendations*

Based on the analysis described above, recommended values for $L$ were established for each algorithm. These are listed in Table 5. Note that the recommended values of $L$ are independent of problem size and complexity. Algorithms using solutions as tabus were less sensitive to the choice of $L$ than were those using tasks as tabus. The performance of algorithms SPR and SPF did not depend on the value of $L$. For the sake of simplicity, a value of $L$ equal to zero is recommended for these two algorithms.

The eight recommended algorithms were used to solve 1,000 identical, randomly generated problems with 500 to 9,000 tasks and 25 to 450 processors. A summary of the

*Table 5.* Recommended values of $L$ for different algorithms.

| Algorithm | $L^*$ | $L_{lo}$–$L_{hi}$ | Remarks |
|---|---|---|---|
| SGF | 15 | 14–40 | Performs well with long lists. |
| SGR | 10 | 7–40 | Performs well with long lists. |
| SPF | 0 | 0 | SPF and SPR are identical when $L = 0$. |
| SPR | | | Slow, but optimal solution was always found. |
| TGF | 17 | ? | Slow, highly variable solution times. |
| TGR | 9 | 6–11 | Best performer. |
| | | | Predictable solution times. |
| | | | Performs poorly with shorter lists. |
| TPF | 7 | 1–15 | Slow, but optimal solution was always found. |
| TPR | 7 | 1–25 | Slow, but optimal solution was always found. |

*Table 6.* Mean and standard deviations of solution times and iteration counts for optimized algorithms solving problems with 9,000 tasks and 450 processors.

| | Iterations to optimum | | Seconds to optimum | |
|---|---|---|---|---|
| Algorithm | avg. | stdv. | avg. | stdv. |
| TGR | 2501 | 151.3 | 7.3 | 1.8 |
| SGR | 3150 | 584.2 | 10.1 | 2.5 |
| SGF | 3618 | 1101.8 | 11.1 | 3.5 |
| TGF | 15964 | 7724.1 | 25.5 | 12.0 |
| TPF | 5626 | 361.0 | 27.5 | 6.1 |
| TPR | 5447 | 396.3 | 29.1 | 6.2 |
| SPF | 4746 | 263.9 | 29.3 | 5.8 |
| SPR | 4745 | 275.9 | 29.7 | 6.2 |

resulting performance data for the case with 500 processors and 9,000 tasks is given in Table 6. Observe that algorithm TGR performed significantly faster and with significantly less variability than all other algorithms. Similar results were observed for all other combinations of tasks and processors. We conclude that algorithm TGR with a list length equal to 9 is the best of the algorithms evaluated here.

## 4. Evaluation

In Section 3 we used test problems for which an optimal solution could always be found. This was done to simplify the process of fine-tuning sub-algorithms and identifying the best combination of features. Here we will evaluate the performance of the winning algorithm (TGR) when confronted with more complex problems. Since the value of the optimal solution for these problems may not be equal to the ideal schedule duration ($T^*$), we will adopt a performance measure first used by Hübscher and Glover (1994). Specifically, the quality of the resulting solution will be measured as $\Delta T = (T - T^*)/T$. First, we will explore how the effort needed to solve a problem is affected by the precision of task durations. Then we will present a benchmark comparison between the effort expended by algorithm TGR to solve random problems with durations in the range 0.0–1.0 and similar results reported by Hübscher and Glover (algorithm HG). We will see that algorithm TGR deals efficiently with more complex problems and that it terminates in fewer iterations than algorithm HG while reaching solutions of similar or higher quality. This performance advantage is particularly apparent for larger test problems.

