Commonalities in local search

Commonalities in local search

R.M.F. Schilham

Commonalities in local search

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof.dr. M. Rem, voor een commissie aangewezen door het College voor Promoties in het openbaar te verdedigen op vrijdag 26 januari 2001 om 16.00 uur

door

Robin Marco Frank Schilham

geboren te Amersfoort

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr. J.K. Lenstra
en
prof.dr. E.L.H. Aarts

Copromotor: dr.ir. H.M.M. ten Eikelder

Contents

Chapter 1

Introduction and motivation

1.1 Brief summary

In this thesis we are concerned with the use of local search heuristics in combinatorial optimization. The motivation of the research is given by the common defect of many local search heuristics that they tend to get stuck in an unpromising part of the search space. We propose a restart mechanism for local search heuristics that attempts to overcome this deficiency.

1.2 Combinatorial optimization

Combinatorial optimization is the discipline that is concerned with finding the best solution to a problem that has a finite or countably infinite set of alternative solutions. These problems arise in situations where discrete decision variables are involved, such as in planning and design.

A *combinatorial optimization problem* is specified by a set of problem instances and is either a *minimization* problem or a *maximization* problem. Each solution has a quantitative quality measure, which is generally referred to as its *cost*. By reversing the sign of the solution cost, a maximization problem is transformed into a minimization problem. Hence, without loss of generality, we will restrict ourselves to minimization problems.

An *instance* of a combinatorial optimization problem is a pair $(\mathcal{S}, f)$, where $\mathcal{S}$ is the *solution set* and f is the *cost function* $f : \mathcal{S} \to \mathbf{R}$. The problem is to find an *optimum* solution, i.e., a solution $s \in \mathcal{S}$ such that $f(s) \leq f(s')$ for all $s' \in \mathcal{S}$. Usually, the solution set and the cost function are given implicitly by means of an algorithm with parameters, rather than explicitly (see Papadimitriou and Steiglitz [54]). For a given problem type,

we have two polynomial algorithms. One of the algorithms tests whether or not its input is a solution. The other algorithm calculates the cost of a solution.

Many combinatorial optimization problems are NP-hard, see Garey and Johnson [24]. It is generally believed that NP-hard problems cannot be solved to optimality within an amount of time that is polynomially bounded in the size of the problem instance. From a computational point of view, obtaining an optimal solution may require an excessive amount of time. Nevertheless, remarkable successes were booked recently in finding optimum solutions to large problems instances; see for instance Applegate, Bixby, Chvátal and Cook [3]. On the other hand, there is much interest in finding near-optimal solutions in an amount of time that is practically feasible. Roughly speaking, there are two options for obtaining near-optimal solutions to NP-hard problems: constructive methods and local search methods. Constructive methods build a single solution, while local search methods start with a given solution and repeatedly try to improve it. In certain cases, constructive methods are known to produce, in polynomial time, solutions with a cost that is within a prespecified bound from the unknown optimum cost. Local search heuristics, in general, have no such performance guarantees, neither with respect to the cost nor with respect to the running time. Nevertheless, for many combinatorial optimization problems, local search heuristics are known to produce excellent solutions within a reasonable amount of time; see for instance Aarts & Lenstra [2].

1.3 Local search heuristics

Local search heuristics date back to the 1950's, when Bock [8] and Croes [18] proposed link exchange algorithms for the traveling salesman problem. Later, the same ideas where applied to a variety of other problems, such as scheduling (Page [53], Nicholson [50]) and graph partitioning (Kernighan & Lin [37]). After a relatively quiet period, the recent increase in computational resources has contributed to a revival of the subject.

Starting from a given initial solution, a local search heuristic repeatedly tries to find a better solution by modifying the current solution slightly, until a stop criterion is met. Applying modifications to the current solution is called a *move* or a *step*. Moving from one solution to the next is sometimes called an *iteration*. The sequence of successive solutions that are visited by a local search heuristic is called the *path* or the *trajectory*.

The neighborhood function describes what kind of modifications to the

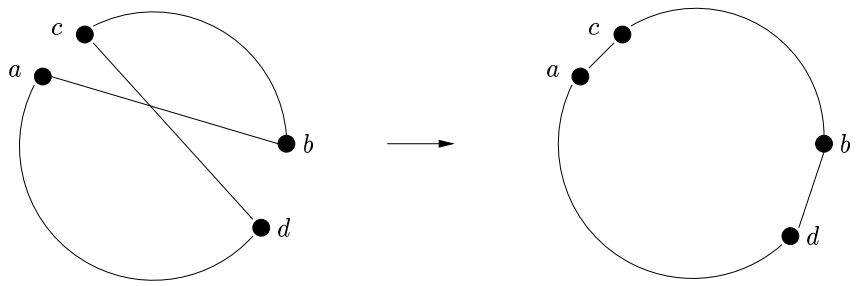

Figure 1.1: A two-exchange move for the traveling salesman problem.

current solution are allowed. Let $(\mathcal{S}, f)$ be an instance of a combinatorial optimization problem. A *neighborhood function* is a mapping $\mathcal{N} : \mathcal{S} \to 2^{\mathcal{S}}$ that defines for each solution $s \in \mathcal{S}$ a set of solutions that are in a sense close to s. The set $\mathcal{N}(s)$ is the *neighborhood* of solution s, and each $s' \in \mathcal{N}(s)$ is called a *neighbor* of s. Because $\mathcal{N}$ imposes a topology upon $\mathcal{S}$, the pair $(\mathcal{S}, \mathcal{N})$ is often referred to as the *search space*.

Traveling salesman. In the *traveling salesman problem* (TSP) we are given a finite set of cities and for each pair of distinct cities, the cost of travel between them. The set of solutions consists of all tours that pass through all the cities exactly once and return to the point of departure. The cost of a tour is equal to the sum of the cost of the edges. We want to find a cheapest tour. A tour is often represented by a graph. The cities correspond to the vertices of the graph and for every pair of subsequent cities in the tour, there is an edge between the corresponding vertices. The *two-exchange neighborhood* replaces two edges in the current solution by two others in such a way that another tour is obtained, see Figure 1.1.

Let $(\mathcal{S}, f)$ be an instance of a combinatorial optimization problem and let $\mathcal{N}$ be a neighborhood function. A solution $s \in S$ is called *locally optimal* with respect to $\mathcal{N}$ when $f(s) \leq f(s')$, for all $s' \in \mathcal{N}(s)$.

Besides the neighborhood function, there are two other ingredients that constitute a local search heuristic: a *search strategy* and a *stop criterion*. The search strategy determines which neighbor of the current solution is chosen as the successor. Common search strategies include *best improvement*, which selects the best neighbor, and *first improvement*, which selects the first neighbor that is better than the current solution while searching the neighborhood of the current solution. Often, the decision which search strategy to employ depends on the neighborhood function. In general, small neighborhoods can be searched faster than big neighborhoods, but big neighborhoods are more likely to contain an improving neighbor. A well-chosen

combination of neighborhood size and search strategy strikes a balance between speed and effectiveness.

Stop criteria are usually expressed in terms of the length of the search process or in terms of the quality of the solutions that are found. Common stop criteria include a bound on the running time, a maximum number of iterations, a maximum number of subsequent non-improving iterations, the current solution being a local optimum, and the current solution having a cost that is lower than a given threshold.

Iterative improvement. Starting from a given solution, an iterative improvement algorithm repeatedly selects a neighbor as long as it improves the current solution. Consequently, the algorithm terminates when the current solution is a local optimum. Iterative improvement for the TSP with the 2-exchange neighborhood is known as 2-opt (Lin [41]).

Another very important issue for successfully applying a local search heuristic is the initial solution. A common way to obtain an initial solution is by means of a construction heuristic, which is either randomized or deterministic. The construction of the initial solution is not part of the local search heuristic itself. Nevertheless, it can have a great impact on the performance of the local search heuristic. The construction of good initial solutions is a major topic of the thesis, as will become clear later on.

1.3.1 Multiple independent runs

The main drawback of iterative improvement is that it gets trapped in the first local optimum that is encountered, which can be very bad. A common way to improve iterative improvement is by performing multiple runs, each started with a different solution, and taking the best result. The runs are independent of each other in the sense that there is no common memory in which they share information about the search process. Obviously, the same technique can also be applied to other local search heuristics that tend to get trapped in poor-quality local optima. When the local search heuristic under consideration is randomized, different results can be obtained even when the heuristic is started with the same solution. In that case, the initial solutions need not be different for each run.

Multiple independent runs are capable of improving the performance of many local search heuristics for a variety of combinatorial optimization problems. Because of the simplicity, multiple independent runs are frequently used in practice, and they often produce better results than a single run of a local search heuristic. On the down side, each run is started from scratch, wasting all efforts of preceding runs.

1.3.2 Escaping from local optima

Ever since the introduction of local search heuristics, a large variety of methods has been proposed to reduce the chance of getting trapped in a poor local optimum. Basically, there are two options. Either one can change the topology of the search space or one can change the search strategy. The topology of the search space can be changed by using a different neighborhood function. Alternatively, *multi-level* local search heuristics use two or more different neighborhood functions and frequently switch from one neighborhood function to another. In *variable depth search*, a relatively simple neighborhood function is turned into a more complex one by letting a single move consist of multiple simple moves. Below, we discuss two popular local search heuristics that have a built-in mechanism to avoid getting trapped in poor local optima, namely *simulated annealing* and *tabu search*. In both heuristics, it is the search strategy that reduces the chance of getting trapped in a poor local optimum.

Simulated annealing. Simulated annealing was introduced by Kirkpatrick, Gelatt and Vecchi [38] and Černý [14]. Each iteration of simulated annealing consists of two parts. First, a neighbor of the current solution is *selected* uniformly at random. Next, it is determined whether or not the neighbor is *accepted* in the sense that it becomes the successor of the current solution. When the neighbor is better than the current solution, it is accepted. When it is worse than the current solution, it is accepted with a certain probability that is reversely related to the cost difference with the current solution. At the ith iteration, the acceptance probability is given by $\exp(-(f(s') - f(s))/t_i)$, where s denotes the current solution, s' is the selected neighbor, and t_i is the *control parameter* at iteration i. The control parameter is gradually decreased such that the probability of accepting a deteriorating move decreases over time.

Tabu search. Tabu search was proposed by Glover [27]. See also Glover [28, 29] and De Werra and Hertz [73]. Tabu search repeatedly selects the best neighbor, even when it is worse than the current solution. The set of neighbors is restricted by a so-called *tabu list*. After effectuating a move, the reverse move is added to the tabu list, indicating that it is forbidden for the next t iterations, where t is called the *tabu tenure*. In this way, tabu search prevents visiting solutions that were recently seen.

1.4 Pathology of local search heuristics

A highly desirable property of local search heuristics is the existence of a trade-off between computational effort and the quality of the solutions that are found. Unfortunately, even some of the most successful heuristics seem to lack this property. Apparently, there are other pitfalls besides the danger of getting trapped in a poor local optimum. In this section we discuss some of the weaknesses that are common to many local search heuristics. These weaknesses are generally referred to as *getting stuck*.

The issue of getting stuck is best illustrated with an example. Chained Lin-Kernighan (CLK) [47] is a well known local search heuristic for the TSP. We did five runs of CLK on the TSPLIB problem instance pla7397 with 7,397 cities (Reinelt [56]), each started in a different solution, and calculated the relative excess permillage above the optimum as a function of the iteration number. The results are given in Figure 1.2. The trajectories of these independent runs are improving very fast in the beginning of the search processes. Quite early in the search process, the trajectories start to diverge. Apparently, something can go wrong at an early stage that is not easily corrected later on. There often appear to be almost no improvements when CLK is given more time. None of the five runs was able to find an optimum tour. About half of the time was wasted, as no further improvements were found. We conclude that CLK is likely to get stuck in a near-optimal solution, and that the result of CLK is quite dependent of the initial solution.

We obtained similar results for other combinatorial optimization problems and local search heuristics. Starting from a given solution, a local search heuristic typically finds many big improvements in the beginning of the search. After a while, it takes longer to find an improvement, and the improvements are smaller. When no improvements are found for a long time, either we have found an optimum, or the heuristic got stuck.

There appears to be a generalized notion of getting stuck in a local optimum, namely getting stuck in a part of the search space that does not contain any further improvements. From now on, we adopt the following definition.

Definition 1.4.1. *A local search heuristic is said to* get stuck *when it traverses a relatively long path without any significant improvements.*

Although this definition is rather vague, is does capture the essence of the common deficiency of many local search heuristics.

