EINDHOVEN UNIVERSITY OF TECHNOLOGY
Department of Mathematics and Computing Science

MASTER'S THESIS

A parallel local search algorithm for the job shop scheduling problem

by

B.J.M. Aarts

Supervisor	: Prof. Dr. E.H.L. Aarts
Advisors	: Dr. Ir. M.G.A. Verhoeven
	: Dr. Ir. H.M.M. Ten Eikelder

April 1996

Acknowledgements

I thank Prof. Dr. E.H.L. Aarts for his supervision and for his efforts to get me graduated. I am particularly indebted to my advisors Huub Ten Eikelder and Marco Verhoeven, for their enthusiastic guidance and support. Furthermore I thank Rob Vaessens, all my friends and fellow students and my parents for their help and support, and I thank Vincent Huijgen for getting me started, and moving, with PVM. I also thank Marco Verhoeven and Wim Nuijten for providing me with their NV styles for LaTeX. Special thanks to Nicole... for everything.

Bas J.M. Aarts
Eindhoven, April 16, 1996

Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

In a *combinatorial optimization problem* the objective is to find an optimal solution among a finite, often very large, number of alternatives [Papadimitriou & Steiglitz, 1982]. Many of these problems are $\mathcal{NP}$-hard, with the direct consequence that optimal solutions can not be obtained within reasonable amounts of computation time for larger instances. A way to cope with these hard problems is the use of so called *local search* algorithms. Extensive performance analysis has revealed that these algorithms can find good quality solutions in acceptable running times for a vast majority of optimization problems. For large instances of $\mathcal{NP}$-hard problems, however, the running time can still be considerable. One way of reducing these running times is by using parallelism. This thesis is about the design of a parallel local search algorithm for the *job shop scheduling problem*.

The organization of this thesis is as follows. The next two sections present a brief summary of the most important aspects of local search in general and of *tabu search* —a variant of local search— in particular. The last section of this chapter defines the job shop scheduling problem and in Chapter 2 the application of local search to this problem is discussed. Chapter 3 discusses the design of a sequential local search algorithm and Chapter 4 imposes some optimizations to this algorithm. Chapter 5 first gives an insight of parallel local search algorithms, followed by the design of two parallel versions of the sequential algorithm of Chapter 4. The results of the different algorithms are presented in Chapter 6.

1.1 Local search

Local search is based on what is perhaps the oldest optimization method known, trial and error. In fact it is so simple that it is surprising just how well it works on a variety of difficult optimization problems. The basic idea behind local search is to inspect the solutions that are *close* to a given solution, the so called *neighbours*, and to restart when a better solution —with respect to a predefined cost function— is obtained. A *pivoting rule* [Yannakakis, 1990] determines which neighbour will be the new current solution. Well known pivoting rules are *first improvement*, which replaces the current solution with the first lower cost neighbour that is found, and *best improvement*, which uses the solution that has lowest cost among all neighbours. Local search presupposes the specification of an instance of a combinatorial optimization problem and a *neighbourhood structure*.

Definition 1.1. An instance of a combinatorial optimization problem is represented by a pair $(\mathcal{S}, f)$, in which the *solution space* $\mathcal{S}$ is the set of *feasible* solutions, and the cost function $f : \mathcal{S} \rightarrow \mathbb{R}$ gives the cost of a solution $s \in \mathcal{S}$. The objective is to find a solution with minimal cost. When for a solution s holds that $(\forall s' : s' \in \mathcal{S} : f(s') \geq f(s))$, i.e. there is no better solution, s is called a *global minimum*. □

proc *Iterative_Improvement* (**var** $s : \mathcal{S}$)
{ precondition: $s \in \mathcal{S}$;
postcondition: s is a local minimum with respect to $\mathcal{N}$
}
|[**var** $W : \mathcal{P}(\mathcal{S})$; $s' : \mathcal{S}$;
$W := \varnothing$;
{invariant P: $(\forall w : w \in W : f(w) \geq f(s))$}
do $\mathcal{N}(s) \setminus W \neq \varnothing \rightarrow$
$s' :\in \mathcal{N}(s) \setminus W$;
if $f(s') \geq f(s) \rightarrow W := W \cup \{s'\}$
$\square\, f(s') < f(s) \rightarrow s,\ W := s',\ \varnothing$
fi
{P}
od
{$\mathrm{P} \wedge \mathcal{N}(s) \setminus W = \varnothing \Rightarrow s$ is a local minimum with respect to $\mathcal{N}$}
]|

Figure 1.1: An iterative improvement algorithm.

A solution being close to another solution is formalized by a neighbourhood structure and an *exchange function*.

Definition 1.2. Let $(\mathcal{S}, f)$ be an instance of a combinatorial optimization problem. A neighbourhood structure $\mathcal{N} : \mathcal{S} \rightarrow \mathcal{P}(\mathcal{S})$ assigns to each solution $s \in \mathcal{S}$ a set of solutions, called the neighbourhood. A solution $s \in \mathcal{S}$ is a *local minimum* with respect to $\mathcal{N}$ if $(\forall s' : s' \in \mathcal{N}(s) : f(s') \geq f(s))$. $\square$

The neighbours of a solution are not given explicitly, but are constructed by an exchange function. When assumed that solutions are represented by sets, most neighbourhoods are based on the replacement of a few elements that constitute a solution.

Definition 1.3. Let the set of elements constituting solutions in $\mathcal{S}$ be given by $\mathcal{E} = \{e \mid e \in s \wedge s \in \mathcal{S}\}$, and let the partial function $\tau : \mathcal{S} \times \mathcal{E}^l \rightarrow \mathcal{S}$ be the exchange function with $l \in \mathbb{N}$. Then, an l-exchange neighbourhood structure $\mathcal{N}_l$ is defined by $\mathcal{N}_l(s) = \{\tau(s, e_1, \ldots, e_l) \mid e_1, \ldots, e_l \in s\}$ for $s \in \mathcal{S}$. $\square$

The size of a neighbourhood grows significantly with the number of elements that is replaced. Therefore, the time complexity of a local search algorithm strongly depends on the exchange function that is used. In particular when the neighbourhood of a solution is equal to the entire solution space, a local search algorithm has the same time complexity as an exhaustive search algorithm, i.e. exponentially in the size of the instance.

Local search algorithms constitute a class of approximation algorithms that are based on repeatedly replacing a solution by a neighbouring solution. The presented imperative improvement algorithm of Figure 1.1 is a basic local search algorithm. The great drawback of this algorithm is its disability to escape from a local minimum, with the risk of a relative poor quality solution upon termination. There are many variations on this basic algorithm known in literature that do not have this drawback. Iterated local search [Johnson, 1990; Martin, Otto & Felten, 1991], tabu search [Glover, 1989; Glover, Taillard & De Werra, 1993], simulated annealing [Kirkpatrick, Gelatt & Vecchi, 1983; Aarts & Korst, 1989] and genetic local search [Goldberg, 1989; Mühlenbein, Gorges-Schleuter & Krämer, 1988] all have different mechanisms to escape from local minima. For a more detailed overview of these variants see [Yannakakis, 1990; Reeves, 1993; Aarts & Lenstra, 1996].

A widely recognized advantage of local search is its general applicability. Only a specification of solutions, a cost function, and a neighbourhood structure are required, which can easily be defined for many problems. An unresolved issue concerning local search is the complexity to find a local optimum. Consequently it is not possible to give a non-trivial upper bound for the number of iterations needed to find a local minimum.

The local search variant that is used for the design of the parallel local search algorithms is tabu search, because of its eagerness and its simplicity to escape from a local minimum. The next section will provide some basic information about tabu search.

1.2 Tabu search

Tabu search was first presented by Glover [1989], and its present form has become an established optimization approach. The general thought behind tabu search is quite simple and intuitively very plausible. Let $(\mathcal{S}, f)$ be an instance of a combinatorial optimization problem, and let $s \in \mathcal{S}$ be the current solution. As stated before, the current solution is usually replaced by a solution with better cost. When, however, s is a local minimum, there are no such neighbours and while the basic iterative improvement algorithm from Figure 1.1 terminates at this point, tabu search continues its search for better solutions. In this case s is replaced by a *certain* neighbour s' from its neighbourhood. It is possible, however, that s was obtained by an exchange on s'. Therefore, replacing s by s' is not desirable, since s' was already inspected in the previous iteration. It is even possible —in fact it's almost always the case— that after a period of time a number of solutions is successively replaced by each other until termination, with the direct consequence that no better solutions are found.

To avoid this *cycling* through a collection of solutions, tabu search keeps track of the last T ($T \in \mathbb{N}^+$) solutions, called *forbidden* solutions, that were obtained during the search. When a solution s is replaced by a *permitted* solution s', s becomes a forbidden solution for the next T replacements. This will prevent cycles of length less than or equal to T. Making a solution forbidden can be realised by placing it in a FIFO-buffer tl —from now on referred to as a *tabu list*— that keeps record of the $|tl|$ solutions most recently visited, with a maximum of T. T is the maximum length of the tabu list, called the *tabu tenure*. This formulation normally has only a theoretical interest, since keeping track of the $|T|$ most recently obtained solutions is often extremely space consuming. In practice only the difference(s) between two successive solutions, i.e. the exchanges, are added to the tabu list. To formalize this, the notion *move* is introduced.

Definition 1.4. Let s be a solution of a combinatorial optimization problem $(\mathcal{S}, f)$. The set $M(s)$ is defined to be the set of *legal* moves m that can be applied to s in order to obtain a neighbouring solution $s' \stackrel{\text{def}}{=} s \odot m$. □

Remark 1.1. Note that this definition is in fact superfluous, since it is an abbreviation of Definition 1.3. □

When a solution s is replaced by a solution s', the move m for which $s' \odot m = s$ has to be added to the tabu list. Therefore, it is useful to employ a set of moves that is *uniquely reversible*, i.e.

$$(\forall s : s \in \mathcal{S} : (\forall m : m \in M(s) : (\exists! m' : m' \in M(s) : (s \odot m) \odot m' = s))).$$

For a move m its reversible is denoted by m^{-1}.

As a stop criterion for tabu search, usually an upper bound for the number of moves is used. In Figure 1.2 a basic tabu search algorithm is presented. In Chapter 4 some modifications are made to this version in order to obtain better quality solutions.

```
proc Tabu_Search(var s : 𝒮)
{ precondition: s ∈ 𝒮;
  postcondition: s is the best solution found in MaxIt iterations
}
|[con MaxIt, T : int; {MaxIt ≥ 0 ∧ T > 0}
  var s_opt : 𝒮; tl :list of ℳ; It : int; m : ℳ;
  {The type of a single move is denoted by ℳ}
  s_opt, tl, It := s, [ ], 0;
  do It < MaxIt →
     m := Propose(s, M(s), tl);
     tl := tl ⊕_T [m^{-1}];
     s := s ⊙ m;
     if f(s) < f(s_opt) → s_opt := s
     □ f(s) ≥ f(s_opt) → skip
     fi;
     It := It + 1
  od;
  s := s_opt
  {s is the best solution found}
]|
```

Figure 1.2: A basic tabu search algorithm.

The function *Propose*$(s,\ M(s),\ tl)$ returns a move based on some predefined relation given the current solution s, the set of legal moves regarding that solution $M(s)$, and the tabu list tl. On account of the description of tabu search given earlier, one would expect this function to look like "$m :\in M(s) \setminus tl$". Sometimes, however, it is desirable to choose a move that is on the tabu list. For example when that move results in a solution with better cost than found so far, or when all the neighbours of that solution are *tabu*. The decision to choose a forbidden move over a permitted one is controlled by a so called *aspiration level*. The second parameter of *Propose* is redundant at this time since it can be derived from s, but it is used in the final version of the algorithm. The operator $\oplus_T$ appends a move to the end of the tabu list, making sure that its length does not exceed T, i.e.

$$tl \oplus_T [m] \stackrel{\text{def}}{=} (\textbf{if } |tl| = T \rightarrow \text{tail}(tl) \ \Box\ |tl| < T \rightarrow tl \ \textbf{fi}) +\!\!+ [m].$$

Given an instance of a combinatorial optimization problem, it is impossible to choose a tabu tenure for which with certainty an optimal solution can be found. Therefore, the choice for this tenure is often based more on intuition than on knowledge.

Remark 1.2. The expression $M(s) \setminus tl$ used before is syntacticly incorrect since $M(s)$ is a set and tl a list. Since its semantics is obvious, this notation is used through out this thesis. $\Box$

To apply this approach to the job shop scheduling problem, questions like *how* to construct an initial solution, *how* to mark a move as forbidden, *what* neighbourhood structure to use, and *which* way to search this neighbourhood have to be answered. Chapters 2 and 3 discuss these choices.