### *4.1. Problem complexity*

The number of potentially different solutions to a multiprocessor scheduling problem depends in part on the precision used to express task duration. A study was conducted to determine how well the algorithm would perform when solving problems where task durations are expressed with a higher degree of precision than the three to six digits used in the

*Table 7.* The effect of task-time precision on solution effort and quality when solving problems with 50 processors and 2,000 tasks with random durations drawn from an (integer) uniform distribution. Ten replications are used. Runs are terminated after 20,000 nonimproving moves.

| Precision (Max digits in task duration) | Average number of iterations | Average $\Delta T \times 10^8$ |
|---|---|---|
| 2 | 1 | 0 |
| 3 | 27 | 0 |
| 4 | 314 | 0 |
| 5 | 2,649 | 0 |
| 6 | 24,943 | 1.0 |
| 7 | 75,115 | 1.9 |
| 8 | 103,110 | 1.6 |
| 9 | 135,043 | 1.2 |
| 10 | 106,285 | 1.3 |
| 11 | 79,027 | 0.7 |
| Double precision | 103,291 | 1.1 |

previous section. Typical results are shown in Table 7 for problems with 50 processors and 2,000 tasks with durations drawn from uniform distributions with a range from $1–10^2$ to $1–10^{11}$. For each class of problems we report the average iteration count for 10 replications, as well as the observed average values for $\Delta T$. We note that optimal solutions were always reached when a precision of $10^5$ or less was used, and that a precision of $1.9 \times 10^{-8}$ or less was obtained when a higher level of precision was used. We also note that the number of iterations needed to reach the final solution increased substantially as the precision of task durations is increased up to nine digits, and that further increases in precision did not lead to further increase in the iteration counts or to higher quality solutions. Spot checks verified the hypothesis that improved solutions would be attained for problems using high precision durations if a stopping rule such as 100,000 nonimproving solutions were used. Needless to say, this improvement will be at the cost of a significant increase in iteration counts. Almost identical results were obtained when exponentially distributed task durations were used.

### *4.2. Pseudo-cycles*

Inspections of solution paths revealed the frequent presence of "pseudo-cycles". In this context, we define a pseudo-cycle as a sequence of solutions where a few tasks cycle between a small number of processors, while, at the same time one or two other task assignments are permanently changed. Since all solutions in a pseudo-cycle are unique, such cycles are not detected when solutions are used. However, our data suggests that pseudo-cycles are effectively broken when an accessible tabu list of random length is used. As suggested in Section 2, information accumulated in long-term memory may also be exploited to discourage the frequent moves associated with pseudo-cycles. To explore this

*Table 8.* Percent change in performance for using Eq. (3) in place of Eq. (1) to evaluate moves. Changes in bold are statistically significant at the 5% level.

| | SGF | SGR | SPF | SPR | TGF | TGR | TPF | TPR |
|---|---|---|---|---|---|---|---|---|
| Iterations | **− 21.5** | − 7.5 | − 4.6 | **− 5.2** | **− 70.1** | **− 2.2** | − 6.4 | **− 3.7** |
| Solution time | **15.3** | **31.7** | 1.0 | 1.7 | **− 42.7** | **35.6** | 2.5 | − 1.7 |

possibility, we modified the evaluation function used in the local search (Eq. (1)) to include the penalty for frequently moved tasks suggested in Eq. (3). The changes in performance due to this modification for problems with 9,000 tasks and 450 processors are shown in Table 8.

The most significant change in solution time (−42.7%) is observed for algorithm TGF. However, the resulting average solution time (14.6 s) is still the slowest time for all greedy algorithms. Although no other improvements in solution times were observed, it is interesting to note that all iteration counts were significantly reduced. Given the significant overhead in execution times, and the resulting minor reduction in iteration count, we do not recommend this approach for the problems studied here. Further research may lead to implementations that better exploit this reduction in iteration counts.

### *4.3. Benchmarking*

Hübscher and Glover (1994) give performance data for an algorithm solving 110 randomly generated problems similar to the ones solved here. Ten problems were generated and solved for eleven different combinations of processors and servers. Problem difficulty ranged from relatively easy (2 processors and 20 tasks) to somewhat difficult (50 processors and 2,000 tasks). All task durations were drawn from a uniform distribution with a range from zero to one. For each set of 10 problems the average number of iterations needed to reach the final solution was reported, as was the average range and average value of the performance measure $\Delta t$. Their results and the corresponding values for algorithm TGR solving similar problems are listed in Table 9.