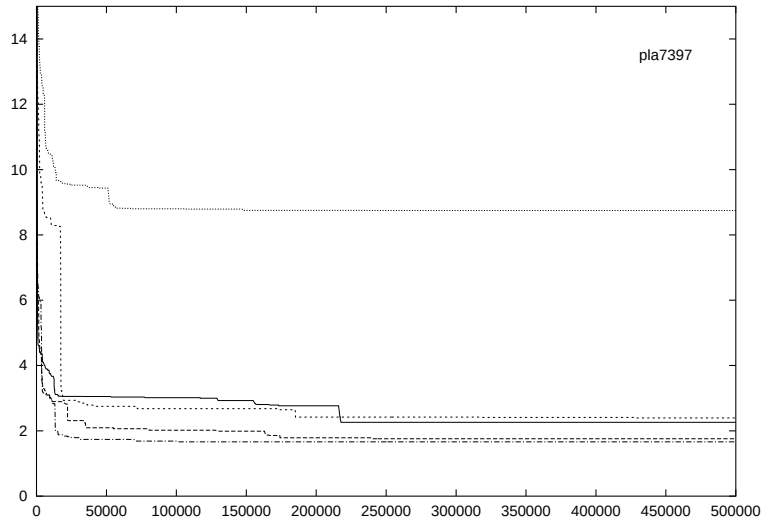

Figure 1.2: The trajectories of five runs of CLK for TSPLIB instance pla7397. The solution quality on the y-axis is given in per mille above the optimum as a function of the iteration number on the x-axis.

convergence

Some heuristics are known to *converge* to an optimum, when given enough time: an optimum is found (with probability 1) as the number of iterations approaches infinity. Unfortunately, these heuristics theoretically require an infinite amount of processing time before actually finding an optimum. Besides that, we do not know the optimum cost of the problem instance in advance, so we cannot tell whether the heuristic has found an optimum or not. As a result, we do not know when to stop.

For many of the most successful heuristics found to date, no convergence results are known. The known convergence results are mainly of theoretical interest and not of much practical value.

Notice that the notion of getting stuck, as we use it, is not the opposite of convergence: even heuristics that are known to converge can get stuck in an unpromising part of the search space, wasting a lot of time.

1.4.1 Cost landscape

To make the behavior of local search heuristics intuitively clear, we use the following metaphor. Consider the problem of finding the lowest point on

the surface of the Earth. The solution set is the set of integer grid points, which are denoted by pairs (x, y) where x specifies the longitude and y the latitude. The cost of a solution is its height with respect to the average sea level. The neighborhood of a point (x, y) is defined as the set of points that can be obtained by increasing or decreasing x or y by one.

Although many local search problems are not as easy to visualize as the Earth metaphor, much of the local search terminology seems to be derived from it. For example, many combinatorial optimization problems have binary decision variables, and in general more than two of them; nevertheless, the triple $(\mathcal{S}, \mathcal{N}, f)$ is often referred to as the *cost landscape*. In what follows, we use the Earth metaphor to clarify some of the local search terminology that is related to the notion of getting stuck.

barriers

Obviously, local search heuristics generally prefer improving moves to deteriorating moves. Consequently, there is a danger of getting stuck in an unpromising part of the search space. It is not hard to imagine that sometimes a number of unattractive moves have to be effectuated before any further improvement can be found. In the Earth example, a region enclosed by mountains is hard to escape from. The mountains form an almost impenetrable obstacle for local search heuristics. These obstacles are known as *barriers*. Many search strategies are unlikely to select a move that deteriorates the cost of the current solution, let alone a whole sequence of deteriorating (uphill) moves. As a result, for many local search heuristics it is unlikely to find any further improvements.

Many local search heuristics have a mechanism that allows uphill moves, notably simulated annealing and tabu search. But even then, only a limited number of subsequent uphill moves is allowed, because local search heuristics should not be too willing to effectuate deteriorating moves. As a result, it can sometimes be very hard to escape from unpromising parts of the search space due to the existence of barriers.

plateaux

A *plateau* is a part of the search space in which all solutions have approximately the same cost. In the Earth metaphor, Holland is an example of a plateau and because it is so flat, finding the lowest point is hard. On a plateau, a local search heuristic can get lost in the sense that there is no clear preference for one solution to the other. Furthermore, when the plateau is

surrounded by barriers, the presence of many cost-equivalent neighbors implies that the heuristic is not likely to choose a neighbor that is significantly worse than the current solution. Consequently, it can be very hard for a local search heuristic to escape from a plateau.

cycling

Neighbor selection rules can sometimes result in revisiting solutions over and over again, a phenomenon which is known as *cycling*. Obviously, cycling prevents that further improvements are found. Therefore, many local search heuristics try to avoid visiting solutions that were visited before. Simply remembering all visited solutions is not an option because of the excessive memory requirements and the complexity of comparing a neighbor with all these solutions. Tabu search uses a form of fingerprinting to identify solutions that were visited recently, while simulated annealing relies on randomization.

1.5 Commonality-preserving restarts

In almost every practical situation, we face the problem that we do not know whether our heuristic found an optimum or got stuck in an unpromising part of the search space with a non-optimal solution. At this point, we are faced with a dilemma: should we stop or continue? In general, we do not know the value of an optimal solution, so we do not know when to stop.

The most obvious remedy against getting stuck is performing multiple independent runs, each started in a different solution, and taking the best solution. Multiple independent runs have a major disadvantage: they ignore the knowledge that has been acquired by preceding runs. We present a restart mechanism that is more subtle in the sense that subsequent runs take advantage of the past by using information from previous runs.

Experiments indicate that solutions produced by multiple independent runs of a local search heuristic have many parts in common. Because the same parts are found over and over again, it is tempting to think of them as being "probably right". We operationalize this finding as follows. Instead of starting from scratch, a subsequent run of the local search heuristic is started in a solution that already contains these so-called *commonalities*. By perturbing the building elements of the best solution found so far in such a way that commonalities have a low chance of being replaced by others, we obtain an initial solution that preserves many commonalities. In this way,

the local search heuristic can immediately start exploring interesting regions of the search space.

The main objective of the thesis is finding a restart mechanism for existing local search heuristics that performs better than a single long run of the heuristic and better than multiple independent runs, given the same total processing time.

1.6 Thesis overview

In Chapter 2, we elaborate on the concept of commonalities. We give a definition, show the abundance of commonalities in good solutions, and sketch a general commonality-preserving restart mechanism.

The first application of commonalities is given in Chapter 3, which is based upon a paper by Schilham and Ten Eikelder [64]. There we improve the finite-time behavior of the Chained Lin-Kernighan heuristic for the symmetric traveling salesman problem by using a commonality-preserving restart mechanism. The commonalities are the edges that constitute the tours.

In Chapter 4, we describe the Whizzkids problem of 1996. Four newspaper boys must deliver newspapers to 120 subscribers in Manhattan. The problem is to cluster and route the addresses and produce a newspaper delivery plan such that the last newspaper is delivered as early as possible. The edges that connect subsequent addresses are used as commonalities. These edges capture the assignment part of the problem as well as the sequencing part. We describe a commonality-preserving restart mechanism for a tabu search heuristic that is based upon Osman's λ-interchange neighborhood and the 2-opt and Lin-Kernighan heuristics for tour improvement. Although the problem is still open, in the sense that there is a gap between the best known upper bound and the best known lower bound, we were able to find solutions with the best known delivery time for the last newspaper.

Chapter 5 gives some new upper bounds for a number of hard problem instances of the job shop scheduling problem. These upper bounds were obtained with a commonality-preserving restart mechanism for the tabu search heuristic of Nowicki and Smutnicki. For each pair of operations that are processed on the same machine, we must specify whether one comes before the other, or vice versa. These elementary precedences are used as our commonality concept. The chapter is based upon a paper by Schilham and Ten Eikelder [63].

The Whizzkids problem of 1997 is a generalization of the job shop

scheduling problem and is studied in Chapter 6. The problem is to find a schedule for a so-called parents' night, at which parents have conversations with the teachers of their kids to discuss their progress at school. For each parent, a partial order is given in which the teachers are to be visited. The objective is to finish the parents' night as early as possible. We adapted the tabu search heuristic of Nowicki and Smutnicki and implemented a commonality-preserving restart mechanism. This time, the commonalities are the pairwise orders in which the conversations take place, from the parents' point of view and from the teachers' point of view. The commonality-preserving restart mechanism is slightly better than multiple independent runs and slightly better than a single long run.

Chapter 7 demonstrates that commonalities may also be useful in finding lower bounds, which is illustrated in the context of the job shop scheduling problem. The commonalities of good solutions are employed to guide the shave algorithm of Martin and Shmoys. Sharper lower bounds were found for a number of open problem instances. The chapter is based upon a paper by Schilham and Ten Eikelder [65].

The final chapter, Chapter 8, contains some general conclusions.

Chapter 2

Commonalities

2.1 Introduction

In order to find solutions to a combinatorial optimization problem with help of a computer, we must decide on how to represent solutions. We discuss solution representations and define what commonalities are. The existence and abundance of commonalities is demonstrated by considering well known local search heuristics for the traveling salesman problem and the job shop scheduling problem. These two problems are typical and notorious combinatorial optimization problems. In the chapters to come, we give two applications of commonalities, thereby showing that commonalities are useful. First of all, we use them as the basis of a restart mechanism for local search heuristics. Furthermore, commonalities are employed to guide the shave algorithm of Martin and Shmoys [45], which derives lower bounds for job shop scheduling problems.

2.2 Building elements

Before we can implement a local search heuristic for a combinatorial optimization problem, we must decide on how to represent solutions. A single solution can have many equivalent representations. The representation can have a significant impact on the performance of the local search heuristic.

An instance of a combinatorial optimization problem can often be associated with a set of so-called *building elements*. The building elements are the elements that constitute solutions. Each solution corresponds to a subset of the set of building elements. Building elements are the parts that distinguish one solution from the other. Neighborhoods are often defined in

terms of exchange functions, which replace a number of building elements by some other building elements while preserving feasibility.

Traveling salesman. In the *traveling salesman problem* (TSP) we are given a finite set of cities and for each pair of distinct cities, the cost of travel between them. The set of solutions consists of all tours that pass through all the cities exactly once and return to the point of departure. The cost of a tour is equal to the sum of the cost of the edges. We want to find a cheapest tour. When the cost of travel between two cities is independent of the order in which they are visited, the problem is called a *symmetric* traveling salesman problem. A tour is often represented by a graph. The cities correspond to the vertices of the graph, and for every pair of subsequent cities in the tour, there is an edge between the corresponding vertices. The building elements are the edges that connect cities in a tour. A tour corresponds to a set of edges, but not every set of edges corresponds to a tour. The 2-exchange neighborhood replaces two edges by two other edges such that a tour is obtained.

Job shop scheduling. In the *job shop scheduling problem* (JSSP) we are given a set $\mathcal{J}$ of n jobs, a set $\mathcal{M}$ of m machines and a set $\mathcal{O}$ of N operations. For each operation $u \in \mathcal{O}$ there is a processing time $p_u \in \mathbf{Z}^+$, a unique machine $M_u \in \mathcal{M}$ on which it must be processed, and a unique job $J_u \in \mathcal{J}$ to which it belongs. Furthermore, a binary precedence relation $\prec$ is given that decomposes $\mathcal{O}$ into chains, one for each job. The problem is to find a start time s_u for every operation $u \in \mathcal{O}$ such that the *makespan*, defined as $C_{\max} = \max_{u \in \mathcal{O}}\{s_u + p_u\}$, is minimized subject to the following constraints:

$$
\begin{array}{ll}
s_u \geq 0 & \text{for all } u \in \mathcal{O}, \\
s_v \geq s_u + p_u & \text{for all } u, v \in \mathcal{O} \text{ with } u \prec v, \text{ and} \\
s_v \geq s_u + p_u \text{ or } s_u \geq s_v + p_v & \text{for all } u, v \in \mathcal{O} \text{ with } M_u = M_v.
\end{array}
$$

The first constraint implies that no machine is available before time 0, the second constraint accounts for the precedence relation $\prec$ such that no operation will start before its predecessor in the chain has finished, and the last constraint is the machine capacity constraint, which stipulates that every machine can process at most one operation at a time. For simplicity, we assume that every job visits a machine at most once. An instance of the job shop scheduling problem can be represented by a *disjunctive graph* (Roy & Sussmann [62]). This is a vertex-weighted mixed graph $G = (\mathcal{O}, A, E)$ where the arc set A consists of *job arcs* connecting consecutive operations of the same job, and the edge set E consists of edges connecting distinct operations that must be executed on the same machine. Each vertex $u \in \mathcal{O}$

has a weight p_u. Every solution σ to the job shop scheduling problem induces a unique acyclic digraph $G(\sigma)$ obtained from G by replacing every edge $\{u, v\} \in E$ by a *machine arc* (u, v) or (v, u). The earliest start time of operation u is equal to the length of a longest path in $G(\sigma)$ up to and excluding u. The cost of the solution, denoted by $c(\sigma)$, is equal to the length of a longest path in $G(\sigma)$. Our set of feasible solutions $\mathcal{S}$ is the set of acyclic digraphs that can be obtained from G. These acyclic digraphs correspond to the set of left-justified schedules. Most neighborhoods for the job shop scheduling problem are based on reversing a machine arc on a longest path in the digraph. Reversing an arc on a longest path always results in a feasible solution, and reversing an arc that is not on a longest path will never improve the solution and can even lead to infeasibility (Van Laarhoven, Aarts & Lenstra [39]). The machine arcs are the building elements.