1.3 Job shop scheduling

Job shop scheduling is an important model in scheduling theory [Lawler, Lenstra, Rinnooy Kan & Shmoys, 1993], which serves as a proving ground for new algorithmic ideas and provides a starting point for the more complicated, practically relevant scheduling models. Furthermore, the problem is $\mathcal{NP}$-hard and local search has proven to belong to the best approximation algorithms for it [Vaessens, Aarts & Lenstra, 1994].

An instance of the job shop scheduling problem consists of a set of jobs and machines, of which each machine can handle at most one job at a time. Each job consists of a chain of operations which *must* be processed in that order during an uninterrupted interval of time, called the processing time. The problem is to find a schedule with minimal length. More formally the problem can be defined as follows.

Definition 1.5. Given are the three finite sets J of jobs, M of machines and O of operations. For each operation $a \in O$ there is a job $j(a) \in J$ to which it belongs, a machine $m(a) \in M$ on which it must be processed and a processing time $d(a) \in \mathbb{N}$. Furthermore, for each operation $a \in O$ its successor in the job is denoted by $sj_a \in O$, except for the last operation in a job, which has no successor. The problem is to find a start time $s(a) \geq 0$ for each $a \in O$ such that the cost function

$$(\mathbf{max}\ a : a \in O : s(a) + d(a))$$

is minimal, subject to

(1) $(\forall a : a, sj_a \in O : s(sj_a) \geq s(a) + d(a))$ and

(2) $(\forall a, b : a, b \in O \wedge a \neq b \wedge m(a) = m(b) : s(b) \geq s(a) + d(a) \vee s(a) \geq s(b) + d(b))$.

□

The cost function gives the length of a schedule. The constraints corresponding to (1) are job precedence constraints, implying that an operation may not be processed if its predecessor is not finished, and those corresponding to (2) are machine capacity constraints, implying that two operations may not be processed on the same machine at the same time. Successive operations of the same job are assumed to be processed on different machines.

In the next chapter a mathematical representation of a schedule is discussed, along with a neighbourhood structure and an exchange function to obtain these neighbours.

CHAPTER 2

Local search and job shop scheduling

In this chapter the application of local search to the job shop scheduling problem is discussed. The first section defines a mathematical representation for a schedule, and the second section deals with the choice of a suitable neighbourhood structure.

2.1 A graph representation

In order apply local search to the job shop scheduling problem, it is most appropriate to use the *disjunctive graph* representation of Roy & Sussmann [1964].

Definition 2.1. A disjunctive graph G is a graph with two kinds of edges: directed and undirected. More formally, a disjunctive graph is a three-tuple (V, A, E) in which V is a set of vertices, A is a set of directed edges, called arcs and E is a set of edges. A vertex weighted disjuncted graph, is a disjuncted graph for which each vertex $v \in V$ has a weight assigned to it. □

Definition 2.2. An instance of the job shop scheduling problem of Definition 1.5 is represented by a vertex weighted disjunctive graph $G = (V, A, E)$ with node set $V \stackrel{\text{def}}{=} O$, arc set $A \stackrel{\text{def}}{=} \{(a, sj_a) | a, sj_a \in O\}$, and edge set $E \stackrel{\text{def}}{=} \{\{a, b\} | a, b \in O \wedge m(a) = m(b)\}$. A weight $d(a)$ is associated with each vertex $a \in V$. □

The arcs represent the job precedence constraints; the edges represent the machine capacity constraints.

A feasible solution is represented by a subset E' of orientations of the edges in E, such that E' gives for each machine a linear ordering of the operations that have to be processed on it, and such that the resulting digraph $D = (V, A \cup E')$ is acyclic. The start time of an operation $a \in V$ is the length of a longest path up to and without a, and the cost $f(D)$ of a feasible solution is the longest path in D.

Example 2.1. Figure 2.1 gives an example and a solution of this representation for an instance with three machines and three jobs, each consisting of three operations, viz.

$$J = \{job_1, job_2, job_3\},\ M = \{m_1, m_2, m_3\} \text{ and } O = \{1, 2, 3, 4, 5, 6, 7, 8, 9\},$$
$$j(1) = j(2) = j(3) = job_1,\ j(4) = j(5) = j(6) = job_2 \text{ and } j(7) = j(8) = j(9) = job_3, \text{ and}$$
$$m(1) = m(4) = m(8) = m_1,\ m(2) = m(6) = m(7) = m_2 \text{ and } m(3) = m(5) = m(9) = m_3.$$

□

A digraph D corresponding with a feasible solution, contains only oriented edges between direct successors and predecessors on a machine, since the orientation of the edges in $E \setminus E'$ is uniquely determined. Therefore, each vertex has at most one successor an predecessor on its machine and in its

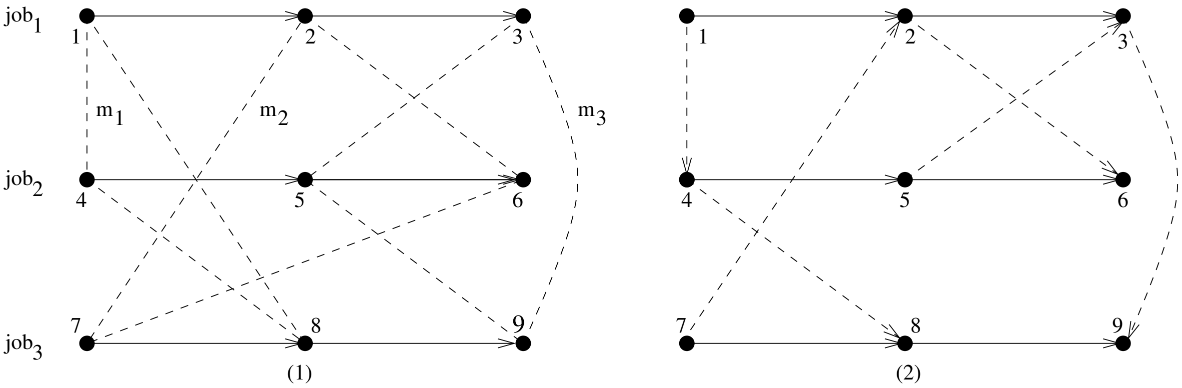

Figure 2.1: Example of an instance (1) and a digraph of a solution of this instance (2).

job. Given this orientation, a feasible start time of an operation a can be determined by setting it to the length of the longest path in D up to a. The cost of a solution presented by a digraph with N operations can be computed in $\mathcal{O}(N)$ time using Bellman's labeling algorithm [Lawler, 1976], since the degree of each vertex in D is at most four. The problem now is to find an orientation of the edges in E that minimizes the length of the longest path in D.

2.2 Neighbourhood structures

In this section some neighbourhoods that have been used in other algorithms are presented, along with their exchange function and properties. Based on these properties a choice for the neighbourhood to be used is made. First some additional notation is introduced, that is used through out the rest of this thesis.

Definition 2.3. Given are a digraph $D = (V, E)$ and a vertex $a \in V$. Its successor and predecessor on the machine and in the job of D are denoted by $sm_{a,D},\ pm_{a,D},\ sj_{a,D}$ and $pj_{a,D}$ respectively. When it is clear that these functions are applied to a digraph D, the subscript D is left out. An operation having no successor in its job or on its machine is denoted by $sj_a = \diamond_s$ or $sm_a = \diamond_s$ respectively. Completely analogous to this, the absence of a predecessor in a job or a machine is denoted by $pj_a = \diamond_p$ or $pm_a = \diamond_p$. Both $\diamond_s$ and $\diamond_p$ are *new* operations, i.e. $\diamond_s \notin V \wedge \diamond_p \notin V$, and they are not equal. □

Definition 2.4. The solution space for an instance of the job shop scheduling problem is the set of all acyclic digraphs that represent a solution of its disjunctive graph. It is denoted by $\mathcal{D}$. □

Most neighbourhoods for the job shop scheduling problem are based on reversing machine capacity arcs in the digraph, i.e. reversing the order in which operations are processed on a machine. Van Laarhoven, Aarts & Lenstra [1992] give the following 1-opt neighbourhood $\mathcal{N}_1$. Given a solution $s \in \mathcal{S}$, represented by a digraph D, a neighbouring solution is obtained by choosing two operations a and b, with $m(a) = m(b)$, that are adjacent and reversing arc (a, b) in D. An arc reversal is the move for this neighbourhood (see Definition 1.4), and the type of this move is an arc, from now on denoted by $\mathcal{A}$. Van Laarhoven, Aarts & Lenstra [1992] have proven that reversing an arc *not* on a longest path in D, can lead to an infeasible solution and that a reversal of an arc on a longest path always results in another feasible solution. Therefore, in order to obtain a feasible solution (i.e. an acyclic digraph), only arcs on a longest path are reversed.

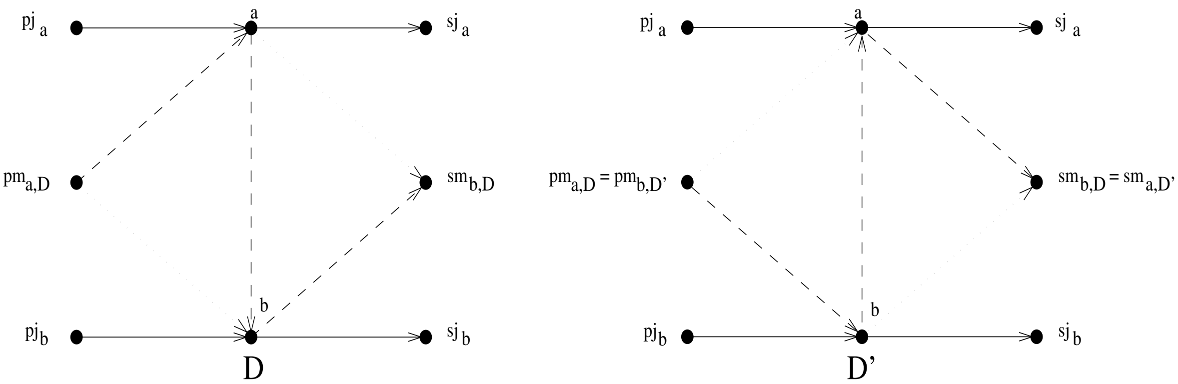

Figure 2.2: Part of a digraph D before and D' after the reversal of arc (a, b). Dotted arcs are redundant machine capacity arcs.

Definition 2.5. Let $D \in \mathcal{D}$, with $D = (V, E)$ and let $a, b \in V$ with $b = sm_a$, then the exchange function *rev* is defined as

$$rev(D, a, b) = (V, (E \setminus \{(a, b), (pm_a, a), (b, sm_b)\}) \cup \{(pm_a, b), (b, a), (a, sm_b)\}.$$

The 1-opt neighbourhood $\mathcal{N}_1$ is then defined as

$$\mathcal{N}_1(D) = \{rev(D, a, sm_a)|\ a, sm_a \in V \wedge (a, sm_a) \text{ on a longest path in } D\}.$$

□

For an instance with N operations, the size of this neighbourhood is $\mathcal{O}(N)$. Figure 2.2 gives an example of an arc reversal.

The neighbourhood of a solution D when using this neighbourhood structure, includes a (possible empty) set of solutions with cost at least $f(D)$. Since these neighbours are considered *not promising*, they are excluded from the neighbourhood. To this extend the notion of a *block* is defined.

Definition 2.6. A block is a maximal sequence of size at least one, consisting of adjacent operations that are processed on the same machine and belong to a longest path. An operation of a block is called *internal* if it is neither the first nor the last operation of that block. An *internal arc* is an arc between two internal operations of a block. The *size* of a block is the number of operations in that block. □

Matsuo, Suh & Sullivan [1988] proved that given a digraph $D \in \mathcal{D}$, a reversal of an internal arc results in a acyclic digraph D' with $f(D') \geq f(D)$. They use the neighbourhood function $\mathcal{N}_{1a}$ for their simulated annealing algorithm. It is defined as

$$\mathcal{N}_{1a}(D) = \{rev(D, a, sm_a)|\ a, sm_a \in V \wedge (a, sm_a) \text{ on a longest path in } D \wedge (a, sm_a) \text{ is not an internal arc}\}.$$

In practice the average difference between $\mathcal{N}_1$ and $\mathcal{N}_{1a}$ is very small, since blocks of size greater than three seldom occur. Nowicki & Smutnicki [1995] proved that given a digraph D, an arc reversal of an arc between the first operation of a first block and an internal operation results in a acyclic digraph D' with $f(D') \geq f(D)$. The same argument holds for the last operation of a last block. To formalize the neighbourhood structure they use, the meaning of an internal arc is redefined.