It is seen that algorithm TGR in all but one case used fewer iterations to reach the final solution. In fact, algorithm TGR needed on the average 34,786 iterations to reach the final solution while, on the average 78,082 iterations were needed for algorithm HG. It should be noted that the test problems solved by the two algorithms were different, although they were generated in the same way. Hence, some of these differences are likely to be due to differences in the two randomly generated problem sets. However, a $t$-test confirmed on the 95% level, the hypothesis that the mean iteration counts were drawn from different populations.

The data in Table 9 also shows that this improvement in efficiency is accompanied by a corresponding improvement in solution quality. The average value of $\Delta T$ for the solutions obtained by algorithm TGR was $1.4 \times 10^{-8}$ while it was $2.1 \times 10^{-8}$ for the solutions obtained by algorithm HG. In no case did algorithm HG yield a solution with a higher quality than the one obtained by algorithm TGR. Again, it should be noted that the two

*Table 9.* Performance of algorithm TGR when solving random problems similar to those solved by Hübscher and Glover. Averages are based on 10 replications. Double precision is used. Runs were terminated after 20,000 nonproductive moves. H&G results are from Hübscher and Glover (1994).

| No. of processors | No. of tasks | Average no. of iterations to final solution | | Average $\Delta T \times 10^8$ | |
|---|---|---|---|---|---|
| | | H&G | TGR | H&G | TGR |
| 2 | 50 | 17,607 | 10,232 | 1.1 | 0.7 |
| 2 | 100 | 19,157 | 18,888 | 0.1 | 0.1 |
| 3 | 100 | 16,384 | 21,248 | 0.8 | 0.4 |
| 3 | 200 | 39,293 | 17,814 | 0.1 | 0.1 |
| 5 | 100 | 36,392 | 21,979 | 4.4 | 3.3 |
| 5 | 200 | 51,982 | 20,367 | 1.4 | 0.3 |
| 10 | 200 | 101,801 | 37,556 | 5.0 | 4.8 |
| 10 | 500 | 84,816 | 32,740 | 1.1 | 0.3 |
| 20 | 500 | 153,526 | 46,149 | 3.2 | 3.2 |
| 20 | 1,000 | 124,540 | 52,381 | 1.8 | 0.5 |
| 50 | 2,000 | 224,408 | 103,291 | 4.5 | 1.1 |

algorithms solved different test problems, hence, some of this difference is likely to be due to differences in the two randomly generated problem sets.

The performance differences between the two algorithms were most significant for larger problems. For example, for problems with 10 or more bins and 200 and more tasks, TGR needed an average of 54,423 iterations to reach the final solution while HG needed an average of 137,818 iterations. The corresponding average solution qualities were $\Delta T = 3.1 \times 10^{-8}$ for HG and $\Delta T = 2.0 \times 10^{-8}$ for TRG. Hence, for more difficult problems, a 61% reduction in average iteration count was accompanied by a 37% improvement in solution quality. This is significant as it suggests that the effort needed to solve problems of increasing size increases at a slower rate for algorithm TGR.

## 5. Conclusion

In this paper we introduced and evaluated eight different approaches to the design of a tabu search algorithm for solving a multiprocessor scheduling problem. This algorithm differs from many similar algorithms in their simplicity, and in their sparse use of heuristic "weights" and "scale factors". Among the design options used to implement these algorithms were probabilistic and local search algorithms, the use of solutions and tasks as tabus, and the use of accessible tabu lists of fixed and random length. It was found that a probabilistic local search algorithm never performed as fast as a greedy search algorithm with appropriate parameters. On the other hand, the performance of the probabilistic search was relatively independent of the length of the tabu list. This suggests that a probabilistic search without any tabu list at all may be the algorithm of choice for one-of-a-kind problems where the time and expense of developing, debugging and fine-tuning a tabu list scheme may not be cost effective. Also, there may be ways of improving run times by using more advanced software design techniques that would make this approach more effective.