Not all combinatorial optimization problems have a natural concept of building elements. For example, in the Earth metaphor, solutions are pairs of integers, which have no internal structure. There is no natural representation of numbers other than the numbers themselves. As a result, the Earth metaphor is not suitable for illustrating the concept of building elements.

2.3 Commonalities

Often, it is intuitively clear that certain building elements are unlikely to be part of a good solution. In a TSP instance, it is unlikely that optimum solutions have many long edges. Hence, good solutions will probably not contain many of these building elements. Similarly, for a given JSSP instance, suppose that the first operation of the first job and the last operation of the second job are to be processed on the same machine. Scheduling the second job before the first job on the machine will probably lead to a bad solution. All good solutions are likely to schedule the former job before the latter one.

Besides these relatively obvious rules of thumb, it turns out that good solutions also have a lot of building elements in common that are not so obvious. We are interested in the building elements that are common to many good solutions. We intend to characterize the typical structure of good solutions in terms of the building elements that they consist of.

Definition 2.3.1. *Let $\mathcal{L}$ denote a multiset of solutions. The* commonality set *of $\mathcal{L}$ is defined as the set of building elements that are common to all solutions in $\mathcal{L}$. Each building element in the commonality set is called a* commonality *of $\mathcal{L}$.*

The notion of commonalities of solutions is intimately related to the notion of distance between solutions. Solutions with zero distance must be equal and have all building elements in common. On the contrary, different solutions cannot have all building elements in common. The number of commonalities is non-increasing in $|\mathcal{L}|$ and obviously depends upon the method that was used to generate the solutions.

For a given multiset $\mathcal{L}$ of good solutions, the commonality set of $\mathcal{L}$ is a partial solution. Intuitively, this partial solution is a blueprint of good solutions that characterizes their essential structure.

We will now investigate the commonalities of solutions obtained with well known heuristics for the TSP and the JSSP.

2.3.1 Existence of commonalities

For the TSP and JSSP, we generated a number of good solutions by means of well known heuristics from the literature: the Chained Lin-Kernighan heuristic (Applegate et al. [6] and Johnson [35]) for the TSP and the tabu search heuristic of Nowicki and Smutnicki [51] for the JSSP. We focus on the question whether good solutions have commonalities and, if so, what the partial solutions that they induce look like.

Hypothesis 1. *Good solutions have many building elements in common.*

Traveling salesman. For TSPLIB instance vm1084, we performed five runs of Chained Lin-Kernighan, each of which is started with a Quick-Borůvka solution (see Chapter 3). Each run consists of 100,000 iterations. The resulting tours are all within one per mille of the optimum. The union of the tours is given in Figure 2.1. The edges in the intersection of the tours are colored black, while the edges in the union minus the intersection are colored grey. The black edges are commonalities, the grey edges are not common to all five solutions. The black edges form a partial tour consisting of a number of tour segments. Notice that the grey edges are not distributed evenly over the space: they appear only in certain cross-over regions where good alternative edges apparently exist. We conclude that the five tours have many edges in common.

Job shop scheduling. For the first three machines, the machine arcs common to five different optimum solutions to instance mt10 (see Fisher and Thompson [20]) are given in Figure 2.2. Only the machine arcs in the transitive reduction of each machine order are given. Apparently, jobs 5 and 6 are very flexible: they can be scheduled before or after any other job. The commonalities define a partial order on each machine. The solutions

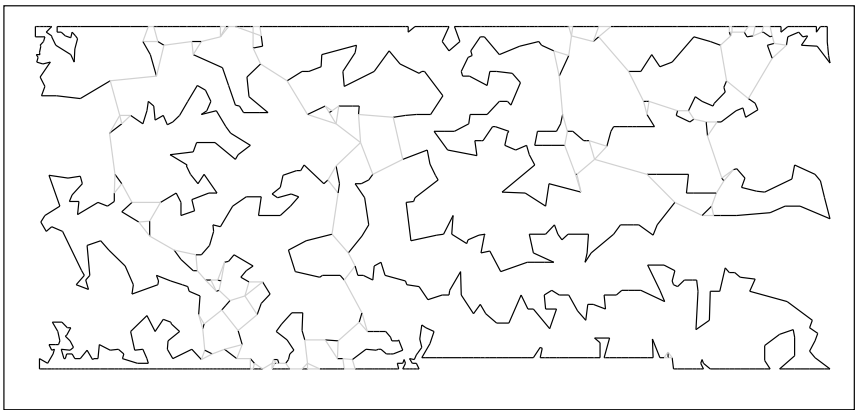

Figure 2.1: Existence of commonalities. The union of five different solutions to TSPLIB instance vm1084. Black edges are common to all five solutions and grey edges are not common to all.

were obtained with the local search heuristic of Nowicki and Smutnicki. The initial solution was obtained with a randomized construction heuristic. We conclude that the solutions have many machine arcs in common.

We conclude that good solutions to TSP's and JSSP's have many commonalities. In addition, we believe that good solutions to other combinatorial optimization problems that involve routing or sequencing also have many commonalities. Next, we study the number of commonalities in relation to the quality of the solutions under consideration.

2.3.2 Abundance of commonalities

We continue our investigation of the commonalities of solutions to TSP's and JSSP's that are produced by well known local search heuristics. Here, we discuss the number of commonalities and the relation with the quality of the solutions. It is trivial that the number of solutions with cost at most x is less than or equal to the number of solutions with cost at most y when $x < y$. We give empirical evidence that, for TSP and JSSP, the number of different building elements in the set of solutions with cost at most x decreases rapidly when x decreases to the optimum cost. Equivalently, the number of commonalities increases when x decreases to the optimum cost.

Hypothesis 2. *The number of commonalities increases with the quality of the solutions.*

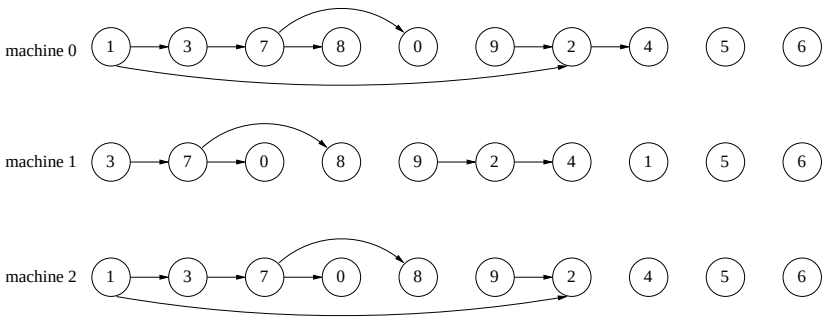

Figure 2.2: Existence of commonalities. Common machine arcs of five optimum solutions to mt10. Only the first three machines are shown.

Traveling salesman. In order to get an impression of the abundance of commonalities in the local optima generated by independent runs of the Chained Lin-Kernighan heuristic, we carried out the following experiment. For TSPLIB-instance rl11849 we did 50 independent runs with various run lengths T. We consider six run lengths: 1, 10, 100, 1,000, 10,000 and 100,000 iterations. For each run length, we count the number of different edges in the local optima produced by r independent runs, for $1 \leq r \leq 50$. Obviously, this number is non-decreasing in r. We calculate the ratio of the number of different edges to the size of the problem instance, in this case, 11,849 cities. For each run length, the ratio is given as a function of r in Figure 2.3. It appears that the number of different edges converges as the number of runs increases. For all run lengths, the number of different edges in the 50 local optima is less than three times the number of edges in a single tour. Furthermore, this number decreases as the run length T increases. The results were obtained with CLK starting from a Quick-Borůvka tour. Similar results were obtained when the initial solution was generated at random. As increasing the number of iterations generally results in finding better solutions, we conclude that the number of commonalities in the local optima produced by CLK increases significantly when the quality of the local optima increases.

Job shop scheduling. We carried out the following experiment. The 25 best local optima found during 25 independent runs of the tabu search heuristic of Nowicki and Smutnicki [51] are stored in a list $\mathcal{L}$. For every machine arc (u, v) in the best local optimum L_1 in $\mathcal{L}$, we determine whether or not all other local optima in $\mathcal{L}$ have u scheduled before v. The percentage of machine arcs in L_1 for which this is the case is given in Table 2.1. The initial solutions are obtained from a randomized construction heuristic. Each

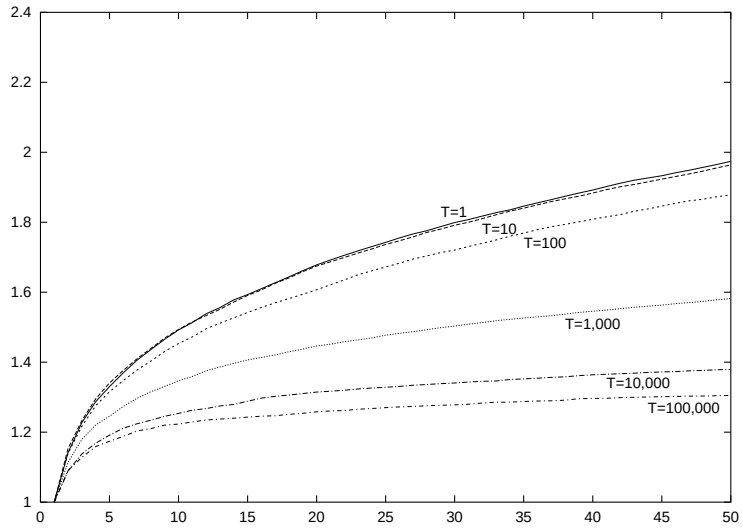

Figure 2.3: Abundance of commonalities for TSPLIB instance rl11849. The relative number of different edges on the y-axis as a function of the number of tours on the x-axis for various run lengths.

independent run halts after T subsequent non-improving iterations. Hence, $T = 0$ corresponds to taking a random initial solution and $T = 1$ to stopping at the first local optimum that is found. We repeat the experiment ten times and take the averages. Notice that for $T = 0$ and $T = 1$ there is exactly one solution in $\mathcal{L}$ for every independent run. For the other values of T this is not necessarily true. Although some runs contribute more to $\mathcal{L}$ than others, it is not the case that $\mathcal{L}$ consists entirely of local optima found during a relatively small number of runs. It is remarkable that the percentages of common machine arcs are very close for $T = 0$ and $T = 1$. On the other hand, the values for $T = 1$ and $T = 10$ differ considerably. For larger T, the percentage of common machine arcs increases to about 80%. Interestingly, the percentages are approximately the same for all problem instances that we considered. As the quality of the solutions generally increases when T becomes larger, we conclude that the number of commonalities in the local optima produced by the tabu search heuristic of Nowicki and Smutnicki increases when the quality of the solutions increases.