Redefinition 2.7. An internal arc is an arc between two internal operations, an arc between the first operation of a first block and an internal operation of that block, or an arc between an internal operation of a last block and the last operation of that block. □

The neighbourhood structure $\mathcal{N}_{1b}$ Nowicki & Smutnicki [1995] use for their tabu search algorithm is now defined as

$$\mathcal{N}_{1b}(D) = \{rev(D, a, sm_a) |\ a, sm_a \in V \wedge (a, sm_a) \text{ on a longest path in } D \wedge \\ (a, sm_a) \text{ is not an internal arc}\}.$$

Nowicki & Smutnicki proved a convenient corollary for $\mathcal{N}_{1b}$:

Corollary 2.1. *If for a solution $s \in \mathcal{S}$ with digraph D holds that $\mathcal{N}_{1b}(D) = \emptyset$ then s is a global optimum of $\mathcal{S}$.*
Proof. See Appendix B of [Nowicki & Smutnicki, 1995]. □

This corollary is especially useful as a termination criterion since in this case it is of no use to search any further. Nowicki & Smutnicki proved optimality for 20 out of the 40 Lawrence-instances([1984]), based on this corollary. In spite of this, it is fair to say that for most instances optimality can not be detected in this way.

Over the years many different neighbourhood structures have been applied to the job shop scheduling problem. Some of them are:

- $\mathcal{N}_4$, used by Balas & Vazacopoulos [1994]. This neighbourhood considers two operations a and b on the same machine such that a occurs prior to b. A neighbour is constructed by inserting a immediately after b, or inserting b immediately before a. Sufficient conditions are derived under which these new schedules are feasible.
- $\mathcal{N}_5$, used by Adams, Balas & Zawack [1988]. For a machine m with an operation on the longest path, a neighbour is obtained by replacing the orientation on m, by any other feasible orientation.
- $\mathcal{N}_7$, used by Applegate & Cook [1991]. This neighbourhood contains all feasible solutions that can be obtained by simultaneously replacing the orientation on $|M| - t$ machines, in which t is small number depending on the number of machines.

For a more complete and detailed discussion about neighbourhoods for the job shop scheduling problem, see [Vaessens, Aarts & Lenstra, 1994].

In every iteration of the algorithm the neighbourhood has to be searched according to some strategy. $\mathcal{N}_{1b}$ is a small neighbourhood, which will result in a very fast neighbourhood search. This and Corollary 2.1 make $\mathcal{N}_{1b}$ an attractive neighbourhood structure and therefore it is used for the design of the parallel tabu search algorithm. Note, however, that a small neighbourhood is not by definition the best possible one.

In the next chapter the neighbourhood search strategy is discussed, along with a more specific way how to mark a reversal as a forbidden reversal. Based on these decisions the first version of the sequential algorithm is presented.

CHAPTER 3

The sequential algorithm

This chapter discusses the sequential algorithm more specifically. At first the neighbourhood search strategy and the representation used for the tabu list are presented, and the final section presents the first version of the sequential algorithm.

3.1 The neighbourhood search strategy

In this section the neighbourhood search strategy is discussed. Given a solution, this strategy determines the neighbour for that solution that is used in the next iteration of the algorithm. The time complexity of this search depends on the size of the neighbourhood and of the complexity of the computation of the cost of a neighbour. Since the size of $\mathcal{N}_{1b}(D)$, for a digraph D, is rather small, the time complexity depends mainly on the latter. As stated before before, a solution is usually replaced by a neighbouring solution with the best cost. For the job shop scheduling problem, the computation of $f(D)$ for a digraph D has time complexity $\mathcal{O}(N)$, when D has N vertices. Nowicki & Smutnicki [1995] use this strategy for their tabu search algorithm. Taillard [1994] proved that, for a digraph D, it is possible to compute a lower bound for the cost of a neighbour of D in constant time, resulting in a significantly faster search for large instances. Before presenting the theorem that proves this, some extra definitions are needed.

Definition 3.1. Given a digraph $D = (V, E)$ and a vertex $a \in V$ the functions st, rt and l, all of type $\mathcal{V} \times \mathcal{D} \to \mathbb{N}$ are defined, in which $\mathcal{V}$ denotes the type of a single vertex. $st(a, D)$ is the length of the longest path up to and without a, $rt(a, D)$ is the length of the longest path starting from and including a and $l(a, D)$ is the length of the longest path through a in D. The start time $s(a)$ of each operation is equal to $st(a, D)$. For all three functions holds that parameter D is left out when no confusion can occur. □

The following theorem generalizes an observation made by Taillard [1994].

Theorem 3.1. *Given are a digraph D and functions st and rt. Let $D' = rev(D, a, b)$ with $b = sm_a$ and (a, b) on a longest path in D. Let $mx(D', a, b)$ denote the length of the longest path through a or b in D'. Furthermore, let l_2 denote the length of the second longest path in D —which can be equal to $f(D)$— and let k be the length of the longest path in D* not *through a or b. Then,*

(1) $$mx(D', a, b) \geq f(D) \Rightarrow f(D') = mx(D', a, b),$$

(2) $$l_2 < mx(D', a, b) < f(D) \Rightarrow f(D') = mx(D', a, b) \text{ and}$$

(3) $$l_2 \geq mx(D', a, b) \Rightarrow f(D') = \max\{mx(D', a, b), k\}.$$

This implies that $mx(D', a, b)$ is a lower bound for $f(D')$.

Proof. The paths through a or b in D are the only paths which possibly do not exist in D' as a consequence of the reversal of arc (a, b). Therefore,

(1) if $mx(D', a, b) \geq f(D)$ then $f(D') = mx(D', a, b)$, because all paths in D' not through a or b have lengths at most $f(D)$.

(2) all paths in D' not through a or b have length at most l_2 with $l_2 < mx(D', a, b)$. All paths in D' through a or b have length at most $mx(D', a, b)$ and at least one path has this length. Therefore $f(D') = mx(D', a, b)$.

(3) all paths in D' not through a or b have length at most k and all paths in D' through a or b have length at most $mx(D', a, b)$. Therefore $f(D') = \max\{mx(D', a, b), k\}$.

□

Theorem 3.1 proves that a non-trivial lower bound for the cost of a neighbour can be computed; Corollary 3.1 proves that this can be done in constant time.

Corollary 3.1. *The lower bound $mx(D', a, b)$ given in Theorem 3.1 can be computed in constant time if $st(c, D)$ and $rt(c, D)$ are available for all vertices c.*

Proof. From Figure 2.2 it can be inferred that

$$\begin{aligned}
st(b, D') &= \max\{st(pm_a, D) + d(pm_a),\ st(pj_b, D) + d(pj_b)\},\\
st(a, D') &= \max\{st(b, D') + d(b),\ st(pj_a, D) + d(pj_a)\},\\
rt(a, D') &= \max\{rt(sj_a, D),\ rt(sm_b, D)\} + d(a), \text{ and}\\
rt(b, D') &= \max\{rt(a, D'),\ rt(sj_b, D)\} + d(b).
\end{aligned}$$

All these expressions can be computed in constant time and since $l(a, D') = st(a, D') + rt(a, D')$, $l(b, D') = st(b, D') + rt(b, D')$ and $mx(D', a, b) = \max\{l(a, D'),\ l(b, D')\}$, $mx(D', a, b)$ as well can be computed in constant time. □

The neighbourhood search strategy is based on the previous theorem and corollary. The advantage of this search is its speed, since for every neighbour a lower bound for the cost can be computed in constant time; the disadvantage, however, is the loss of accuracy, since a neighbour with the smallest lower bound, is not necessarily a neighbour with the lowest cost. This makes it hard to apply an aspiration level to this neighbourhood search strategy, since the exact cost of a neighbour is not known. This makes it impossible to decide in $\mathcal{O}(1)$ whether a neighbour is a better solution than the best solution found so far. Before the neighbourhood search strategy is described some definitions are needed.

Definition 3.2. Given a digraph $D = (V, A \cup E')$ in which the set A represent the job precedence constraints and the set E' the machine capacity constraints. Let $\lambda(D)$ denote the set of arcs that are on a longest path in D. $M(D)$ denotes the set of legal arc reversals on such a longest path, i.e.

$$M(D) = \lambda(D) \setminus A \setminus \{e \mid e \in E' \wedge e \text{ is an internal arc}\}.$$

□

Given a digraph D and a tabu list tl the function *Propose*$(D, M(D), tl)$, with $M(D) \neq \emptyset$, returns an arc based on the following strategy. As long as $M(D) \setminus tl = \emptyset$, the arc that has been on the tabu list for the longest time is removed from this list. Since $M(D) \neq \emptyset$ this repetition always terminates. Secondly, for all arcs $(a, b) \in M(D) \setminus tl$ the lower bound $mx(D', a, b)$ is computed. Finally, the arc for which this lower bound is minimal is returned. It is possible that a returned arc was tabu before the execution of *Propose*, but is not tabu on return. This is due to the side effect of *Propose*: the shortening of the tabu list. Figure 3.1 presents an abstract algorithm of the above description.

```
func Propose(D : 𝒟; M : set of 𝒜; var tl : list of 𝒜) : 𝒜
{ precondition: D ∈ 𝒟 ∧ M = M(D) ∧ M ≠ ∅ ∧ st and rt have been computed;
  return: an arc in M
}
|[var m, m′ : 𝒜;
  do M \ tl = ∅ →
      {shorten the tabu list}
      tl := tail(tl)
  od;
  m :∈ M \ tl;
  {notation: m = (a, b) ⇒ mx(D′, m) = mx(D′, a, b)}
  forall m′ ∈ M \ tl do →
      if mx(D′, m′) < mx(D′, m) → m := m′
      □ mx(D′, m′) ≥ mx(D′, m) → skip
      fi
  od;
  return m
]|
```

Figure 3.1: The neighbourhood search strategy.

Remark 3.1. If the set $M(D)$ is empty the function *Propose* is not called, because in that case digraph D represents an optimal solution (see Corollary 2.1). $\square$

Before the function *Propose* is called , a longest path has to be constructed in order to obtain $M(D)$. This construction has time complexity $\mathcal{O}(N)$. The worst time complexity of *Propose* is $\mathcal{O}(N + T)$. Since $T \ll N$ the time complexity of *Propose* is $\mathcal{O}(N)$. Therefore, the time complexity of the neighbourhood search strategy is $\mathcal{O}(N)$.

The second parameter of *Propose* is initialized with $M(D)$ and in order to construct this set, a longest path has to be available. For this purpose ànd for the computation of the lower bound for the cost of all the neighbours, both $st(a)$ and $rt(a)$ have to be available. The next section presents the abstract algorithms that respectively compute $st(a)$ and $rt(a)$ for each a in V.

3.2 Computation of the longest path

In this section the algorithms that are needed to initialize st and rt are derived. The procedure *Init_rt* has to compute the length of the longest path from all vertices in a digraph D and, similarly, *Init_st* has to compute the length of the longest path up to all vertices in D.

To compute rt the following invariants are used

$\{P_0\}$ $\quad$ $(\forall a : a \in V_0 : rt(a) \text{ is computed})$ and

$\{P_1\}$ $\quad$ $(\forall a : a \in V_1 : \{sm_a, sj_a\} \subseteq V_0 \cup \{\diamond_s\})$.

V_1 is the set of operations for which both the successor on the machine as well as the one in the job are a member of V_0. Therefore, it is possible to compute rt for these operations. The length of a longest path in a digraph D is the maximum of all $rt(a)$ for which $pm_a = pj_a = \diamond_p$. The algorithm that is

proc *Init_rt*($D \in \mathcal{D}$; **var** $rt : \mathcal{V} \times \mathcal{D} \to \mathbb{N}$)
{ precondition: $D \in \mathcal{D}$, $D = (V, E)$;
  postcondition: for all vertices $a \in V$, $rt(a)$ has been computed
}
|[**var** $V_0,\ V_1$: **set of** $\mathcal{V}$; $a : \mathcal{V}$;
  $V_0,\ V_1 := \emptyset,\ \emptyset$;
  $\{P_0 \wedge P_1\}$
  forall $a \in V$ **do** $\to$
    if $sm_a = \diamond_s \wedge sj_a = \diamond_s \to V_1 := V_1 \cup \{a\}$
    $\Box\, sm_a \neq \diamond_s \vee sj_a \neq \diamond_s \to$ skip
    fi
  od;
  $\{P_0 \wedge P_1\}$
  do $V_1 \neq \emptyset \to$
    $a :\in V_1$;
    if $sm_a = \diamond_s \wedge sj_a = \diamond_s \to rt(a) := d(a)$
    $\Box\, sm_a = \diamond_s \wedge sj_a \neq \diamond_s \to rt(a) := rt(sj_a) + d(a)$
    $\Box\, sm_a \neq \diamond_s \wedge sj_a = \diamond_s \to rt(a) := rt(sm_a) + d(a)$
    $\Box\, sm_a \neq \diamond_s \wedge sj_a \neq \diamond_s \to rt(a) := \max\{rt(sm_a),\ rt(sj_a)\} + d(a)$
    fi;
    if $sm_{pj_a} \in V_0 \cup \{\diamond_s\} \to V_1 := V1 \cup \{pj_a\}$
    $\Box\, sm_{pj_a} \notin V_0 \cup \{\diamond_s\} \to$ skip
    fi;
    if $sj_{pm_a} \in V_0 \cup \{\diamond_s\} \to V_1 := V1 \cup \{pm_a\}$
    $\Box\, sj_{pm_a} \notin V_0 \cup \{\diamond_s\} \to$ skip
    fi;
    $V_0,\ V_1 := V_0 \cup \{a\},\ V_1 \setminus \{a\}$
    $\{P_0 \wedge P_1\}$
  od
  $\{P_0 \wedge P_1 \wedge V_1 = \emptyset \wedge D$ is acyclic $\Rightarrow$ for all vertices $a \in V\ rt(a)$ has been computed$\}$
]|

Figure 3.2: The initialization of rt.

obtained with the invariants P_0 and P_1 is known as Bellmann's labeling algorithm [Lawler, 1976]. It is presented in Figure 3.2.