The use of accessible tabu lists of random length improved the performance of all algorithms. This improvement was particularly significant for algorithms using tasks as tabus. Hash coding without conflict resolution proved to be an effective way to manage solutions on the tabu list. The efficiency of this approach was evidenced by a barely noticeable increase in solution times as the length of the list increased. Long tabu lists were effective when solutions were used. Although hash-coded solutions were effective list entries, the use of recently moved tasks proved to be even more effective when an accessible list of random length was used. This increase in efficiency can at least in part be attributed to the fact that no list management was needed when tasks were used.

All algorithms were sensitive to the choice of the list length parameter. Too short a list resulted in exceptionally poor performance for algorithms using a greedy local search. Longer lists quickly degraded performance when tasks were tabus. Performance was degraded much more slowly when solutions were used. In all cases, it would be better to use a list length longer than the optimal one than one shorter than the optimal one. Large and complex problems were more sensitive to the choice of list length than were smaller, simpler problems.

## References


Anderson, Charles A., Kathryn Fraughnaugh, Mark Parker, and Jennifer Ryan. (1993). "Path Assignment for Call Routing: An Application of Tabu Search," *Annals of Operations Research* 41, 301–312.

Barnes, J.W. and M. Laguna. (1993). "A Tabu Search Experience in Production Scheduling," *Annals of Operations Research* 41, 141–156.

Barnes, J.W., M. Laguna, and F. Glover. (1995). "An Overview of Tabu Search Approaches to Production Scheduling Problems." In D.E. Brown and W.T. Scherer (eds.), *Intelligent Scheduling System*. Kluwer Academic Publishers, pp. 101–127.

Blazewicz, J., K.E. Ecker E. Pesch, G. Schmidt, and J. Weglarz. (1995). *Scheduling Computer and Manufacturing Processes*. Berlin: Springer-Verlag.

Glover, F. (1986). "Future Paths for Integer Programming and Links to Artificial Intelligence," *Computers and Operations Research* 13, 533–549.

Glover, F. (1995). "Tabu Search Fundamentals and Uses, Revised and Expanded: April 1995." Technical Report, Graduate School of Business, University of Colorado, Bolder, CO.

Glover, F. and M. Laguna. (1997). *Tabu Search*. Boston: Kluwer Academic Publishers.

Graham, R.L. (1969). "Bounds for Certain Multiprocessing Anomalities," *SIAM J. Appl. Math*. 17, 263–269.

Graham, R.L. (1976). "Bounds on Performance of Scheduling Algorithms." In E.G. Coffman, Jr. (ed.), *Scheduling in Computer and Job Shop Systems*. Chapt. 5, New York: John Wiley.

Johnson, D.S., A Demers, J.D. Ullman, M.R. Garey, and R.L. Graham. (1974). "Worst Case Performance Bounds for Simple One-Dimensional Packing Algorithms," *SIAM Journal on Computing* 3, 299–326.

Hübscher, Roland and F. Glover. (1994). "Applying Tabu Search with Influential Diversification to Multiprocessor Scheduling," *Computers in Operations Research* 21, 877–844.

Kämpke, Thomas. (1988). "Simulated Annealing: Use of a New Tool in Bin Packing," *Annals of Operations Research* 16, 327–332.

Knuth, D. (1973). *The Art of Computer Programming: Sorting and Searching*. Reading, MA: Addison-Wesley.

Laguna, M. and F. Glover. (1993). "Integrating Target Analysis and Tabu Search for Improved Scheduling Systems," *Expert Systems With Applications* 6, 287–297.

Løkketangen, A. (1995). "Tabu Search as a Metaheuristic Guide for Combinatorial Optimization Problems," Dr. Scient. Thesis, Institute for Informatics, University of Bergen, Bergen, Norway.

Morton, T.E. and David W. Pentico. (1993). *Heuristic Scheduling Systems*. New York, NY: John Wiley & Sons.

Osman, I. and J.P. Kelly (eds.). (1996). *Meta-Heuristics. Theory and Applications*. Boston: Kluwer Academic Publishers.

Taillard, Eric D. (1994). "Parallel Taboo Search Techniques for the Job Shop Scheduling Problem," *ORSA Journal on Computing* 6(2), 108–117.