For TSP's and JSSP's, we conclude that the number of commonalities increases with the quality of the solutions. Similar results are expected for other combinatorial optimization problems that involve routing or sequencing. We believe that it is no coincidence that so many good solutions have

instance/T	0	10^0	10^1	10^2	10^3	10^4	10^5	10^6
ta21	35.7	36.2	63.2	70.6	72.3	77.7	78.3	82.2
ta22	36.4	37.1	64.6	65.7	74.7	77.2	78.7	80.1
ta23	35.8	38.4	62.3	69.9	73.9	77.5	79.8	83.9
ta24	36.4	37.3	64.3	68.4	73.1	76.7	83.0	88.9
ta25	35.9	37.2	64.3	72.1	73.1	73.0	75.2	79.4
ta26	35.9	38.2	63.9	70.3	71.4	75.7	80.2	80.2
ta27	36.2	38.2	61.9	70.3	74.1	79.0	81.3	83.9
ta28	36.7	37.8	63.0	67.7	76.6	79.5	80.5	82.0
ta29	36.3	38.2	66.7	71.9	78.8	83.6	85.2	86.1
ta30	35.4	36.8	62.2	69.7	74.1	73.6	77.9	83.5
yn1	37.1	39.8	64.9	73.2	79.1	81.1	82.0	85.5
yn2	37.1	38.0	65.6	70.5	74.7	78.7	80.8	82.6
yn3	36.3	36.6	61.1	70.7	76.0	80.2	85.5	87.8
yn4	32.8	34.4	57.2	70.2	72.0	78.4	80.6	81.8

Table 2.1: Abundance of commonalities. The average percentage of machine arcs in the best local optimum found that are common to the 25 best local optima found during 25 independent runs of the tabu search heuristic due to Nowicki and Smutnicki for the JSSP, for various problem instances and run lengths T.

so many characteristics in common. In the next section, we demonstrate that commonalities have applications in combinatorial optimization.

2.4 Applications of commonalities

Given that commonalities are abundant and that their quantity increases with the quality of the solutions, we could think of commonalities as characterizations of what good solutions look like: "commonalities are probably right".

Hypothesis 3. *Commonalities have useful applications in combinatorial optimization.*

We exploit the abundance of commonalities in two ways: in a restart mechanism for local search heuristics, and in a shave algorithm for the job shop scheduling problem that distills information from good solutions to guide the shaving process. Both applications are sketched below. The details are given in the remainder of the thesis.

2.4.1 Commonality-preserving restarts

In Chapter 1, we demonstrated that local search heuristics have a tendency to get stuck in an unpromising part of the search space. In an attempt to reduce the time that is wasted in unpromising areas, we perform multiple shorter runs, instead of a single long run. Unlike multiple independent runs, that start each run from scratch, we take advantage of the existence and abundance of commonalities of the solutions found by preceding runs in the following way. As stated before, it is often intuitively clear that certain building elements are likely to be part of a good solution. When the same building elements are found over and over again, they are "probably right", and we could as well put them in the initial solution of subsequent runs. This observation is the basis of our commonality-preserving restart mechanism, which we will describe below.

During the search, a list of solutions $\mathcal{L} = (L_1, L_2, \dots)$ is maintained such that $c(L_i) \leq c(L_j)$ whenever $i < j$. In every run, one or more solutions are inserted into $\mathcal{L}$. The insertion criterion of a solution depends on the problem at hand and on the local search heuristic; the goal is to obtain a diverse collection of promising solutions. First, we perform one or more independent runs, starting from a solution that is obtained with a constructive heuristic. When $\mathcal{L}$ contains at least two solutions, the initial solution for the next run is a perturbed copy of L_1, the best solution found so far, which is obtained as follows. In the *selection phase*, we determine which building elements in L_1 are accepted. Each building element in L_1 is accepted with a certain probability that depends upon the number of solutions in $\mathcal{L}$ that contain the building element. Building elements that are common to many solutions in $\mathcal{L}$ have an acceptance probability that is close to 1. Less common building elements have a relatively low acceptance probability. More generally, we require that good solutions in $\mathcal{L}$ have more influence on the selection process than bad solutions. Therefore, we give every $L_i \in L$ a weight to represent its relative quality in $\mathcal{L}$. This time, the acceptance probability of a building element not only depends upon the number of solutions of which it is part, but also on their relative quality. The building elements that are accepted will become part of the initial solution of the next run. As the result of the selection phase is a partial solution, in the *augmentation phase* we insert building elements at random in such a way that a complete solution is obtained. A new search is started from the thus constructed solution. By incorporating the commonalities of the solutions found by preceding runs in the initial solution of the next run, we are able to prune unfruitful portions of the search space, although we do not fix any building elements.

The list $\mathcal{L}$ acts as a global memory that attempts to capture what good solutions look like. Since $\mathcal{L}$ changes over time, building elements that first appeared "right" can turn out to be "wrong" later.

Our restart mechanism has at least two parameters. First, we must determine the *run length* at which the local search heuristic under consideration tends to get stuck. In general, longer runs are required for larger instances. In addition to the run length parameter, one or more parameters influence the acceptance probabilities of the building elements. Implicitly, the latter parameters determine the *amount of perturbation* that is inflicted upon the best solution found so far.

The main objective of the commonality-preserving restart mechanism is to reduce the chance of getting stuck in an unpromising part of the search space. Furthermore, commonality information can save time by directing the search to interesting parts of the search space. Obviously, our restart mechanism can also get stuck, for instance, when all solutions in $\mathcal{L}$ are in the same unpromising part of the search space that is surrounded by barriers. As a result, we must carefully select an insertion criterion in order to obtain a sufficiently diverse collection of solutions in $\mathcal{L}$.

related work

Many successful local search heuristics that are found in the literature employ some sort of memory to guide the search process. Back in 1973, Lin and Kernighan proposed their famous variable depth search heuristic for the TSP [42]. They observed that local optima found during the course of the algorithm have certain edges in common. They exploited this observation by fixing these common edges in the sense that it was forbidden to break them in subsequent steps. In this way, a memory is used to guide further search.

Later, other memory-based local search heuristics emerged, including tabu search and genetic algorithms. Tabu search exploits certain forms of memory to guide the search process to explore the solution space beyond local optimality. Simple forms of tabu search only use a short term memory to avoid cycling. More advanced forms of tabu search also employ longer term memories in such a way that promising areas can be searched more thoroughly (intensification), and the search can be directed to unexplored parts of the search space (diversification). The latter form of tabu search is also known as *adaptive memory programming*, or AMP. Many local search heuristics fit into this category, see Taillard et al. [70] and Glover & Laguna [30]. The main characteristics of AMP are the following:

- A set of solutions or some interesting characteristics of solutions is memorized.
- Initial solutions are generated using the data contained in the memory.
- The initial solution is improved by invoking a local search heuristic.
- The resulting solution is inserted into the memory.

For example, Rochat & Taillard [60] implemented a highly successful heuristic for vehicle routing problems. Vehicle routing problems (VRP's) are concerned with the delivery of goods to customers by a number of vehicles. A solution consists of a set of tours, one for each vehicle, such that each customer is visited exactly once. The objective is to minimize the total distance travelled by the vehicles. The heuristic consists of two phases. In the first phase, a number of reasonable solutions are generated by a tabu search heuristic and the tours in these solutions are memorized. In the second phase, a new initial solution is constructed by a process that is based upon selecting a number of non-overlapping tours. A new search is started from the thus constructed provisory solution. The tours in the best solution found are added to the memory, and the process is repeated.

Closely related to AMP is the notion of *vocabulary building*, see Glover & Laguna [30]. The basic idea is to identify interesting fragments of solutions as a basis for generating combinations. During the search process a memory consisting of such fragments is maintained, new fragments are inserted into the memory, and certain fragments are combined into larger fragments. The obvious difference with our approach is that commonality-preserving restarts operate on the level of building elements, while AMP and vocabulary building manipulate larger solution fragments.

Rosing & ReVelle [61] introduced *heuristic concentration*, which is closely related to AMP and vocabulary building. Their two-stage heuristic is particularly suited for location problems in which the number of facilities is given in advance

A technique that, like our commonality-preserving restart mechanism, operates on building elements is *the principle of persistent attractiveness* (Glover & Laguna [30]). This principle is concerned with building elements that persist in being attractive to bring into a solution but, unfortunately, the moves to accomplish this are rarely selected because most of the time other moves seem slightly more promising. For each building element a number of characteristics are maintained during the search process, including the number of iterations in which it is part of the solution, the number

of times that a move that brings it into the solution is selected, and the number of times that such a move is among the best moves available in the neighborhood of a solution. When a given building element is not frequently part of the solution and the moves that would bring it into the solution are frequently one of the best moves available although they are not frequently selected, these moves deserve a higher evaluation. Based upon this information, the move evaluation function is modified such that this kind of move is favored. The difference with our approach is that information about building elements is employed within the local search heuristic, i.e. the neighbor selection rule is adapted, while we only employ the information in the construction of an initial solution.

Combinatorial optimization problems can often be stated in terms of decision variables that must be assigned a value from a given domain. Glover & Laguna define *highly consistent variables* as variables that are more frequently found in elite solutions. Furthermore, they define *strongly determined variables* as variables that would cause the greatest disruption by changing their values. The idea is to identify the more consistent and strongly determined variables, and then to generate solutions that give these variables their "preferred values" in a restart mechanism or by modifying the neighbor selection rule. It is clear that these notions are closely related to the concept of commonalities. Unlike our approach, these variables are often fixed during the search.

Genetic algorithms were invented by Holland [33] in 1975. Starting from a set of initial solutions, the *population*, a number of *generations* are produced. Each subsequent generation is obtained as follows. First, a number of solution pairs, the *parents*, are selected from the population, based on their cost. A *crossover* operation is applied to each of these solution pairs resulting in a new solution, the *offspring*, which combines features of both parent solutions. Each offspring is usually subjected to a randomized *mutation* operation that slightly perturbs it. Finally, the offspring is added to the population to form the next generation. Commonality-preserving restarts can be viewed as a genetic approach where the offspring can have more than two parents. The solution list $\mathcal{L}$ corresponds to the population. Generating an initial solutions corresponds to a crossover step, while applying local search to the initial solution corresponds to a mutation step.

2.4.2 Shaving

Another application of commonalities is given in Chapter 7, where they are employed to speed-up the shaving algorithm due to Martin and Shmoys

[45], which derives lower bounds for job shop scheduling instances. Shaving is a systematic way to update the *processing window* of each operation in schedules of length at most T, for a given schedule length T. Each operation is required to start within its processing window, in order to let the schedule complete by time T. Initially, the processing window of an operation is specified by the sum of the process times of its job predecessors, the *head*, and the sum of the processing times of its job successors, the *tail*. Given these heads and tails, a preemptive lower bound is calculated for each machine. For each operation, shaving tries to reduce the processing window as follows. When it is shown that fixing the start time of an operation at a certain value always results in a schedule of length more than T, it is justified to remove that value from the processing window of the operation. These updates can have consequences for the heads and tails of other operations. Reducing processing windows can lead to improved preemptive lower bounds for the machines. In addition, when the processing window of an operation becomes empty, schedules of length at most T cannot exit, hence $T + 1$ is a lower bound. The order in which the operations are shaved can have a significant influence on the running time of the shave algorithm. Given a number of good solutions, we attempt to use commonalities in order to determine a shave order that heuristically optimizes the propagation of updates.

Chapter 3

Traveling salesman problem

3.1 Introduction

In the *traveling salesman problem* (TSP) we are given a finite set of cities and for each pair of distinct cities the cost of travel between them. We restrict ourselves to the symmetric case, where the cost of travel between cities is the same in both directions. We want to find a cheapest *tour* that passes through all the cities exactly once and returns to the point of departure. The TSP is well studied in the literature, see Lawler, Lenstra, Rinnooy Kan and Shmoys [40], Johnson and McGeoch [35], Jünger, Reinelt and Rinaldi [36], and Reinelt [57].

Currently, one of the most effective TSP heuristics is the Chained Lin-Kernighan algorithm (CLK) proposed by Martin, Otto and Felten [46, 47]. Although CLK produces very good solutions, it appears that CLK can get stuck in an unpromising part of the search space, meaning that no significant improvements are found when given more time. A common way to avoid this behavior is by performing multiple independent runs. The main drawback of this approach is that all efforts of preceding runs are wasted. We present a randomized algorithm that repeatedly invokes Chained Lin-Kernighan while maintaining a global memory that is used to construct a reasonable initial solution from which the next search starts. The best solution found during every run is added to the global memory. We are interested in edges that are common to many good solutions in the global memory. These *commonalities* are thought of as being "probably right" and will be part of the new initial solution with high probability. An initial solution for the next run is obtained by perturbing the best solution found so far, based on the commonality information. In this *commonality-preserving restart mechanism*,

information gathered during previous runs is used in subsequent runs.

We select seven instances from the TSPLIB set of test instances collected by Reinelt [56, 58] with sizes ranging from 3,795 to 85,900 cities. For each problem instance, we fix the total number of CLK iterations and determine the best way to use them. We compare four strategies that are based on CLK: a single long run, multiple independent runs starting from a solution obtained by a constructive heuristic, restarts from a solution that is obtained by perturbing the best solution found so far at random, and commonality-preserving restarts.