Remark 3.2. Some vertices in V have no predecessors in their job. The algorithm, however, evaluates the expression sm_{pj_a}, which is not defined in this case. To this extend $sm_{\diamond_p}$ is defined as $\diamond_p$, since then operation $\diamond_p$ is not added to V_1. Completely analogous to this $sj_{\diamond_p}$ is defined as $\diamond_p$ too. Based on the symmetry with *Init_st*, both $pj_{\diamond_s}$ and $pm_{\diamond_s}$ are defined as $\diamond_s$. □

Because for all operations in V rt has to be computed, the time complexity of *Init_rt* for a digraph with N operations is $\mathcal{O}(N)$. In the next chapter an optimization for this algorithm is presented. The algorithm for *Init_st* is similar to *Init_rt*.

3.3 The tabu list

In this section issues like *how* to mark an arc as forbidden and *what* value for the tabu tenure to use for the tabu list are discussed.

When using a tabu list with a maximum length of T, cycles of length less than or equal to T are prevented (see Section 1.2). Because T is usually a small number, only a few solutions are *remembered* by the tabu list. Therefore, a tabu list is often referred to as a *short term memory*. It is clear that such a tabu list does not prevent cycles with length greater than T. Two strategies are often used to try to cope with these *long* cycles.

- Maintain a long term memory along with the tabu list in order to *detect* long cycles.
- Frequently alternate the tabu tenure in order to (try to) *avoid* long cycles.

Because an accurate detection of long cycles is quite difficult and time consuming, the second alternative is chosen. However, there is no guarantee that this strategy will indeed *prevent* long cycles from occurring.

Until now an arc was marked as a forbidden arc by adding it to the tabu list. An algorithm based on this representation is much to slow, since evaluation of the expression $e \in tl$ for an arc e has average time complexity $\mathcal{O}(T)$. Since this expression is evaluated several times in every iteration of the algorithm, a different representation must be used in order to obtain a faster algorithm.

Definition 3.3. The function $tl' : \mathcal{A} \to \mathbb{N}$ is called a *tabu function*. For each $e \in \mathcal{A}$, $tl'(e)$ is the iteration number in which the arc e^{-1} was reversed. Then the following relation holds between the two representations:

$$e \in tl \equiv tl'(e) + T \geq It,$$

in which It is the current iteration number. □

Evaluation of the right hand expression has time complexity $\mathcal{O}(1)$, granted tl' is implemented as an array. After the reversal of an arc e, the arc e^{-1} is simply made tabu with the statement $tl'(e^{-1}) := It$, automatically making it tabu for T iterations. Initially $tl(e) = -\infty$ for all edges e.

Regarding the implementation of this representation, the memory complexity of function tl' is $\mathcal{O}(N^2)$. For large N this is considered a serious problem and therefore the following is done. When an arc (a, b) is reversed all the arcs (b, x) for $x \in V$ are made tabu. At first this is correct, since (b, a) is the only arc in D' that satisfies this condition. When, however, there was an arc (b, c) in D, there is an arc (a, c) in D' after the reversal. When this arc is reversed the arc (b, a) disappears and the arc (b, c) is created. This arc is tabu according to tl' —assumed that b is still tabu— but the arc (c, b) was never reversed (in fact, it possibly never existed). See Figure 3.3 for an example.

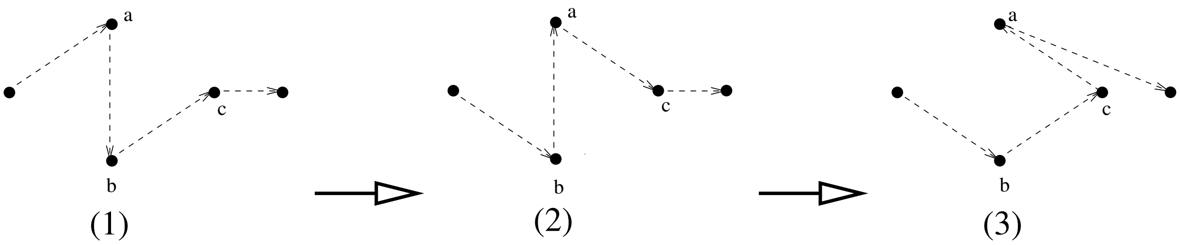

Figure 3.3: Example of a part of an instance (1), the same part after the reversal of arc (a, b) (2), and after the reversal of arc (a, c) (3).

This representation introduces a small loss of accuracy, but a huge gain of space, since the memory complexity of this representation is $\mathcal{O}(N)$. This is because tl' can now be represented as a function from $\mathcal{V}$ to $\mathbb{N}$ instead of $\mathcal{A}$ to $\mathbb{N}$. The decision to employ this representation is therefore well justified. The abstract algorithms presented from now on continue to use the notation that has been used in the previous algorithms, i.e. the list notation, to avoid confusion.

In the final section of this chapter, all the previous functions are combined to construct a first version of the sequential tabu search algorithm for the job shop scheduling problem.

3.4 The algorithm

This section presents a first version of the sequential algorithm based on the algorithms of the previous sections.

The initial digraph used for this algorithm is constructed randomly. This is done regarding the future parallel version of the algorithm in which different initial solutions are needed. The time complexity of this initialization algorithm is $\mathcal{O}(N)$. The drawback of randomly constructed schedules is their poor quality.

The algorithm operates as follows. At first the length of the longest path from and to all vertices

---

proc *Tabu_Search*(**var** $D : \mathcal{D}$)
{ precondition: $D \in \mathcal{D}$;
  postcondition: D is the graph with the shortest path found
}
|[**con** $MaxIt,\ T$: int; $\{MaxIt \geq 0 \wedge T > 0\}$
  var $D_{opt} : \mathcal{D};\ tl$:**list of** $\mathcal{V}$; It : int; $e : \mathcal{A}$; $st,\ rt : \mathcal{V} \times \mathcal{D} \to \mathbb{N}$; M: **set of** $\mathcal{A}$;
  $D_{opt},\ tl,\ It := D,\ [\ ],\ 0$;
  do $It < MaxIt \to$
    Init_rt(D, rt);
    Init_st(D, st);
    Construct(M);
    $\{M = M(D)\}$
    if $M = \varnothing \to$ {D represents a global minimum} **return**
    $\square\ M \neq \varnothing \to$ skip
    fi;
    $e :=$ *Propose*$(D,\ M,\ tl)$;
    {notation: $e = (a, b) \Rightarrow e_0 = a \wedge e_1 = b$}
    $tl := tl \oplus_T [e_1]$;
    $D := rev(D, e_0, e_1)$;
    if $f(D) < f(D_{opt}) \to D_{opt} := D$
    $\square\ f(D) \geq f(D_{opt}) \to$ skip
    fi;
    $It := It + 1$
  od;
  $D := D_{opt}$
]|

Figure 3.4: A tabu search algorithm for the job shop scheduling problem.

in the graph is computed by respectively *Init_rt* and *Init_st*. Then the set of legal moves $M(D)$ is constructed. This algorithm is not presented since all it does is the construction of a longest path and leave out the internal- and job capacity arcs. If $M(D) = \emptyset$, digraph D represents a global minimum and the algorithm is terminated. Otherwise an arc reversal is proposed by the function *Propose*, and finally the tabu list and the current graph are updated. It it clear that the algorithm of Figure 3.4 looks a lot like the general tabu search algorithm of Figure 1.2.

Some remarks are in place, since the implemented algorithm is somewhat different from the presented algorithm.

Remark 3.3. The tabu tenure T in the algorithm is still a constant. This is done to avoid unnecessary confusion. In the implemented algorithm, however, it is randomly generated every $T + 2$ iterations. See Section 6.1 for more details. □

Remark 3.4. The abstract algorithm constructs the set $M(D)$ before calling the function *Propose*. The procedure *Construct* has to construct a longest path and leave out the internal- and job capacity arcs. The function *Propose* traverses this path and evaluates the result of each arc reversal on this path. The first operation of a longest path can be obtained in $\mathcal{O}(M)$ time, when keeping record of the first operation on each machine and when rt has been computed. When this first operation is known, a longest path does not have to be constructed in advance, since it can be reconstructed during the traversal, saving both time and space. □

The next chapter presents some optimizations to this algorithm. The optimizations will concern the speed of the algorithm and the quality of the found solution.

CHAPTER 4

Optimizations to the algorithm

This chapter discusses two optimizations to the sequential algorithm in the previous chapter. The first one optimizes the speed of the algorithm and the latter will improve the quality of the solutions.

4.1 Optimizing *Init_rt* and *Init_st*

The computations of the length of the longest path to and from all vertices are the most expensive operations —with respect to the computation time— of the algorithm. Profiling has revealed that these two operations consume up to 90 percent of the total computation time. Since in every iteration of the algorithm these computations are performed, optimization of these operations will significantly decrease the amount of computation time.

The time complexity of *Init_rt* is $\mathcal{O}(N)$ and its execution takes $2N$ iterations: N iterations to initialize the set V_0, and N iterations to compute rt for all vertices. This can be optimized by keeping record of the last operation on each machine. In this case the initialization part of *Init_rt*, i.e. the construction of the set V_0, can be done in $\mathcal{O}(M)$ instead of $\mathcal{O}(N)$. This results in an algorithm with the same time complexity of $\mathcal{O}(N)$, but with only $N + M$ iterations during execution. This makes a great difference for large instances, since for most of these instances $M \ll N$. There is, however, a much better optimization possible for improving the execution time of *Init_rt*.

The algorithm derived in Section 3.2 computes the longest path from each operation in the digraph. The value of $rt(a, D)$, for a vertex $a \in D$ —i.e. the length of the longest path from a in D— does not always change after the reversal of an arc, i.e. $rt(a, D) = rt(a, D')$. If rt is implemented as an array, the value for these vertices in the domain of rt do not have to computed since the *old* value is correct. The following theorem proves this.

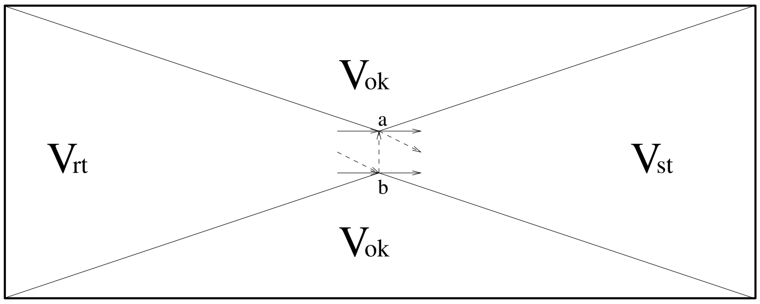

Figure 4.1: A digraph D' obtained from digraph D by the reversal of arc (a, b). The values $st(c, D')$ and $rt(c, D')$ for $c \in V_{ok}$ are equal to $st(c, D)$ and $rt(c, D)$ respectively.

Theorem 4.1. *Let $D' = (V, E)$ be a digraph in $\mathcal{D}$, (b, a) be an arc in E and let D' be $rev(D, a, b)$. L is defined as the set of vertices with no successors in the job and on the machine, i.e. $L = \{a | a \in V \wedge sj_a = sm_a = \diamond_s\}$. If for a vertex $c \in D'$ holds that there is no path from c to any vertex in L through vertex a then $rt(c, D') = rt(c, D)$.*

Proof. If for a vertex $c \in D'$ there is no path from c to a vertex in L through vertex a then there also is no path from c to a vertex in L through b, since (b, a) is an arc in D'. Because all paths in D' from c do not contain a or b these paths are the same as in D, since paths in D through a or b are the only paths in D that possibly do not exist in D'. □

According to the previous theorem, rt only has to be computed for the vertices for which there exists a path through a. Algorithm *Part_Init_rt* in Figure 4.2 first computes the set V_{rt} of vertices from which there is a path through a (see Figure 4.1). The value of rt of the vertices in $V \setminus V_{rt}$ is correct by construction of V_{rt} and Theorem 4.1. Next the algorithm computes the length of the longest path from each vertex c in V_{rt} in the same way as *Init_rt* does. The invariants used to derive *Part_Init_rt* are:

$\{P_0\}$ $(\forall c : c \in V \setminus V_0 : rt(c)$ is correct$)$,

$\{P_1\}$ $(\forall c : c \in V_1 : \{sm_c, sj_c\} \subseteq V \setminus V_0 \cup \{\diamond_s\})$ and

$\{P_2\}$ $(\forall c : c \in V_0 :$ there is a path from c through $a)$.