The commonality-preserving restart mechanism improves the finite-time performance of CLK: given a sufficiently large and practically feasible time span, we are able to find better solutions and increase the robustness. Independent runs and restarts with random perturbation also improve upon a single long run, but they are not as good as the commonality-preserving restart strategy.

We start with a description of the Lin-Kernighan heuristic in Section 3.2. In Section 3.3, we describe the Chained Lin-Kernighan implementation of Applegate, Bixby, Chvátal and Cook and demonstrate that it tends to get stuck. Section 3.4 describes our commonality-preserving restart mechanism for Chained Lin-Kernighan. The experimental setup and the computational results are given in Sections 3.5 and 3.6, respectively.

3.2 Lin-Kernighan

Suppose that we have a n-city TSP, with $c(u, v)$ denoting the cost of travel between city u and v. The Lin-Kernighan heuristic [42] is a *variable depth* local search heuristic for symmetric TSP's. We assume that tours are oriented. In a given tour, the successor of city u is denoted by $succ(u)$ and the predecessor is denoted by $pred(u)$. The basic operation is to invert a subsequence $(u, \ldots, v)$ of a tour, denoted by $flip(u, v)$. See Figure 3.1. When

$$c(pred(u), u) + c(v, succ(v)) > c(pred(u), v) + c(u, succ(v)),$$

the tour is improved. The well known 2-opt algorithm (see Croes [18]) continues until no improving flip move is found.

Flip operations are also the building elements of Lin-Kernighan. While the 2-opt algorithm searches for a single flip, Lin-Kernighan attempts to build a sequence of flips such that the combined effect of these flips produces a better tour. By allowing that intermediate tours are more costly than the

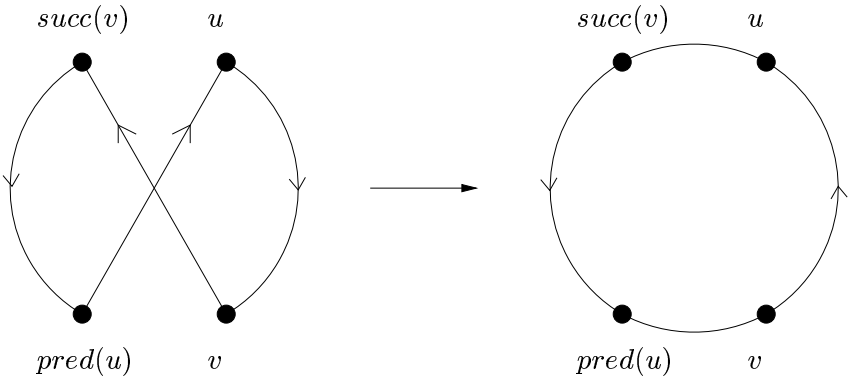

Figure 3.1: The move $flip(u, v)$ inverts the cities between u and v.

initial tour, Lin-Kernighan can escape from local optima where the 2-opt algorithm would get stuck and terminate.

Given an initial tour τ_0, we construct a sequence of flips, each initial subsequence of which appears to be promising in the sense that it might lead to a better tour. Let τ denote the tour that we obtain when the sequence of flips is applied to τ_0. For a city u, let $pred(u)$ and $succ(u)$ denote the predecessor and the successor of u in τ, respectively.

We follow the description of Applegate et al. [4]. Let $base$ be a fixed vertex. We consider only flips of the form $flip(succ(base), probe)$, for vertices $probe$ that are distinct from $pred(base)$, $base$ and $succ(base)$. Such a flip replaces the pair of edges $(base, succ(base))$ and $(probe, succ(probe))$ by $(succ(base), succ(probe))$ and $(base, probe)$, see Figure 3.2. When

$$\begin{aligned} &c(base, succ(base)) + c(probe, succ(probe)) > \\ &\quad c(succ(base), succ(probe)) + c(base, probe), \end{aligned} \tag{3.1}$$

the move is improving. Instead of such an improving move, for the first flip, Lin-Kernighan requires only that

$$c(base, succ(base)) - c(succ(base), succ(probe)) > 0,$$

which tries to improve a single edge in the tour.

For the subsequent flips, let Δ be a variable that is initially zero and after each $flip(succ(base), probe)$ is incremented by the local improvement

$$\begin{aligned} &c(base, succ(base)) - c(succ(base), succ(probe)) + \\ &\qquad c(probe, succ(probe)) - c(base, probe); \end{aligned}$$

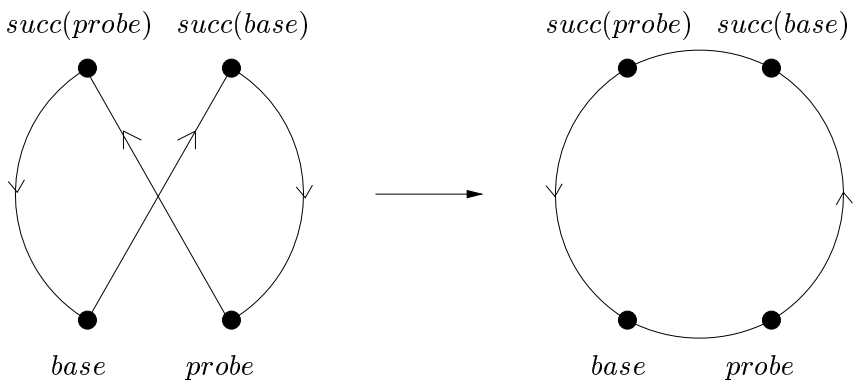

Figure 3.2: The move flip($succ(base), probe$).

as a result, the cost of τ is always equal to the cost of τ_0 minus Δ.

For the subsequent flip moves, it is required that probe is a *promising vertex*, which means that

$$\Delta + c(base, succ(base)) - c(succ(base), succ(probe)) > 0.$$

By allowing that Δ is negative, intermediate tours can be more costly than the initial tour. When an improving tour is found, the sequence of flips is memorized and the search continues for even better tours.

As vertex *base* is fixed, vertex *probe* is promising when $c(succ(base), succ(probe))$ is small enough. When checking for a promising vertex, we consider the edges incident with vertex $succ(base)$, ordered by increasing cost. When considering such an edge $(succ(base), a)$ we let $probe = pred(a)$, see Figure 3.3. When more than one promising vertex exists, we select the one that maximizes

$$c(pred(a), a) - c(succ(base), a), \tag{3.2}$$

which includes another term from equation (3.1). In order to reduce the computation times, only a limited number of vertices a are considered. These vertices a are called the *neighbors* of vertex $succ(base)$; frequently, only the k nearest vertices are considered, for some fixed k. A neighbor a of $succ(base)$ is called promising when $probe = pred(a)$ is a promising vertex. The flip search procedure is outlined below.

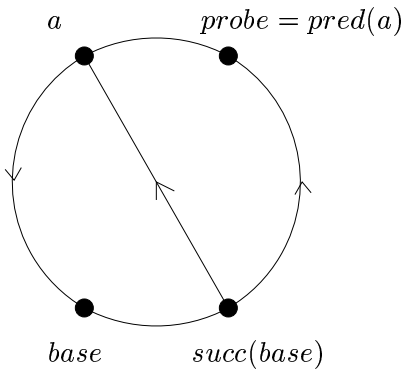

Figure 3.3: Finding a promising vertex.

Flip Search:

$\Delta = 0$
`while there exist promising neighbors of` $succ(base)$
  `let` a `be the promising neighbor of` $succ(base)$ `that maximizes equation (3.2)`
  $\Delta = \Delta + c(base, succ(base)) - c(succ(base), a) + c(pred(a), a) - c(base, pred(a))$
  `add` $flip(succ(base), pred(a))$ `to the flip sequence`
`endwhile`

In order to enhance the flip search procedure, a backtracking scheme is used. Instead of concentrating only on the most promising neighbor, alternative neighbors are also considered. In their original paper [42], Lin and Kernighan proposed that five choices for the first flip in the sequence, and for each of these flips another five choices for the second flip in the sequence should be explored. At the other levels, Lin and Kernighan allow only one flip. In the implementation of Applegate et al., a backtracking scheme is denoted by a tuple $(b_1, b_2, \ldots, b_z)$, for some integer z. The value b_i is called the *breadth* of level i because at most b_i alternative flips are explored at level i, and no backtracking is permitted at any level lower than z. At level i the best b_i flips according to equation (3.2) are explored. Alternative flip search procedures have been proposed by Mak and Morton [43], Reinelt [57], and Johnson and McGeoch [35].

If the search is successful in finding a sequence of flips having an initial subsequence that improves the tour, this subsequence is applied to the tour. Otherwise, Lin-Kernighan halts. For further details, the reader is referred to Applegate et al. [4, 6] and Johnson [35].

3.3 Chained Lin-Kernighan

For a long time, the Lin-Kernighan heuristic was the method of choice for finding excellent solutions to symmetric TSP's. An obvious way to obtain even better solutions is by performing, as long as time permits, multiple independent runs of Lin-Kernighan, each started in a different solution, and taking the best. Martin, Otto and Felten [47] improved upon these independent runs by perturbing the best Lin-Kernighan tour in a certain way and reapplying the algorithm. We refer to their algorithm as Chained Lin-Kernighan, to match the chained local optimization concept introduced by Martin and Otto [48].

Chained Lin-Kernighan consists of a number of iterations and each iteration invokes the Lin-Kernighan heuristic. In the first iteration, Lin-Kernighan is started from an initial solution obtained by a constructive heuristic. In the subsequent iterations, Lin-Kernighan is started from the best solution found so far, after it is perturbed by a so-called *double bridge* move, see Figure 3.4. A double bridge move is a special 4-opt move that consists of two 2-opt moves that are illegal in the sense that they transform the tour into two disjoint cycles. A double bridge move has the property that it can alter the global shape of the tour and it cannot be undone very easily by subsequent invocations of Lin-Kernighan. If this effort produces a better tour, we discard the old LK tour and work with the new one. Otherwise, we continue with the old tour and apply a different double bridge move. CLK halts after a given number T of iterations, which is called the run length.

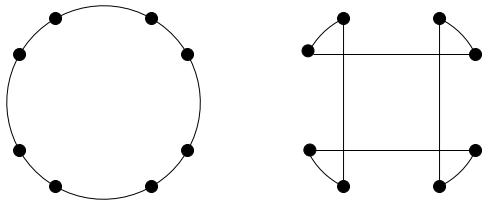

Figure 3.4: Double bridge move.

We use the CLK implementation of Applegate, Bixby, Chvátal and Cook, which is part of the Concorde project [3]. Their CLK implementation has many parameters, including the constructive heuristic for the initial solution, the backtracking scheme of Lin-Kernighan, and the class of double bridge moves.

We tried to select the best possible parameters for CLK. In a personal

communication with one of the authors, the main parameters were determined. These parameters are optimized for instances with 10,000 or more cities. The initial tour is generated by the Quick-Borůvka constructive heuristic, which is based on the minimum-weight spanning tree algorithm of Borůvka [9]. It works as follows. First, the cities are sorted according to their first coordinate. The cities are processed in order and for each city u with degree less than two, the cheapest edge $\{u, v\}$ over all cities v having degree less than two is inserted, provided that it does not create a subtour. According to Applegate et al. [6], the Quick-Borůvka heuristic is not as effective as certain other constructive heuristics, but it appears to work well for CLK. The backtracking scheme that we adopt is $(4, 3, 3, 2)$. There are many ways to generate a double bridge move. We will use double bridge moves for which the edges are geometrically close. The choice of the edges is randomized, hence CLK is randomized.

From now on, we shall treat CLK as a black box. We demonstrate that it tends to get stuck and that it can be improved by restarting a number of times. Any other heuristic for the TSP could have been used instead of CLK, but it was more appealing to use CLK because it is among the most effective heuristics for the symmetric TSP.

3.3.1 CLK gets stuck

For a number of TSPLIB [56, 58] test instances we did five independent runs of CLK. The initial solution is produced by the Quick-Borůvka constructive heuristic. We calculated the relative excess permillage above the best known upper bound as a function of the iteration number. The results are given in Figure 3.5. The trajectories of these independent runs are improving very fast in the beginning of the search processes. Quite early in the search process, the trajectories start to diverge. Apparently, something can go wrong early in the search process, that is not easily corrected later on. Often, there appear to be almost no improvements when CLK is given more time. Similar results were obtained when the initial solution is generated at random. We conclude that CLK runs the risk of getting stuck in an unpromising part of the search space.

Some runs produce tours that are significantly better than the tours produced by other runs. By performing multiple shorter runs, we attempt to increase the robustness of CLK without sacrificing the quality of the solutions found.

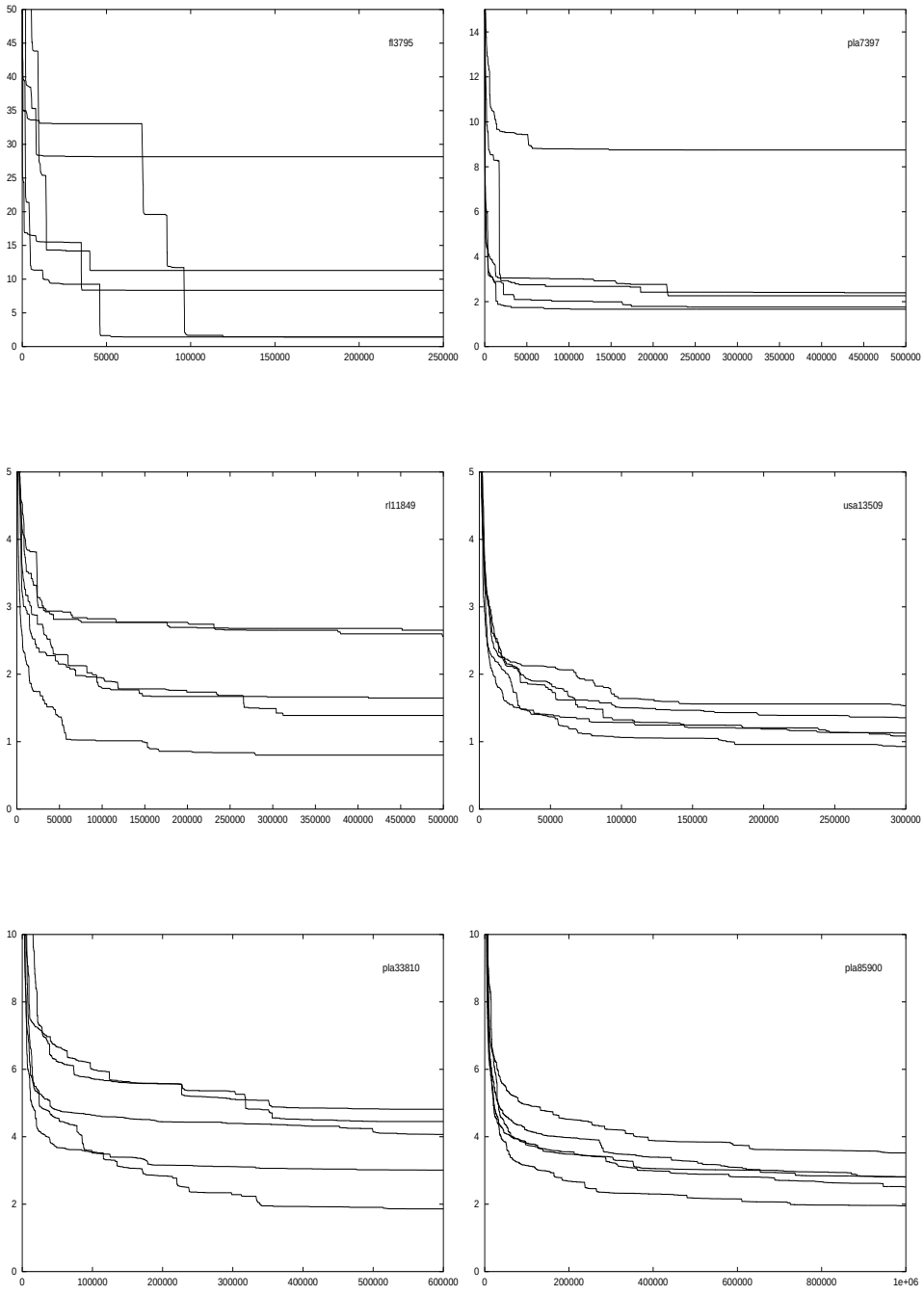

Figure 3.5: The trajectories of five runs of CLK for a number of TSPLIB instances. The solution quality on the y-axis is given in per mille above the best known tour as a function of the iteration number on the x-axis.

3.4 Restarting Chained Lin-Kernighan

Suppose that we have a problem instance and a maximum number of CLK iterations T_{total}. Because a single long run of CLK is likely to get stuck, we hope that a sequence of shorter runs can improve the performance of CLK.

The most obvious alternative to a single long run is multiple independent runs. This strategy has only one parameter, the run length. We compare a single long run to multiple shorter runs on a time-equivalent basis, hence, a run length of T CLK iterations implies that we can perform $\lfloor T_{\text{total}}/T \rfloor$ runs. The drawback of this approach is that each run starts from scratch and does not take advantage of the knowledge gathered during the preceding runs.

In the remainder of the section we propose two alternative restart mechanisms for Chained Lin-Kernighan. First, we describe a commonality-preserving restart mechanism and then we sketch restarts with random perturbation, which is more or less a special case of the commonality-preserving restart mechanism.

3.4.1 Commonality-preserving restart mechanism

A tour is often represented as a graph that has a vertex for every city and an edge for every pair of cities that are visited subsequently. A solution corresponds to a subset of the set of *building elements*, consisting of all undirected edges $\{u, v\}$. The cost of a tour σ, denoted by $c(\sigma)$, is equal to the sum of the cost of the edges that constitute it. Given this representation of tours, we will now introduce a *commonality* concept based upon it.

Definition 3.4.1. *The* commonality set *of a multiset Σ of tours is defined as the subset of edges that are common to all tours in Σ. Each edge in the commonality set is called a* commonality *of Σ.*

Obviously, the number of commonalities in Σ is non-increasing in the number of tours in Σ.

With the commonality concept, we try to characterize the general structure of good solutions. When certain edges are common to many good solutions, they are interpreted as being "probably right" and we intent to construct initial solutions that preserve them, since it appears to be useless to exclude them from the next initial solution.

In order to get an impression of the abundance of commonalities in the local optima generated by independent runs of the Chained Lin-Kernighan heuristic, we carried out the following experiment. For a number of TSPLIB instances, we did 50 independent runs with various run lengths T. For each

problem instance and run length, we counted the number of different edges in the local optima produced by the first r independent runs, for $1 \leq r \leq 50$. Obviously, this number is non-decreasing in r. We calculate the ratio of the number of different edges to the size of the problem instance. For each problem instance and run length, the ratio is given as a function of r in Figure 3.6. It appears that the number of different edges converges as the number of runs increases. For all instances, the number of different edges in the 50 local optima is less than three times the number of edges in a single tour. Furthermore, this number decreases as the run length T increases, indicating that the number of commonalities increases with the quality of the solutions. The results were obtained with CLK starting from a Quick-Borůvka tour. Similar results were obtained when the initial solution was generated at random. We conclude that the local optima produced by the CLK heuristic have many commonalities.

We will now describe our commonality-preserving restart mechanism. During the search, a list of tours $\mathcal{L} = (L_1, L_2, \ldots, L_k)$ is maintained such that $c(L_i) \leq c(L_j)$ whenever $i < j$. After every run, the best solution found is inserted into $\mathcal{L}$. We start with $k = 2$, where L_1 and L_2 are obtained by two independent runs starting from a Quick-Borůvka solution. The two independent runs are followed by a number of restarts. For every restart, an initial solution L_0 is constructed as follows. Initially, L_0 is equal to L_1, which is the best solution found so far. In the *selection phase*, we determine which edges in L_0 are accepted. Each edge $\{u, v\} \in L_0$ is accepted with a certain probability that depends on the number of tours in $\mathcal{L}$ that have the edge. Naturally, we want that good solutions in $\mathcal{L}$ have more influence on the selection process than bad solutions. Therefore, we give every $L_i \in \mathcal{L}$ a weight $w(L_i) \in [0, 1]$ to represent its relative quality in $\mathcal{L}$. The solutions are weighted such that good solutions have a higher weight than bad solutions:

$$w(L_i) := \rho^{1000\frac{c(L_i)-c(L_1)}{c(L_1)}} \in [0, 1],$$

for some $\rho \in (0, 1]$. The exponent of ρ is the *excess permillage* of $c(L_i)$ over $c(L_1)$. When $\rho = 1$, all L_i have equal weight, but when $\rho < 1$, a penalty is given for every per mille above $c(L_1)$. Furthermore, the weight of L_1 is always 1 and as $\rho \downarrow 0$, $w(L_i) \downarrow 0$ when $c(L_i) > c(L_1)$.

Consider an edge $\{u, v\} \in L_0$. When the sum of the weights of the solutions in $\mathcal{L}$ that have that edge is close to the total weight of the solutions in $\mathcal{L}$, the edge is highly desirable and should be part of the new initial tour with high probability. Therefore, we define the *acceptance probability* of

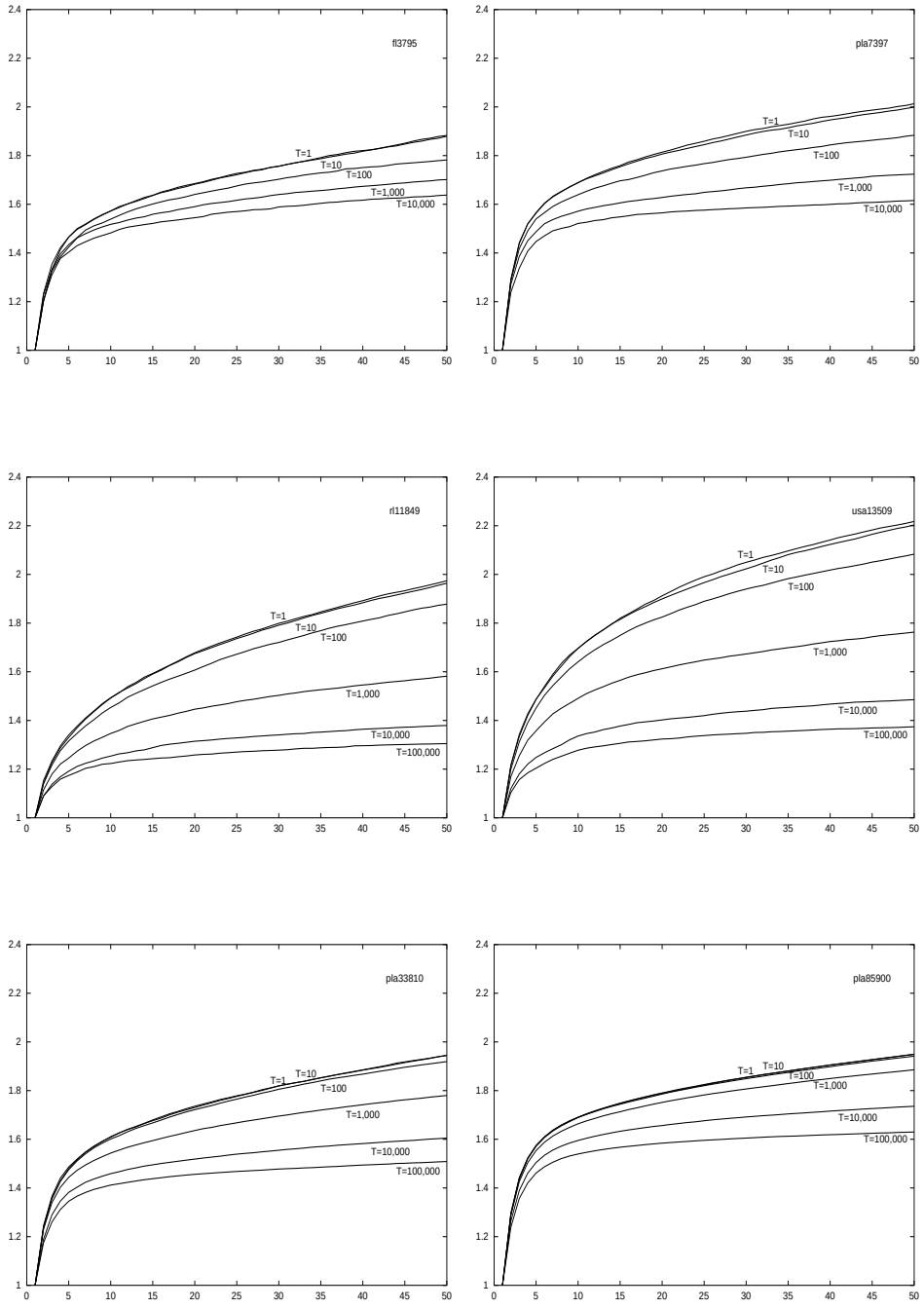

Figure 3.6: The relative number of different edges as a function of the number of tours for various run lengths.