Remark 4.1. The second repetition of *Part_Init_rt* looks similar to the one in *Init_rt*. However, there is a significant difference, since the interpretation of the set V_0 is different. In *Init_rt* V_0 represents the set of vertices for which rt is correct; in *Part_Init_rt* V_0 represents the set of vertices for which rt has to be computed. □

The algorithm *Part_Init_st* is quite similar to the algorithm in Figure 4.2. The time complexity of *Part_Init_rt* is still $\mathcal{O}(N)$, like *Init_rt*, but its execution takes only $2|V_{rt}|$ iterations. Based on symmetry, the execution of *Part_Init_st* takes $2|V_{st}|$ iterations (see Figure 4.1). One iteration of the initialization part of *Init_rt* takes less time than one iteration of the initialization part of *Part_Init_rt*, but the gain of the number of iterations in comparison with the last versions of *Init_rt* and *Init_st* is $2(N + M) - 2(|V_{rt}| + |V_{st}|)$. Since $|V_{rt}| + |V_{st}| \leq N + 2$ this gain is practically always positive. Empirical analysis has revealed that this optimization decreases the computation time of the algorithm with 35 to 40 percent.

The next section optimizes the sequential algorithm in order to obtain better quality solutions.

4.2 Intensification of solutions

In order to obtain good quality solutions with the derived algorithm in Figure 3.4, the value for the maximum number of iterations has to be chosen very large. The tabu search algorithm of Taillard [1994] —using only a short term memory— needed up to two million iterations to find a global optimum for the ft10-instance [Fisher & Thompson, 1963]. Since a single iteration of the algorithm in Figure 3.4 has a short execution time, the value $MaxIt$ can be chosen large, while maintaining a reasonable execution time. This strategy to find good quality solutions is a very poor one, because it is possible that such an algorithm searches only a small part of the solution space, due to long cycles.

In order to make sure that a tabu search algorithm searches a bigger part of the solution space, one of the two following strategies is often used.

- **Diversification.** Upon termination of the algorithm, repeatedly restart with a different initial solution.

$Part\_Init\_rt\ (D : \mathcal{D};\ \mathbf{var}\ rt : \mathcal{V} \times \mathcal{D} \to \mathbb{N}; e : \mathcal{A})$
$\{$ precondition: $D \in \mathcal{D}$, $D = (V, E) \wedge D$ was constructed by the reversal of arc $e \wedge e = (a, b)$;
postcondition: rt has been computed for all vertices in V
$\}$
$|[\mathbf{var}\ V_0,\ V_1 : \mathbf{set\ of}\ \mathcal{V};\ c : \mathcal{V};$
$V_0,\ V_1 := \{a, b\},\ \{a, b\};$
$\{P_2\}$
$\mathbf{do}\ V_1 \neq \varnothing \to$
$c :\in V_1;$
$\mathbf{if}\ pm_c \notin V_0 \cup \diamond_p \to V_0,\ V_1 := V_0 \cup \{pm_c\},\ V_1 \cup \{pm_c\}$
$\square\ pm_c \in V_0 \cup \diamond_p \to \text{skip}$
$\mathbf{fi};$
$\mathbf{if}\ pj_c \notin V_0 \cup \diamond_p \to V_0,\ V_1 := V_0 \cup \{pj_c\},\ V_1 \cup \{pj_c\}$
$\square\ pj_c \in V_0 \cup \diamond_p \to \text{skip}$
$\mathbf{fi};$
$V_1 := V_1 \setminus \{c\};$
$\{P_2\}$
$\mathbf{od};$
$\{P_2 \wedge V_0 = V_{rt}\}$
$V_1 := \{a\};$
$\{P_0 \wedge P_1\}$
$\mathbf{do}\ V_1 \neq \varnothing \to$
$c :\in V_1;$
$\mathbf{if}\ sm_c = \diamond_s \wedge sj_c = \diamond_s \to rt(c) := d(c)$
$\square\ sm_c = \diamond_s \wedge sj_c \neq \diamond_s \to rt(c) := rt(sj_c) + d(c)$
$\square\ sm_c \neq \diamond_s \wedge sj_c = \diamond_s \to rt(c) := rt(sm_c) + d(c)$
$\square\ sm_c \neq \diamond_s \wedge sj_c \neq \diamond_s \to rt(c) := \max\{rt(sm_c),\ rt(sj_c)\} + d(c)$
$\mathbf{fi};$
$\mathbf{if}\ sm_{pj_c} \notin V_0 \cup \{\diamond_p\} \to V_1 := V1 \cup \{pj_c\}$
$\square\ sm_{pj_c} \in V_0 \cup \{\diamond_p\} \to \text{skip}$
$\mathbf{fi};$
$\mathbf{if}\ sj_{pm_c} \notin V_0 \cup \{\diamond_p\} \to V_1 := V1 \cup \{pm_c\}$
$\square\ sj_{pm_c} \in V_0 \cup \{\diamond_p\} \to \text{skip}$
$\mathbf{fi};$
$V_0,\ V_1 := V_0 \setminus \{c\},\ V_1 \setminus \{c\}$
$\{P_0 \wedge P_1\}$
$\mathbf{od}$
$\{P_0 \wedge P_1 \wedge V_1 = \varnothing \wedge D$ is acyclic $\Rightarrow$ for all vertices $c \in V\ rt(c)$ has been computed$\}$
$]|$

Figure 4.2: The initialization of rt.

- **Intensification.** During execution of the algorithm keep record of a set of good solutions. Upon termination, restart with a neighbour of one of these solutions —*that has* NOT *been used before*— as the initial solution.

The disadvantage of diversification is the total loss of information that was obtained during the

proc *Tabu_Search'*(s: $\mathcal{S}$; tl: **list of** $\mathcal{M}$; **var** bul: **stack of**($\mathcal{S}\times$**set of** $\mathcal{M}\times$**list of** $\mathcal{M}$); **var** s_{opt}: $\mathcal{S}$)
{ precondition: $s \in \mathcal{S} \wedge s_{opt} \in \mathcal{S} \cup \{s_\infty\}$;
}
|[**con** $MaxIt, T$: int; $\{MaxIt > 0 \wedge T > 0\}$
  var It: int; m: $\mathcal{M}$;
  $It := 0$;
  do $It < MaxIt \rightarrow$
    $m :=$ *Propose*$(s, M(s), tl)$;
    if $f(s) < f(s_{opt}) \wedge M(s) \setminus \{m\} = \varnothing \rightarrow s_{opt} := s$
    $\Box\, f(s) < f(s_{opt}) \wedge M(s) \setminus \{m\} \neq \varnothing \rightarrow s_{opt}, bul := s, \text{push}(\langle s, M(s) \setminus \{m\}, tl\rangle, bul)$
    $\Box\, f(s) \geq f(s_{opt}) \qquad\qquad\qquad \rightarrow$ skip
    fi;
    $tl := tl \oplus_T [m^{-1}]$;
    $s := s \odot m$;
    $It := It + 1$
  od;
]|

Figure 4.3: A general tabu search algorithm used for intensification.

Load(**var** bul: **stack of**($\mathcal{S}\times$**set of** $\mathcal{M}\times$**list of** $\mathcal{M}$); **var** tl: **list of** $\mathcal{M}$; **var** s: $\mathcal{S}$)
{ precondition: $s \in \mathcal{S} \wedge bul \neq \bot$;
}
|[**var** m: $\mathcal{M}$; W: **set of** $\mathcal{M}$;
  $\langle s, W, tl\rangle, bul :=$ top(bul), pop(bul);
  $\{W \neq \varnothing\}$
  $m :=$ *Propose*(s, W, tl);
  if $W \setminus \{m\} = \varnothing \rightarrow$ skip
  $\Box\, W \setminus \{m\} \neq \varnothing \rightarrow bul := \text{push}(\langle s, W \setminus \{m\}, tl\rangle, bul)$
  fi;
  $tl := tl \oplus_T [m^{-1}]$;
  $s := s \odot m$
]|

Figure 4.4: Initialization of the solution for the next iteration.

preceding executions of the algorithm. This, in combination with the fact that diversification is provided by the parallel version of the algorithm in the next chapter, imply that intensification is the better alternative in this case.

A stated before in Section 1.2, keeping track of solutions is very space consuming. Therefore only a few solutions can be stored for intensification. The set of stored solutions is represented by a *stack* bul. Every time a solution with lower cost than so far is found, it is *pushed* on this stack and, upon termination of the algorithm, a neighbour of the solution on *top* of the stack is used as the initial solution for the next iteration of the algorithm. When all the neighbours of the solution on top of the stack have been used, this solution is *popped* from the stack. This implies that, along with the actual solution, all the moves that lead to its *un inspected* neighbours have to be stored as well.

The algorithm *Tabu_Search'* presented in Figure 4.3 performs a single run of the tabu search. There are only some minor differences with the algorithm in Figure 1.2. After the function *Propose* has returned a move m, the following is done in addition to the previous tabu search algorithm, when the current solution is better than the best solution found so far. If m is the only legal move for this solution (i.e. $M(s) = \{m\}$), s is not stored for intensification, since the only neighbour of s ($s \odot m$), will be the solution used in the next iteration. When, however, $\{m\} \subset M(s)$ there is more than one neighbour of s. In this case s will be pushed on the stack, along with the set of moves that lead to its neighbours — except m itself— and the current tabu list. Therefore the type of the stack *bul* is the Cartesian product $\mathcal{S} \times$ **set of** $\mathcal{M} \times$ **list of** $\mathcal{M}$.

Some remarks are in place, since the implemented version of *Tabu_Search'* is slightly different from this general version.

Remark 4.2. The implemented algorithm only stores solutions that are a local minimum. To check whether a solution is a local minimum can be done in $\mathcal{O}(1)$ in the implemented algorithm. □

Remark 4.3. The implemented algorithm uses the tabu list representation of Definition 3.3. Since this representation uses the current iteration number to mark a reversal as forbidden, this number has to be stored as well. □

Figure 4.4 presents the abstract algorithm *Load* that performs the initialization for the next execution of *Tabu_Search'*. A stored solution s, and its legal moves W—that will lead to un inspected neighbours— will be popped from the stack. The initial solution for the next iteration is a neighbour $s \odot m$ with $m \in W$. Here the second argument of *Propose* is used, since the set of moves that lead to un inspected neighbours can not be derived from the solution. If $W \setminus \{m\}$ is not empty then s has more than one neighbour and s is pushed on the stack again along with $W \setminus \{m\}$ and the tabu list tl

A general tabu search algorithm is presented in Figure 4.5. Initially the stack *bul* is the empty stack, denoted by $\perp$. When *Tabu_Search'* terminates, procedure *Load* is executed when there are still solutions on the stack for intensification. If the stack is empty the algorithm terminates.

The time complexity of this algorithm depends strongly on the following two aspects:

- The number of times a better solution is found. Each better solution is stored for intensification; when more solutions are stored, the time complexity will increase.

Tabu_Search_With_Intensification(**var** s: $\mathcal{S}$)
{ precondition: $s \in \mathcal{S}$, s_∞ is a special constant with $f(s_\infty) = \infty$;
}
|[**var** bul : **stack of**($\mathcal{S} \times$ **set of** $\mathcal{M} \times$ **list of** $\mathcal{M}$); run: Bool; tl: **list of** $\mathcal{M}$; s_{opt}: $\mathcal{S}$;
  tl, bul, s_{opt}, run := [ ], $\perp$, s_∞, true;
  do $run \rightarrow$
    Tabu_Search'(s, tl, bul, s_{opt});
    if $bul = \perp \rightarrow run$:= false
    □ $bul \neq \perp \rightarrow$ *Load*(bul, tl, s)
    fi
  od;
  $s := s_{opt}$
]|

Figure 4.5: A general tabu search algorithm with intensification.

- The size of the neighbourhood of the stored solutions. Since all neighbours of a stored solution are used as an initial solution once, it is clear that this size will have great influence on the time complexity.

Since the time complexity to find a local minimum is not known (see Section 1.1), the time complexity to find a better solution given a solution is also not known.

This algorithm is the final version of the sequential algorithm. It is a general algorithm, and therefore it can be applied to all combinatorial optimization problems. A tabu search search algorithm with intensification for the job shop scheduling problem can easily be obtained by combining this algorithm and the one of Section 3.4. The results of this algorithm are presented in Section 6.1.

The next chapter first presents some information about parallel local search in general, followed by the derivation of two parallel versions of this algorithm.

CHAPTER 5

The parallel algorithm

This chapter discusses the design of two parallel tabu search algorithms based on the derived sequential algorithm of the previous chapter. The first section provides some background information about parallel local search in general, and the last two sections each discuss a version of the parallel tabu search algorithm.

5.1 An introduction to parallel local search

This section discusses the underlying concepts used in the various approaches to introduce parallelism in local search, abstracting from implementation issues. There are two possible approaches to introduce parallelism in local search: a *tailored* and a *general* approach.

A tailored approach requires parallel execution of a problem specific part of the algorithm. Cost computation, for example, is a typical problem dependent part of a local search algorithm. If one of these problem dependent parts is very time consuming, the running time of the algorithm can be decreased substantially by efficient parallel execution of this part. An example of parallel cost computation for the job shop scheduling problem is given by Taillard [1994], who uses a parallel longest path algorithm to compute the length of a schedule.

A general approach —unlike a tailored one— can be applied to a broad class of local search problems. Again, two approaches are possible: *single-walk* and *multiple-walk* parallelism. In a single-walk algorithm only a single walk in the solution space is carried out, whereas in a multiple-walk algorithm several walks are performed simultaneously.

Multiple-walk parallelism. Multiple-walk algorithms have inherent parallelism, which can be exploited by distributing the walks over a number of processors, and performing them simultaneously. The class of multiple-walk algorithms can be divided in algorithms that perform multiple interacting walks and algorithms that perform multiple independent walks.

The approach that uses multiple independent walks, can be considered as the most straightforward one to introduce parallelism in local search, since no communication is required between different walks. A disadvantage of multiple independent runs is its *load imbalance*, i.e. the difference in the amount of work of each processor. This is due to the different computation times of each single walk.

Interaction of walks can be used to faster direct the search into promising areas of the solution space. When processors interact with each other, they exchange information. This information —such as the occurrence of low cost solutions in a population of solutions, or the availability of good parts of individual solutions— can speed up the search. Load balancing can be achieved by defining an equal amount of work to be performed between two interacting walks.

Two ways to express the performance of a parallel algorithm are its *speed-up*, which compares the execution time on one processor with that of P processors, and its *efficiency*.

Definition 5.1. Let $T(P, N)$ denote the execution time of the algorithm on P processors, for an instance with *problem size* N. The speed-up s and the efficiency e can be defined as

$$s(P, N) = \frac{T(1, N)}{T(P, N)},$$
$$e(P, N) = \frac{s(P, N)}{P}.$$

□

Another important issue is the scalability of an algorithm, which refers to its performance for increasing number of processors.

Good efficiencies *—to find a solution with a given quality—* can be obtained with multiple-walk parallelism, but the amount of speed-up depends on the time needed to find the first local minimum, the time needed to find subsequent local minima, and the desired solution quality. See [Verhoeven, 1996] for more information.

Single-walk parallelism. In an algorithm with single-walk parallelism one or more consecutive steps in the solution space are made in parallel. The class of single-walk algorithms can be divided in algorithms that perform a single step and algorithms that perform multiple steps in the solution space.

Single-step parallel local search is based on parallel neighborhood exploration. Each processor inspects only a part of the entire neighbourhood and proposes a move given a current solution. Only one of these moves is effectuated in order to construct a new solution. A way to apply this approach to the derived algorithm for the job shop scheduling problem is to examine the arc reversals on *all* longest paths, in stead of only one. Unfortunately, for small instances there is in most cases exactly one longest path, so no speed-up can be achieved. For larger instances, with more than one longest path, the drawback of this approach is its scalability, since there is a maximum to this number of paths. When using more processors then there are paths, speed-up does not increase and efficiency gets worse.

An advantage of single-step parallelism is its general applicability. Moreover, it results trivially in a distributed neighborhood structure that is isomorphic with a traditional neighborhood structure that uses the same exchange function. A distributed neighbourhood structure $\mathcal{D}$ is isomorphic with a neighbourhood structure $\mathcal{N}$ if, for any set of processors, holds that each local minimum of $\mathcal{N}$ is a local minimum of $\mathcal{D}$. Therefore, it is easily guaranteed that the parallel algorithm finds the same quality solutions as the sequential algorithm.

Multiple-step parallel local search is also based on parallel neighbourhood exploration, but, instead of only effectuating one of the proposed moves, (a subset of) all the proposed moves is effectuated. Since effectuation of a proposed move can prohibit effectuation of other moves, the proposed moves are combined such that a feasible solutions is constructed. This implies that not all proposed moves can in fact be effectuated. Essential to achieve a good speed-up with multiple-step parallelism is that a large proportion of the proposed moves can in deed be effectuated in the combination step. The applicability of multiple-step parallelism therefore depends strongly on the problem at hand, as the problem must allow effectuation of several proposed moves for a given solution.

A way to apply this approach to the derived algorithm for the job shop scheduling problem is the following [Verhoeven & Aarts, 1995]. Let D be a digraph, (a, b) be a machine capacity arc on a longest path and let $D' = rev(D, a, b)$. If the second longest path in D does not go through a or b, Theorem 3.1 implies that the length of the longest path through a or b in D' is the length of the longest path in D' if it is longer than the second longest path in D; otherwise the second longest path in D is the longest path in D'. Hence, the second longest path in D is the longest path in many neighbours D' of D. A similar argument shows that the third longest path in D is the longest path in many neighbours

D'' of D'. A repetition of this argument shows that the p-th longest path in D is the longest path in many digraphs $D^{(p)}$, which can be obtained from D by p subsequent (cost decreasing) arc reversals. An algorithm in which processor p, $1 \leq p \leq P$ proposes an arc reversal on the p-th longest path, and effectuating a subset of these reversals is an example of a single-step multiple-walk algorithm.

Empirical analysis, however, has revealed that for this algorithm —being executed on 20 processors— an average of only four reversals, of all proposed reversals, can indeed be effectuated for instances with 300 operations. Trivially, this results in a poor speed-up of only four.

Based on the very good results of the sequential algorithm, and the difficulty to obtain good speed-up with the proposed approaches towards single-walk parallelism, multiple-walk parallelism is used for the design of the parallel tabu search algorithms. This is done in the next section.

5.2 The parallel algorithm with independent walks

This section discusses the design of the parallel tabu search algorithm with independent walks. In the next section, synchronisation is added to this algorithm in order to obtain better quality results in less computation time.

A parallel algorithm based on multiple independent walks is very simple since no global communication is needed during execution. The only communications needed are the initial broadcasting of the data to all the processors, and the collecting of the individual results upon termination.

The designed algorithm is based on the so called *master-slave topology* in which an additional algorithm is being executed on the *master*. This algorithm first starts the sequential tabu search algorithm on each *slave*, and then sends the data needed to construct an initial solution to all slaves. Since all slaves construct their initial schedule randomly, probably no two slaves start from the same initial solution. When all the slaves are terminated, the master collects the solutions from the slaves and it returns the best one of these solutions.

The quality of the solution found by such an algorithm, being executed on P processors, equals the quality of the best solution obtained with P executions of the sequential version of the algorithm. The computation time equals the worst computation time of these P executions.

Since no communication is needed, such algorithms have good efficiency. For the derived tabu search algorithm for the job shop scheduling problem, however, it is hard to obtain a good speed-up on a large number of processors when using randomly constructed initial solutions. As mentioned before the disadvantage of this algorithm is its load imbalance, since upon termination of a slave, other slaves continue their execution, while the terminated slave remains idle.

As stated before, good speed-up can be achieved if it is the objective to find a sub-optimal solution. When, however, the goal is to find a global optimum, all slaves have to terminate. In this case $T(P, N)$ is usually at least $T(1, N)$, which results in an extremely poor speed-up. Therefore, it is useful to apply a different criterion to judge the quality of a parallel algorithm: the algorithm's ability to find better quality solutions than the sequential algorithm, in the same amount of time.

It is very well possible that some slaves are searching parts of neighbourhood, that do not contain good quality solutions. Due to long cycles (see Section 3.3), these slaves may not be able to escape from these parts. In this case it makes sense to replace the set of solutions used for intensification, with a set that contains better solutions. This suggestion is used for the design of a multiple interacting walk algorithm in the next section.

proc *Slave_Synchronize*(**var** *bul*: **stack of** ($\mathcal{S}\times$**set of** $\mathcal{M}\times$**list of** $\mathcal{M}$))
|[*Send*($Master$, *bul*);
  {Wait for master}
  Receive($Master$, *bul*)
]|

Figure 5.1: A synchronisation algorithm for the slaves.

5.3 The parallel algorithm with interacting walks

In this section the design of the final parallel algorithm is discussed: a parallel interacting multiple-walk algorithm. The algorithm does the following. Periodically, all the slaves send all their schedules, plus the legal moves and the tabu list —stored for intensification— to the master. Then each slave waits until they receive the new solutions from the master, before the tabu search continues. See Figure 5.1 for an abstract algorithm of the above description.

The master receives all these solutions from the slaves, and stores them in the set S_{cur}. When all the solutions are received, the master constructs a subset S_{new} of all these schedules according to some criterion, i.e. the set S_{cur} is *stripped*. Then the set S_{new} is equally partitioned in p parts, for which p is the number of non-terminated slaves. Finally, to each slave that has not terminated yet, a part of this partition is send. Figure 5.2 presents an abstract algorithm for the synchronisation part of the master. Like the algorithm with multiple independent walks, upon termination of all the slaves, the master collects the best solution from all slaves and returns the best one. Note that, in order to reduce the number of communications, not the master, but the slaves initiate the synchronisation.

proc *Master_Synchronize*( )
|[**con** P; {$P \geq 1$, P is the number of processors}
  var W: **array** $[1..P]$ **of stack of** ($\mathcal{S}\times$**set of** $\mathcal{M}\times$**list of** $\mathcal{M}$);
    i: int; S_{cur}, S_{new}: **set of stack of** ($\mathcal{S}\times$**set of** $\mathcal{M}\times$**list of** $\mathcal{M}$);
  do $ThereAreTasksRunning \rightarrow$
    $S_{cur} := \emptyset$;
    forall i: $1 \leq i \leq P \wedge$ *ThereIsATaskRunningOnProcessor*(i) **do** $\rightarrow$
      Receive(i, $W[i]$);
      $S_{cur} := S_{cur} \cup \{W[i]\}$
    od;
    S_{new} := *Strip*(S_{cur});
    W := *Partition*(S_{new});
    { $W[1] \ldots W[P]$ is a partition of $S_{new} \wedge$
      $\neg$ *ThereIsATaskRunningOnProcessor*(i) $\Rightarrow W[i] = \perp \wedge$
      $(\forall i, j : 1 \leq i < j \leq P \wedge W[i] \neq \perp \wedge W[j] \neq \perp : -1 \leq |W[i]| - |W[j]| \leq 1)$
    }
    forall i: $1 \leq i \leq P \wedge$ *ThereIsATaskRunningOnProcessor*(i) **do** $\rightarrow$
      Send(i, $W[i]$)
    od
  od
]|

Figure 5.2: A synchronisation algorithm for the master.

This algorithm still suffers from load imbalance, but not as much as the algorithm with independent walks, since every slave is assigned an equally large set of schedules.

Remark 5.1. A way to reduce the load imbalance even more is established if not the number of solutions, but the number of neighbours of these solutions is equally partitioned. In this case the sum of all neighbours of all stored solutions on each slave is (almost) equal, for all slaves. □

The efficiency of this algorithm depends on the number of synchronisations, i.e. on the time interval between two subsequent synchronisations. When assumed that the constructed subset S_{new} is not always equal to the set of received solutions, the speed-up of this algorithm is significantly better than that of the previous algorithm. The implementation of the procedure *Strip* is completely user-dependent: its implementation is discussed in Section 6.3.

Remark 5.2. The implemented algorithm is based on this abstract algorithm. The difference is that in the implemented algorithm only the costs of the stored solutions are send to the master, since sending all solutions to the master is to space consuming. Based on these costs the master decides which slaves have to send (receive) schedules to (from) other slaves. □

The results of both the independent and the interacting algorithm, are presented in respectively Section 6.2 and Section 6.3.

CHAPTER 6

The results

In this chapter the results of the various algorithms are presented.

6.1 Results of the sequential algorithm

In this section the results of the sequential tabu search algorithm with intensification (TSI), derived in Chapter 4, are presented.

6.1.1 Parameter setting

It is very hard, if not impossible, to find an optimal parameter setting for a tabu search algorithm. A setting that results in a good solution for one instance, may not work so well for other instances. The quality of the schedule depends highly on the settings of

1. **the upper bound for the number of arc reversals.** When a large number of arc reversals is allowed on a schedule, usually a better result is obtained.
2. **the upper bound for the schedules that are stored for intensification.** Trivially, intensification over more schedules, often results in a better quality solution.