$\{u, v\}$ as

$$p_{\text{acc}}(u, v) := \left(\frac{\sum_{\{L_i \in L | \{u,v\} \in L_i\}} w(L_i)}{\sum_{L_i \in L} w(L_i)} \right)^{\psi} \in [0, 1],$$

for some $\psi \in [0, \infty)$. The parameter ψ controls the amount of perturbation. When $\psi = 0$, all $\{u, v\} \in L_0$ are accepted and L_0 is not perturbed. As $\psi \to \infty$, only edges $\{u, v\} \in L_0$ that are common to all $L_i \in \mathcal{L}$ are accepted.

The result of the selection phase is a partial tour, consisting of a number of tour segments. In the *augmentation phase* we reconnect the segments of the partial tour at random. We repeatedly select two endpoints at random that do not belong to the same segment and join them by an edge. When there are no more endpoints, we have a complete tour. Testing whether or not two endpoints belong to the same segment is implemented with a union-find data structure. By connecting endpoints at random, many bad edges are created. It is likely that a single run of Lin-Kernighan gets rid of most of the really bad edges. What remains are a number of interesting alternative edges that could lead to further improvements beyond L_1.

Our restart mechanism has three parameters: the run length T, the weight parameter ρ, and the perturbation parameter ψ. Intuitively, the run length determines how well the vicinity of the best solution found so far is explored. A long run requires a rather big perturbation since the surroundings of the best solution found so far are thoroughly explored. Similarly, a short run should be followed by a small perturbation. The number of edges to be perturbed is determined by the parameter ψ. The parameter ρ is related to the concepts of *intensification* and *diversification* in the following way. When $\rho = 1$, all tours $L_i \in L$ have the same weight and their "votes" are considered equally important during the selection phase. In case $\rho < 1$, the tours with cost $c(L_1)$ have more influence than the other tours in $\mathcal{L}$, resulting in a higher acceptance probability of the edges in L_0. The former case resembles diversification while the latter resembles intensification.

3.4.2 Restarts with random perturbation

In order to determine whether or not the commonality information is useful, we compare our restart mechanism with restarts that perturb the best solution found so far at random. The first CLK run is started from a Quick-Borůvka solution. After the initial run, we do a number of restarts, while maintaining the best solution found so far, denoted by σ^*. The initial solution of the next CLK run is a perturbed version of σ^*. Every edge in σ^* is accepted with a fixed probability p, which is a parameter of the restart

Instance	LB	UB	T^*	T_{total}	time
fl3795	28,772	28,772	25,000	600,000	$5.6 \cdot 10^3$
pla7397	23,260,728	23,260,728	25,000	600,000	$4.6 \cdot 10^3$
rl11849	923,288	923,288	30,000	720,000	$5.3 \cdot 10^3$
usa13509	19,982,859	19,982,859	25,000	600,000	$6.4 \cdot 10^3$
d18512	645,198	645,255	40,000	960,000	$6.7 \cdot 10^3$
pla33810	66,005,185	66,059,941	50,000	1,200,000	$12.0 \cdot 10^3$
pla85900	142,307,500	142,409,553	75,000	1,800,000	$22.0 \cdot 10^3$

Table 3.1: Best knows upper bounds and lower bounds, the critical point and the total number of CLK iterations, and the running times for the long run in seconds on a Pentium III/450 MHz.

mechanism. In the augmentation phase, the tour segments are connected at random in the same way as we did for the commonality-preserving restart mechanism.

3.5 Experimental setup

Suppose that we have a problem instance and we do not want to spend more than a given number T_{total} of CLK iterations. We try to determine the best strategy to use them: a single long run (LR), multiple shorter independent runs (IR), commonality-preserving restarts (CP), and restarts with random perturbation (RP).

Our test instances are taken from TSPLIB. For all problems, either the value of a provably optimal solution or an interval given by the best known lower and upper bounds is listed in Table 3.1. The list dates from 25 October 1999 [59]. All problem instances are geometric. The pla instances use the rounded up Euclidean norm, the other instances use the rounded Euclidean norm. Problem instance fl3795 is a drilling problem, the pla instances are derived from programmable logic arrays, and the other instances are city coordinates.

In order to select appropriate run lengths, we examine, for each test instance, the trajectories of five long CLK runs, see Figure 3.5. In the beginning of the search many large improvements are found, but after a while the steepness decreases and the trajectories start to diverge. For each test instance, we estimate the *critical point* T^* at which the five independent runs are slowed down significantly while at the same time the trajectories

are considerably diverged. The advantage of this "visual" method is that T^* will depend on the size of the instance, which seems plausible.

For all strategies except LR, we must decide which run lengths to use. We will always have that the number of runs times the run length is fixed for a given problem instance. To ensure divisibility, the possible run lengths and T_{total} are a multiple of T^*. For every instance, the total number of iterations T_{total} is chosen to be equal to $24T^*$. We consider the following run lengths: T^*, $2T^*$, $3T^*$, $4T^*$ and $6T^*$. As a result, instead of a single run of $24T^*$ iterations we can also have 24 runs of length T^*, 12 runs of length $2T^*$, 8 runs of length $3T^*$, 6 runs of length $4T^*$ or 4 runs of length $6T^*$. Notice that the total number of iterations does not grow very fast with the size of the instance; in fact, it grows sub-linearly. The corresponding running times for a single long run are given in Table 3.1. Because of the limited number of restarts, there is almost no computational overhead. Consequently, the running times of all four strategies are approximately the same.

In the commonality-preserving restart mechanism, we start with two independent runs. The number of subsequent restarts is 22, 10, 6, 4 or 2, depending on the run length. As a result of a number of initial experiments, we select $\rho \in \{0.9, 0.98, 1.0\}$ and $\psi \in \{0.25, 0.5, 1.0\}$. The values for ρ and ψ are chosen in such a way that two of them are fairly extreme, and the other one is reasonable.

Restarts with random perturbation starts with a single run. After the initial run, we do 23, 11, 7, 5 or 3 restarts, depending on the run length. The parameter p is chosen in such a way that the total number of edges that is perturbed is approximately the same as in the commonality-preserving restart mechanism described above. We found that the amount of perturbation inflicted by the commonality-preserving restart mechanism ranges between 2 and 6 percent when $\psi = 0.25$ and between 7 and 16 percent when $\psi = 1.0$. As a result, we select $p \in \{0.85, 0.9, 0.95\}$.

The major advantage of fixing the number of iterations for every instance is that the experiments can be carried out on different machines in parallel without concern of their different speeds.

3.6 Results

In order to identify proper parameter values, we perform a number of computational experiments. Due to the randomized nature of CLK, the outcome of each experiment is a random variable. We are interested in the average and standard deviation of the outcomes of our computational experiments.

For a given problem instance, strategy (LR, IR, CP, RP), and parameter set, the average of the outcomes of a number of computational experiments gives an indication of the effectiveness of the heuristic, while the standard deviation is a measure of the robustness. When comparing parameter sets, our primary objective is to minimize the average result and our secondary objective is to minimize the standard deviation.

Due to the immense popularity of the TSP and the TSPLIB-instances, many researchers have been hunting for better solutions. In the literature, most results concerning upper bounds for TSP instances are expressed in terms of the permillage above the optimum or the best known lower bound. The excess permillage of the solutions found by our computational experiments typically ranges between 0 and 4. Given this small range, we use a significance level of 0.1 per mille to distinguish between the different parameter values.

We did ten computational experiments for every problem instance, strategy, and parameter set. Consequently, for every instance we did 10 times LR, 50 times IR, 150 times RP, and 450 times CP, giving a total of 660 experiments per instance. As we consider seven problem instances, the total number of computational experiments is 4,620. Because of the practical limitations imposed by the running times, see Table 3.1, we assume that ten experiments are sufficient to distinguish between the different parameter values. We try to identify the best parameter values and seek a consistent relation between the parameter values and the size of the problem instance.

3.6.1 Very best parameters

It is tempting to simply select the best parameters. For each restart strategy and problem instance, we single out the best parameter set, i.e., the one with the lowest average and in case of a draw between two or more parameters sets, the one with the lowest standard deviation. The results of the long run and each of the restart strategies are given in Table 3.2. Except for the largest instance, pla88590, we conclude that IR is often significantly better than LR, with respect to the average and the standard deviation. The average of RP is generally better than the average of IR, and the standard deviation is about the same. RP gives the same results for pla85900 as LR. In addition, the average result of CP is slightly better than that of RP, while the standard deviation is approximately the same. Furthermore, CP produces better results for pla85900 than LR does.

strategy		fl3795	pla7397	rl11849	usa13509	d18512	pla33810	pla85900
LR	T	$24T^*$	$24T^*$	$24T^*$	$24T^*$	$24T^*$	$24T^*$	$24T^*$
	av	**5.0**	**2.0**	**3.0**	**1.2**	**0.9**	**3.6**	**2.6**
	sd	2.8	0.8	0.8	0.3	0.1	0.7	0.3
IR	T	T^*	$3T^*$	$2T^*$	$2T^*$	$6T^*$	$6T^*$	$6T^*$
	av	**1.5**	**1.4**	**1.9**	**1.1**	**0.9**	**3.3**	**2.7**
	sd	1.1	0.3	0.2	0.2	0.1	0.6	0.2
RP	T	T^*	$3T^*$	T^*	$3T^*$	$3T^*$	$6T^*$	$6T^*$
	p	0.85	0.85	0.85	0.90	0.90	0.85	0.95
	av	**1.1**	**1.2**	**1.6**	**0.9**	**0.9**	**3.1**	**2.6**
	sd	1.2	0.3	0.3	0.1	0.1	0.5	0.3
CP	T	$2T^*$	T^*	$2T^*$	$3T^*$	$3T^*$	$3T^*$	$6T^*$
	ρ	0.98	0.98	0.90	1.0	1.0	0.98	1.0
	ψ	1.0	0.25	0.5	0.5	0.25	0.25	0.25
	av	**1.1**	**1.1**	**1.3**	**0.8**	**0.8**	**3.0**	**2.5**
	sd	1.0	0.3	0.2	0.2	0.1	0.4	0.3

Table 3.2: The average (av) and standard deviation (sd) of the results corresponding to the long run and corresponding to the best parameters for each of the three restart strategies.

Instance	LR	IR	RP	CP
fl3795	5.0	[1.5, 3.8]	[1.1, 5.7]	[1.1, 5.1]
pla7397	2.0	[1.4, 1.7]	[1.2, 1.6]	[1.1, 1.7]
rl11849	3.0	[1.9, 2.1]	[1.6, 1.9]	[1.3, 2.1]
usa13509	1.2	[1.1, 1.3]	[0.9, 1.2]	[0.8, 1.1]
d18512	0.9	[0.9, 1.2]	[0.9, 1.2]	[0.8, 1.1]
pla33810	3.6	[3.3, 3.7]	[3.1, 3.7]	[3.0, 3.8]
pla85900	2.6	[2.7, 3.4]	[2.6, 3.4]	[2.5, 3.4]

Table 3.3: Overview of the range of the results in permillage above the best known upper bound, over all parameter sets for each strategy.

Unfortunately, it appears that for each of the three restart strategies, the parameters behave erratically in the sense that there is no clear relationship between the parameter values and the size of the problem instance. In the next section, we present an alternative parameter selection methodology.

3.6.2 Consistent parameters

For each test instance and strategy, the results of the best parameter sets and the results of the worst parameter sets are given in Table 3.3. Apparently, the ranges are often relatively small. Furthermore, we found that, for all restart strategies, there are many alternative parameter sets that perform almost as well as the very best parameters. In this section, we attempt to identify, for each of our restart strategies, a consistent relationship between each of the parameters and the size of the problem instance. In this way, we hope to be able to select appropriate parameters for problem instances other than the test instances.