On one hand these upper bounds should be large in order to obtain good results; on the other hand, the computation time of the algorithm depends a lot the settings of these two parameters. Therefore, this setting has to be a compromise between good quality solutions, and acceptable computation times.

As stated before, the tabu tenure T is not constant, but it is randomly generated every $T + 2$ iterations. However —in order to prevent T from getting an extreme value— it is not completely random. T is generated in a way that —on the average— it is not smaller than the size of the neighbourhood. Each iteration, the size of the neighbourhood of current solution s is stored in a *frequency array* f, i.e each iteration the following assignment is performed

$$f(|\mathcal{N}_{1b}(s)|) := f(|\mathcal{N}_{1b}(s)|) + 1.$$

Every $T + 2$ iterations the value t , for which $f(t) = (\mathbf{max}\ i\ :\ i\ \in\mathbb{N}^+\ :\ f(i))$, is assigned to T and a random value smaller than three is added, i.e. $T\ := t{+}random(3)$. In a way, this will prevent the occurrence of long cycles, since T is usually not smaller than the size of the neighbourhood.

A maximum of five schedules is stored for intensification. In order to speed up the algorithm, the upper bound for the number of arc reversals is altered during execution, in the following way. Let the integer value r denote the number of solutions that has been popped from the stack, since a solution was pushed on this stack for the last time. In the implemented version of the algorithm, $MaxIt$ is assigned to in the following way

$$MaxIt := 5000,$$

if the current solution is better than the best solution found so far, and

(6.1) $$MaxIt := 5000 - 1000 \times r,$$

if a neighbour of a stored schedule is used as the initial schedule for the next iteration. In this way termination of the algorithm is advanced, when no better schedules are obtained during the search, since then the value of $MaxIt$ decreases.

6.1.2 The results of TSI

The algorithm is applied to the benchmark that is introduced by Vaessens [1995]. This benchmark contains the *hard-to-solve* Lawrence-instances [1984] —and two of the easier ones— and the famous 10×10 instance ft10 of Fisher & Thompson [1963]. The algorithm is executed on an Intel® Pentium™ 90 processor.

Pro-blem	Makespan				CPU[sec.]				Devia-tion[%][e]
	Lower bound[a]	Found by TSI[b]			Termination[c]		Up to makespan[d]		
		Best	Average	Worst	Average	Worst	Average	Worst	
10 jobs, 5 machines									
la02	655	655	655	655	4.9	7	0.5	2	0.00
10 jobs, 10 machines									
ft10	930	930	934	949	35.1	57	16.0	39	0.00
la19	842	842	843	849	24.0	32	9.7	22	0.00
15 jobs, 10 machines									
la21	1046	1048	1053	1055	42.9	61	22.1	42	0.19
la24	935	939	944	944	45.1	61	19.9	26	0.43
la25	977	977	980	984	49.0	79	29.6	65	0.00
15 jobs, 15 machines									
la36	1268	1268	1277	1278	49.6	83	18.6	46	0.00
la37	1397	1403	1414	1422	85.6	144	41.5	76	0.43
la38	1196	1196	1206	1215	96.6	161	46.1	96	0.00
la39	1233	1233	1238	1243	94.0	117	48.5	75	0.00
la40	1222	1228	1234	1299	62.7	101	22.0	78	0.49
20 jobs, 10 machines									
la27	1235	1241	1255	1270	51.2	75	23.7	42	0.40
la29	(1142)	1163	1175	1204	64.3	101	32.3	43	1.75

[a]The lower bound of the length of the schedule. For la29 this lower bound is smaller than its upper bound, which is 1153; for all the other instances this lower bound is the optimum.
[b]The solutions found by TSI: the best solution, the average solution and the worst solution, all over 10 runs.
[c]The average and the worst termination time of the algorithm over 10 runs.
[d]The average and worst time to find the best makespan, over 10 runs.
[e]The deviation of the best found makespan with respect to the lower bound. The average deviation to these 13 lower bounds is 0.32%.

Table 6.1: The results of TSI.

Since the algorithm starts from a randomly constructed schedule, every instance in the benchmark is solved ten times, in order to get a clearer image of the algorithm's performance. Note that in all cases the average time needed to find a sub-optimal schedule is significantly smaller (2.2 times) than the average time needed for termination.

6.1.3 Comparing TSI with other algorithms

In this section the results of TSI are compared with three other algorithms. These algorithms are a tabu search algorithm by Nowicki & Smutnicki [1995] (TSAB), a variable-depth algorithm by Balas & Vazacopoulos [1994] (GLS), and a simulated annealing algorithm by Aarts, Van Laarhoven, Lenstra & Ulder [1992] (SA1). For each of the algorithms, a superscript b followed by a number x, indicates that the reported makespan is the best one found after x runs; the computation time is the total time over all runs. A superscript m followed by a number x, indicates that the makespan is the mean over x runs; the computation time is the average over these runs.

Pro-	Makespan					CPU[sec.]				
blem	OPT^a	TSAB^{b1}	GLS^{b4}	SA1^{m5}	TSI^{b10}	TSAB^b	GLS^c	SA1^d	TSI^e	TSI^f
10 jobs, 5 machines										
la02	655	655	655	669	655	8	33	19	49	7
10 jobs, 10 machines										
ft10	930	930	930	969	930	30	153	99	351	57
la19	842	842	842	855	842	60	134	94	240	32
15 jobs, 10 machines										
la21	1046	1055	1047	1083	1048	21	222	243	429	61
la24	935	948	938	962	939	184	243	235	451	61
la25	977	988	982	1003	977	155	330	255	490	79
15 jobs, 15 machines										
la36	1268	1275	1269	1307	1268	623	455	602	496	83
la37	1397	1422	1400	1440	1403	443	268	636	856	144
la38	1196	1209	1208	1235	1196	165	464	636	966	161
la39	1233	1235	1233	1258	1233	325	577	592	940	117
la40	1222	1234	1233	1256	1228	322	367	597	627	101
20 jobs, 10 machines										
la27	1235	1259	1236	1282	1241	66	435	492	512	75
la29	(1153)	1164	1157	1233	1163	493	627	471	643	101

[a] The best known upper bound.
[b] The CPU-time on an Intel® 386 DX.
[c] The CPU-time on a Sparc 330.
[d] The CPU-time on a VAX 8650.
[e] The CPU-time on an Intel® Pentium™ 90.
[f] The worst computation time over 10 runs.

Table 6.2: Comparison of TSI with other algorithms.

The average deviation of TSAB, GLS and SA1 with respect to the lower bound is 0.97%, 0.37% and 3,35% respectively.

It is hard to form a clear image based on the comparison of these results, because of some great differences. First of all, all algorithms are tested on different hardware, performing a different number of runs. Secondly, all algorithms start from solutions with different quality. TSI, for example, starts with a randomly (as in no heuristic) generated solution, where TSAB starts from a very good initial solution, constructed with an insertion algorithm (INSA) by Werner & Winkler [1992]. The displayed time for TSAB, however, does not include the computation time to construct this initial solution. Since the time complexity of INSA is $\mathcal{O}(N^3 M^2)$ (N operations, M machines), the presented time is —to put it mildly— *somewhat* misleading.

The last column in Table 6.2 displays the worst computation time over ten runs. This time is presented, since it is the computation time of the parallel multiple-walk algorithm of Section 5.2.

The sum of the computation time equals 7050 seconds, for ten runs. The computer independent sum of computation times according to Vaessens [1995] is 28571 second.

6.2 Results of the parallel algorithm with multiple independent walks

This section presents the results from the multiple independent walk algorithm of Section 5.2. When P runs are performed simultaneously, the quality of the obtained solution is equal to that of P executions of the sequential algorithm —when using the same parameter settings. Therefore, only the speed-up results are presented. The computation time to find a sub-optimal solution for the instances of the benchmark of Vaessens [1995], is presented in the last column of Table 6.2.

The algorithm is is applied to the larger Taillard-instances [1993] for 1, 5, 10, 20 and 40 processors (plus one master). As stated before, a poor speed-up is obtained when the objective is find an optimal solution. Therefore, the presented computation times in Table 6.3, are the computation times needed to find a sub-optimal solution, with a deviation of 1% to the best known upper bound. Since a lot of these Taillard-instances are hard to solve, an extreme parameter setting is used, in order to increase the possibility that the algorithm obtains this sub-optimal solution, on a small number of processors. For all the instances with problem size 30×20 and for instance ta33, a deviation of 2% is used, since these particular instances are very hard to solve. The upper bound for the number of iterations is 20000, and a value of $4000\times r$ is subtracted when a neighbour of a solution is used as the initial solution for the next iteration (see assignment 6.1 on page 32). Furthermore, randomization of the tabu tenure is done in the same way as for the sequential algorithm, but a random value smaller than five instead of three is added. This is done in order to achieve more randomization. The algorithm is executed on a network containing mostly Sparc ELC's and Sparc 5's, and some other machines; furthermore, it is a public network. It is not justifiable to judge the algorithm's performance, when performing only a single run, due to the following aspects.

- The algorithm starts from a randomly constructed solution, and it employs randomization of the tabu tenure.
- The machines in the network differ in clock speed and in amount of memory.
- Other people can execute tasks on any machine in the network.

The first aspect has great influence on the quality of the found solution while the latter two influence

P	1		5		10		20		40	
$J\times M$	T	s	T	s	T	s	T	s	T	s
30×15[a]	485.7	1.0	348.5	1.4	323.6	1.5	191.0	2.5	161.4	3.0
30×20[b]	557.9	1.0	406.4	1.4	288.4	1.9	214.0	2.6	173.1	3.2
50×15[c]	201.7	1.0	38.7	5.2	17.9	11.2	17.5	11.5	21.1	9.6
50×20[d]	416.3	1.0	252.7	1.6	201.0	2.1	149.5	2.8	129.6	3.2
100×20[e]	324.1	1.0	91.4	3.5	71.8	4.5	68.8	4.7	67.2	4.8

[a] instances: ta31, ta32, ta33, ta34 and ta35.
[b] instances: ta41, ta42, ta43, ta44 and ta45.
[c] instances: ta51, ta52, ta53, ta54 and ta55.
[d] instances: ta61, ta62, ta63, ta64 and ta65.
[e] instances: ta76, ta77, ta78, ta79 and ta80.

Table 6.3: The speed-up results for the multiple independent walk algorithm.

the amount of computation time to find this solution.

In order to get a more representative image, the following is done. For each number of processors, P, a *virtual machine* with P processors is randomly constructed. A virtual machine with P processors can be seen as a *multi processor system* with P processors, each of them connected with all the other processors. For each problem size, five instances are tested and for each of these instances w runs are performed, such that w is the smallest number for which either five out of these w runs result in a sub-optimal solution, or for which w equals ten.

Termination of the algorithm with this extreme parameter setting takes a long time, sometimes up to 20 minutes. Therefore, the algorithm is terminated if a sub-optimal solution is not found within ten minutes, under the assumption that one will never find one. For each pair (instance and number of processors) the average computation time is determined. Each entry in Table 6.3 is the sum over the five averages of the instances with corresponding problem size, divided by five.

For problem size 50×15 and P equals 5 and 10, a speed-up is achieved, for which the efficiency exceeds 1. This is due to the fact that, in some cases, the algorithm does not find a sub-optimal solution on one processor, resulting in large computation times. For more processors the algorithm is able to find more sub-optimal solutions, resulting in smaller computation times. The performed tests reveal that, on the average, the algorithm is able to find more sub-optimal solutions when being executed on more processors, in fewer time. Nevertheless, the speed-up results are somewhat disappointing.

The next section presents the speed-up results of the multiple interacting algorithm.

6.3 Results of the parallel algorithm with multiple interacting walks

This section presents the results of the multiple interacting walk algorithm of Section 5.3. the first section presents the speed-up results of the algorithm, and the last section presents the results obtained by applying the algorithm to the benchmark of Vaessens [1995] and to some of the Taillard- and Storer-instances, [1992].

6.3.1 The speed-up results

This section presents the speed-up results of the multiple interacting walk algorithm. The parameter setting that is used is the same as the setting for the multiple independent walk algorithm. Due to the variety of machines in the network, it is not wise to synchronize every fixed number of iterations of the algorithm, because of the introduction of idle time on faster machines. Therefore, the interval between two subsequent synchronisations is a fixed number of seconds. For the test this number is five, which corresponds with 1000 iterations on a Sparc 5 and 500 iterations on a Sparc ELC, for a 100×20 instance. The function *Strip* is implemented in the following way. A real variable $Epsilon$ is introduced, with initial value 15. When the master receives all solutions from all slaves, he determines the solution with the best cost, s_{best}. The set S_{new} is constructed in the way such that it contains all solutions s, for which

$$\frac{f(s) - f(s_{best})}{f(s_{best})} \times 100 < Epsilon,$$

i.e. all solutions with less than $Epsilon\%$ deviation to s_{best}. After the partitioning and broadcasting of S_{new} the value of $Epsilon$ is divided by 1.5 if and only if it is at least 0.1. In this way function *Strip* gets stricter with every synchronisation.

Like in the previous section, a virtual machine is constructed completely random. For every instance, w runs are performed, such that w is the smallest number for which either eight out of these w

P	1		5		10		20		40	
$J \times M$	T	s	T	s	T	s	T	s	T	s
30×15^a	336.1	1.0	264.4	1.3	225.35	1.5	193.5	1.7	200.9	1.7
30×20^b	510	1.0	434.8	1.2	371.5	1.4	305.7	1.7	207.3	2.5
50×15^c	151.3	1.0	47.1	3.2	26.9	5.6	29.6	5.1	28.5	5.3
50×20^d	466.2	1.0	295.7	1.6	207.3	2.2	168.2	2.8	157.3	3.0
100×20^e	257.1	1.0	108.3	2.4	95.1	2.7	92.0	2.8	96.4	2.7

[a] instances: ta31, ta32, ta33, ta34 and ta35.
[b] instances: ta41, ta42, ta43, ta44 and ta45.
[c] instances: ta51, ta52, ta53, ta54 and ta55.
[d] instances: ta61, ta62, ta63, ta64 and ta65.
[e] instances: ta76, ta77, ta78, ta79 and ta80.

Table 6.4: The speed-up results for the multiple interacting walk algorithm.

runs result in a sub-optimal solution, or for which w equals sixteen. Since the algorithm is executed on a public network, not just the machines, but also the connections between those machines differ in speed. This is yet another disturbing factor, and therefore more runs are performed than with the multiple independent walk algorithm. For each pair (instance and number of processors) the average computation time is determined by leaving out the best and worst computation times of these w runs, and taking the average over the resulting $w-2$ runs. The results are presented in Table 6.4. Each entry in this table is the sum over the five averages of the instances with corresponding problem size, divided by five.

In contradiction with the expectation, the speed-up results are very disappointing. In fact, in most cases they are worse than the results of the multiple independent walk algorithm. Some possible reasons for this are the following.

- The time interval between two subsequent synchronisations is too small. Because of this, *lazy* slaves throw away solutions that can turn out to be well in the future.
- The computation time on one processor is better for the interacting algorithm than for the independent algorithm. This makes it harder to obtain a good speed-up.
- The value of $Epsilon$ declines too fast, making the behaviour of function $Strip$ too strict.
- The lower bound for $Epsilon$ is too small ($\frac{1}{10}\%$).

The reason for the poor speed-up is probably a combination of all these, and perhaps even some others.

6.3.2 Some quality results

This section presents the results of the parallel tabu search algorithm with intensification (PTSI). Just like the sequential algorithm, the algorithm is applied to the Vaessens-benchmark. Furthermore, some results are presented of the algorithm's capability for the Taillard- and the Storer, Wu & Vaccari-instances ([1993, 1992]).

The parameter setting for PTSI is as follows. The maximum number of arc reversals is 6000. A value of $1200\times r$ is subtracted when a neighbour of a solution is loaded from the stack. A maximum of five schedules is stored for intensification, and the time interval between two synchronisations is five seconds. $Epsilon$ has an initial value of 7.5, and while it is at least 0.5, it is divided by 1.5. The algorithm is executed on 31 processors: one master and 30 slaves.

If PTSI does not find a global minimum, a different parameter setting is used. This setting is: a maximum of 6500 arc reversals, a subtraction of $577\times r$, ten schedules stored for intensification and a

Pro-blem	Makespan					CPU[sec.]			
	OPT[a]	RGLS-5[b1]	TSI[b10]	PTSI[b1]	PTSI′[b1]	RGLS-5[b]	TSI[c]	PTSI[d]	PTSI′[e]
10 jobs, 5 machines									
la02	655	655	655	655		8	49	34	
10 jobs, 10 machines									
ft10	930	930	930	930		247	351	97	
la19	842	842	842	842		269	240	78	
15 jobs, 10 machines									
la21	1046	1046	1048	1046		612	429	217	
la24	935	935	939	938	938	682	451	125	169
la25	977	977	977	977		616	490	117	
15 jobs, 15 machines									
la36	1268	1268	1268	1268		920	496	185	
la37	1397	1397	1403	1397		822	856	236	
la38	1196	1196	1196	1196		1281	966	303	
la39	1233	1233	1233	1233		1131	940	210	
la40	1222	1224	1228	1226	1224	1590	627	237	234
20 jobs, 10 machines									
la27	1235	1235	1241	1235		315	512	130	
la29	(1153)	1164	1163	1163	1159	1062	643	203	267

[a] The best known upper bound.
[b] The CPU-time on a Sparc 330.
[c] The CPU-time on an Intel® Pentium™ 90.
[d] The CPU-time on a network containing 22 Sparc ELC's, 7 Sparc 5's and 3 Sparc Classic's.
[e] The CPU-time on a network containing 22 Sparc ELC's, 7 Sparc 5's and 3 Sparc Classic's, with a different parameter setting.

Table 6.5: The results of PTSI.

time interval of ten seconds in between synchronisations. This algorithm is denoted by PSTSI′. The results of the algorithm are presented in Table 6.5. This table also contains the results of TSI and of RGLS-5, a guided local search algorithm by Balas & Vazacopoulos [1994].

The best results reported so far have been obtained by RGLS-5; a single run achieves a mean deviation to the lower bounds of 0.170%. PTSI achieves a mean deviation of 0.197% in a single run in a total of 2172 seconds. The sum of computer independent computation times is hard to give, because of the variety of machines in the network. Since most of these machines are Sparc ELC's, this sum is approximately 4100 seconds; seven times as fast and twice as good as TSI (performing ten runs, but finding worse solutions)! For two of the three instances for which PTSI can not obtain an optimal solution, PTSI′ finds a better solution. When assuming that PTSI′ is able to find an optimal solutions for the instances for which PTSI can, PTSI′ has a mean deviation of 0.155% for these 13 instances in single run.

For the Taillard- and Storer, Wu & Vaccari-instances it is hard to obtain solutions that are better than the best known solutions yet. This because of the personal computations by Balas & Vazacopoulos [1994], who invested a lot of time in solving these instances. PTSI is able to find near optimal solutions for all instances within 15 to 20 minutes. Note that this is the termination time, not the time to find this solution. For ten instances PTSI found a better upper bound than found so far. For six of these instances this is the best known upper bound[1] and for four instances, the shuffle algorithm of Applegate & Cook [1991] was able to obtain an even better solution, starting from the solutions obtained by PTSI[2].

In contradiction with the poor speed-up results of both the independent and the interacting algorithm, the quality results of PTSI, for this benchmark, are overwhelming. The mean deviation of 0.155% to this benchmark is the best result obtained by any algorithm in a single run, and the fact that, at this moment, PTSI found the best solutions ever for ten instances —to which several other very good algorithms were applied— is the icing on the cake.

[1] Instance with best known upper bound between parentheses: ta16 (1368), ta21 (1647), ta24 (1651), ta29 (1625), ta50 (1951) and swv13 (3146).

[2] ta15 (1342), ta30 (1596), swv08 (1770) and swv15 (2940), personal computations by Rob Vaessens using the shuffle algorithm of Applegate & Cook.

Bibliography

AARTS, E.H.L., AND J. KORST [1989], *Simulated Annealing and Boltzmann Machines*, Wiley, Chichester.

AARTS, E.H.L., P.J.M. VAN LAARHOVEN, J.K. LENSTRA, AND N.J.L. ULDER [1992], *Job Shop Scheduling by Local Search*, Memorandum COSOR 92-29, Eindhoven University of Technology.

AARTS, E.H.L., AND J.K. LENSTRA (eds.) [1996], *Local Search in Combinatorial Optimization*, Wiley, New York.

ADAMS, J., E. BALAS, AND D. ZAWACK [1988], The shifting bottleneck procedure for job shop scheduling, *Management Science* **34**, 391–401.

APPLEGATE, D., AND W. COOK [1991], A computational study of the job-shop scheduling problem, *ORSA Journal on Computing* **3**, 149–156.

BALAS, E., AND A. VAZACOPOULOS [1994], *Guided Local Search with Shifting Bottleneck fo Job Shop Scheduling*, Management science research report #MSRR-609, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, Pennsylvania.

FISHER, H., AND G.L. THOMPSON [1963], Probabilistic learning combinations of local job-shop scheduling rules, in: J.F. Muth and G.L. Thompson (eds.), *Industrial Scheduling*, Prentice Hall, Englewood Cliffs.

GLOVER, F. [1989], Tabu search – part I, *ORSA Journal on Computing* **1**, 190–206.

GLOVER, F., E. TAILLARD, AND D. DE WERRA [1993], A user's guide to tabu search, *Annals of Operations Research* **41**, 3–28.

GOLDBERG, D.E. [1989], *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Reading.

JOHNSON, D.S. [1990], Local optimization and the traveling salesman problem, *Proc. 17th Colloquium on Automata, Languages, and Programming*, LNCS 443, Springer-Verlag, 446–461.

KIRKPATRICK, S., C.D. GELATT, JR., AND M.P. VECCHI [1983], Optimization by simulated annealing, *Science* **220**, 671–680.

LAARHOVEN, P.J.M. VAN, E.H.L. AARTS, AND J.K. LENSTRA [1992], Job shop scheduling by simulated annealing, *Operations Research* **40**, 113–125.

LAWLER, E.L. [1976], *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart & Wilson, New York.

LAWLER, E.L., J.K. LENSTRA, A.H.G. RINNOOY KAN, AND D.B. SHMOYS [1993], Sequencing and scheduling: Algorithms and complexity, in: S.C. Graves, A.H.G. Rinnooy Kan, and P. Zipkin (eds.), *Handbooks in Operations Research and Management Science 4*, North-Holland.

LAWRENCE, S. [1984], *Resource Constrained Project Scheduling: An Experimental Investigation of Heuristic Scheduling Techniques (Supplement)*, Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh.

MARTIN, O., S.W. OTTO, AND E.W. FELTEN [1991], Large-steps markov chains for the travelling salesman problem, *Complex Systems* **5**, 299–326.

MATSUO, H., C.J. SUH, AND R.S. SULLIVAN [1988], *A controlled search simulated annealling method for the general job shop scheduling problem*, Working Paper 03-04-88, Graduate School of Business, The University of Texas at Austin.

MÜHLENBEIN, H., M. GORGES-SCHLEUTER, AND O. KRÄMER [1988], Evolution algorithms in combinatorial optimization, *Parallel Computing* **7**, 65–85.

NOWICKI, E., AND C. SMUTNICKI [1995], A fast taboo search algorithm for the job shop problem, *Management Science*, to appear.

PAPADIMITRIOU, C.H., AND K. STEIGLITZ [1982], *Combinatorial Optimization: Algorithms and Complexity*, Prentice Hall, Englewood Cliffs.

REEVES, C.R. (ed.) [1993], *Modern Heuristic Techniques for Combinatorial Problems*, Blackwell, London.

ROY, B., AND B. SUSSMANN [1964], *Les problèmes d'ordonnancement avec constraints disjonctives*, Note DS 9 bis, SEMA, Paris, France.

STORER, R.H., S.D. WU, AND R. VACCARI [1992], New search spaces for sequencing problems with apllication to job shop scheduling, *Management Science* **38**, 1495–1509.

TAILLARD, E. [1993], Benchmarks for basic scheduling problems, *European Journal of Operation Research* **64**, 278–285.

TAILLARD, E. [1994], Parallel taboo search techniques for the job shop scheduling problem, *ORSA Journal on Computing* **6**, 108–117.

VAESSENS, R.J.M. [1995], *Generalized Job Shop Scheduling: Complexity and Local Search*, Ph.D. thesis, Eindhoven University.

VAESSENS, R.J.M., E.H.L. AARTS, AND J.K. LENSTRA [1994], *Job shop scheduling by local search*, Memorandum COSOR 94-5, Eindhoven University of Technology, Netherlands.

VERHOEVEN, M.G.A. [1996], *Parallel Local Search*, Ph.d. thesis, Eindhoven University, Netherlands.

VERHOEVEN, M.G.A., AND E.H.L. AARTS [1995], *Parallel Local Search and Job Shop Scheduling*, Technical report, Netherlands.

WERNER, F., AND A. WINKLER [1992], *Insertion Techniques for the Heuristic Solution of the Job Shop Problem*, Technical report, Technische Universität "Otto von Guericke", Magdenburg, Germany.

YANNAKAKIS, M. [1990], The analysis of local search problems and their heuristics, *Proc. 7th Symposium on Theoretical Aspects of Computer Science*, LNCS 415, Springer-Verlag, Berlin, 298–311.