Although the two parameters of RP and the three parameters of CP are not necessarily independent of each other, we investigate for each of these parameters in isolation the relation with the size of the problem instance by aggregating the results over the other parameters. Aggregating means that for every value of a parameter, we calculate the average and standard deviation of the results of all parameter sets that have that value for the parameter. Consequently, we have far more than ten experimental results for every parameter value, which will generally improve the reliability of the results. In this way, we hope to find a consistent relation, i.e., a trend, between the parameter and the size of the problem instance. For each parameter, we first identify the trend, and then we select the best parameter value that

satisfies the trend. As following the trend is more important than obtaining the very best results, in some cases, minor concessions (compared to Table 3.2) are required. A detailed analysis for each of the restart strategies is given below.

Analysis of IR

Multiple independent runs have only one parameter, the run length. We calculate the averages over the ten runs and seek a relationship between the run length and the size of the problem instance.

We give a brief summary of the results. For the smallest instance, fl3795, the best results are obtained with the shortest run length. The results monotonously get worse for longer runs. The results of the medium sized instances, pla7397, rl11849 and usa13509, roughly resemble a u-shape, i.e., the results get worse to the left and to the right of the best run length. For the larger instances, d18512, pla33810, and pla85900, the results get monotonously better when the run length is increased; the best results are obtained with the longest run length. We conclude that there is a clear relationship between the run length and the size of the problem instance. The preferred run length grows with the size of the problem instance.

The only concession (compared to Table 3.2) to obtain a consistent relationship is the run length of pla7397, at the expense of a slightly increased standard deviation. For each problem instance, the preferred run length and the corresponding results are listed in Table 3.4.

Analysis of RP

The restart mechanism that randomly perturbs the best solution found so far (RP) has two parameters: the run length T and p, which determines the amount of perturbation.

To determine the run length T, we aggregate the outcomes of the experiments for a given instance and T over the three values of p. Consequently, we have 30 experiments for every instance and T. The results are summarized as follows. For fl3795, the best results are obtained with the shortest run length and the results monotonously get worse for larger T. For pla7397, rl11849, usa13509, and d18512, the results are approximately u-shaped and the best results are obtained with medium run lengths ($3T^*$, $2T^*$, $3T^*$, and $3T^*$, respectively). The results for pla33810 are also approximately u-shaped and the best results are obtained with a run length of $4T^*$. The results for pla88900 monotonously improve for larger T. We conclude that there is

a clear relationship between the run length and the size of the problem instance: larger instances require relatively longer runs.

For each problem instance we identify the best value for p by aggregating over all values of T. We give a summary of the aggregated results. For instances pla7397 and rl11849, the best results are obtained with the smallest value of p, which corresponds to a relatively high amount of perturbation. The other instances prefer the medium or large value for p. Larger instances tend to prefer the largest value for p. The only exception is the smallest instance, which also requires the largest value for p. We conclude that the relationship with the instance size is not very strong. The acceptance probability tends to increase in the size of the instance, implying that larger instances need relatively less perturbation than smaller instances.

In order to obtain a consistent relation between the parameters and the size of the problem instance, two concessions are required. Compared to Table 3.2, we increase the run length for rl11849 at the expense of a higher average and standard deviation. For pla33810, parameter p is increased and the run length is decreased, resulting in a higher average. The selected values and the corresponding results are summarized in Table 3.4.

Analysis of CP

The commonality-preserving restart mechanism has three parameters: the run length T, the perturbation parameter ψ, and the weight parameter ρ. For each parameter, we seek a relation with the size of the problem instance by aggregating the results over all values of the other parameters.

First we determine the run length T. For each value of T, the results of the experiments are aggregated over all values of ρ and ψ. There are nine parameter sets with the same value of T and we compare the averages found by these nine parameter sets with the averages found by the nine parameter sets that correspond to different values of T. Consequently, for every value of T we have 90 computational experiments. The results are summarized as follows. For fl3795, pla7397, and rl11849, the shortest run length gives the best results and the results deteriorate monotonously for larger values of T. The results for usa13509, d18512, and pla33810 are u-shaped ($2T^*$, $3T^*$, and $3T^*$, respectively). For pla88900, the results get monotonously better for larger T. The longest run length gives the best results. There appears to be a clear relationship between the run length and the size of the problem instance. Small instances require a short run, while larger instances prefer a longer run.

In order to determine ψ, we aggregate the results over all values of T

and ρ. Consequently, we have 150 experiments for every given instance and ψ. The results are summarized as follows. For fl3795, $\psi = 1$, which corresponds to relatively low acceptance probabilities and thus a high amount of perturbation, gives the best results. For pla7397, all values of ψ give approximately the same results. For the remaining instances, rl11849, usa13509, d18512, pla33810, and pla88900, ψ should be small in general, which corresponds to relatively high acceptance probabilities, and thus a low amount of perturbation. We conclude that parameter ψ tends to decrease in the instance size, but is mainly constant; most instances prefer $\psi = 0.25$, which is equivalent to 2 to 6% of perturbation.

We attempt to find a relationship between ρ and the size of the problem instance by aggregating the results of the experiments over all values of T and ψ. As a result, we have 150 experiments for every given instance and ρ. The results are summarized as follows. For fl3795, pla7397, and rl11849, there is only a slight tendency that low and medium values for ρ give the best results. For usa13509, d18512, pla33810, and pla88900, there is only a slight tendency that medium and high values for ρ give the best results. Therefore, ρ appears to grow with the size of the problem instance. In general, the preferred value of ρ is either 0.98 or 1.0, indicating a tendency towards diversification, since all solutions in $\mathcal{L}$ have approximately the same weight.

A number of concessions are required in order to obtain a consistent relation between each parameter and the size of the problem instance. For each test instance, the selected parameter values and the corresponding results are listed in Table 3.4.

Comparing the strategies

Given the results in Table 3.4, we compare the single long run to the three strategies based upon multiple shorter runs.

For the majority of the test instances, we conclude that IR is a robust way to improve upon the performance of LR, as can be deduced from Table 3.4. Only pla85900 is better off with LR. The standard deviation is often considerably smaller than or equal to the one corresponding to the long run.

For all test instances, RP produces equivalent or better solutions on average than IR. For the largest test instance, pla85900, RP gives the same result as LR. The standard deviation is approximately the same as for IR. We conclude that RP is better than or equivalent to IR and LR.

strategy		fl3795	pla7397	rl11849	usa13509	d18512	pla33810	pla85900
LR	T	$24T^*$	$24T^*$	$24T^*$	$24T^*$	$24T^*$	$24T^*$	$24T^*$
	av	**5.0**	**2.0**	**3.0**	**1.2**	**0.9**	**3.6**	**2.6**
	sd	2.8	0.8	0.8	0.3	0.1	0.7	0.3
IR	T	T^*	$2T^*$	$2T^*$	$2T^*$	$6T^*$	$6T^*$	$6T^*$
	av	**1.5**	**1.4**	**1.9**	**1.1**	**0.9**	**3.3**	**2.7**
	sd	1.1	0.5	0.2	0.2	0.1	0.6	0.2
RP	T	T^*	$3T^*$	$3T^*$	$3T^*$	$3T^*$	$4T^*$	$6T^*$
	p	0.85	0.85	0.85	0.90	0.90	0.95	0.95
	av	**1.1**	**1.2**	**1.8**	**0.9**	**0.9**	**3.2**	**2.6**
	sd	1.2	0.3	0.4	0.1	0.1	0.5	0.3
CP	T	T^*	T^*	T^*	$2T^*$	$3T^*$	$3T^*$	$6T^*$
	ρ	0.98	0.98	0.98	0.98	0.98	0.98	1.0
	ψ	1.0	0.25	0.25	0.25	0.25	0.25	0.25
	av	**1.1**	**1.1**	**1.3**	**0.9**	**0.9**	**3.0**	**2.5**
	sd	1.3	0.3	0.3	0.1	0.1	0.4	0.3

Table 3.4: Consistent parameters for each of the three restart strategies.

For all test instances CP produces solutions which are on average equivalent to or better than the solutions produced by RP. The standard deviation is approximately the same as for IR. We conclude that CP is better than or equivalent to RP, IR and LR.

3.7 Conclusion

Chained Lin-Kernighan (CLK) is one of the most successful heuristics for symmetric TSP's. We illustrated that CLK tends to get stuck in an unpromising part of the search space. Instead of a single long run of CLK, we study three strategies that are based upon multiple shorter runs.

A common way to avoid getting stuck is by performing multiple independent runs. Multiple independent runs (IR) often drastically improve upon the performance of a single long run (LR), especially for the smaller instances. This is probably partially because the parameters of CLK were optimized for instances with 10,000 cities or more.

The major drawback of multiple independent runs is that information gathered during previous runs is discarded in the subsequent runs. We propose two alternative restart mechanisms that are based upon the following observation. Computational experiments indicate that good local optima generated by CLK have many edges in common. These so-called *commonalities* are interpreted as being "probably right". In an early paper, Lin and Kernighan [42] already found that good solutions to a TSP have many edges in common and suggested that these edges should be fixed during the remainder of the search. Instead of fixing these commonalities in subsequent runs, we propose two restart mechanisms that preserve many commonalities when constructing an initial solution for the subsequent run. We assume that the best solution found by each run is stored in a list $\mathcal{L}$.

A simple way to construct an initial solution that preserves many commonalities is by perturbing a copy of the best solution found so far. In our computational experiments, the amount of perturbation ranges between 5 and 15 percent of the edges and depends on the size of the problem instance. By perturbing only a small fraction of the edges, many commonalities are preserved. These restarts with random perturbation (RP) turn out to be better than or equivalent to LR and IR, in the sense that the average and standard deviation are often lower.

A more complex commonality-preserving restart mechanism (CP) constructs an initial solution for the next run by perturbing a copy of the best solution found so far in the following way. Each edge in the best solution

found so far is subjected to an acceptance trial. The probability that an edge is not removed depends upon the number of solutions in $\mathcal{L}$ that have the edge. When many solutions in $\mathcal{L}$ have the edge, the probability that it is part of the initial solution of the next run is high. For all test instances, CP produced a lower average and standard deviation than the other strategies.

When we analyzed the outcomes of our computational experiments, we noted that the very best results lead to inconsistent parameter values with respect to the size of the problem instance. By aggregation, the trend is determined for each parameter. Only minor concessions are required to obtain a consistent relation between most parameters and the size of the problem instance.

The three restart strategies based upon multiple shorter runs that we have considered have a standard deviation which is often significantly lower than that of LR. The standard deviation is approximately the same for the three restart strategies IR, RP and CP. We conclude that restarting improves upon the robustness and reduces the chance of getting stuck in an unpromising part of the search space.

Chapter 4

Whizzkids '96

4.1 Introduction and motivation

In 1996 and 1997, the Department of Mathematics and Computing Science at the Technische Universiteit Eindhoven organized a contest in cooperation with the IT-firm CMG and the newspaper De Telegraaf. The purpose of the contest was to increase the interest in mathematics and computer science, especially among high school students.

In 1996, the participants had to construct a newspaper delivery plan. We describe the assignment and the process of finding a hard problem instance. A number of local search heuristics that were employed to find upper bounds are discussed. We conclude with a commonality-preserving restart mechanism. The underlying tabu search heuristic is based upon well known ingredients, such as Osman's λ-interchange neighborhood, the 2-opt heuristic, and the Lin-Kernighan heuristic. Good solutions are obtained within minutes. Solutions of cost 1,183, which is the best known upper bound, are found within hours. Commonality-preserving restarts are slightly better than multiple independent runs and slightly better than a single long run.

4.2 Assignment

One of our tasks was to design a problem that is easily grasped by laymen, while computationally challenging. Many combinatorial optimization problems meet these requirements. Our choice fell upon the class of vehicle routing problems (VRP's). These problems are easily visualized and therefore suitable for a variety of solution approaches, including paper and pencil.

VRP's are in general NP-hard. Several optimization algorithms have been proposed for VRP's, see Cornuéjols & Harche [17], Christofides, Hadjiconstantinou & Mingozzi [15], Araque et al. [7], and Fisher [21]. In most cases, these exact algorithms are not capable of solving VRP's when the number of customers exceeds 50. As a result, one often resorts to local search heuristics, see for instance Gendreau, Laporte & Potvin [26].

Besides being computationally challenging, there were other requirements. For practical reasons, the size of the problem instance should not be too large. On the other hand, the problem size had to exceed the problem sizes that optimization methods are capable of solving. Furthermore, it was required that the problem is non-standard, such that it was not easy to adapt existing route planning software.

In order to avoid difficulties when calculating and comparing distances, we do not use the Euclidean norm, but use Manhattan distances. The problem with the Euclidean norm is that, according to Garey, Graham & Johnson [23], there is no known algorithm that compares sums of square roots in polynomial time. The distance between two customers is defined as the difference between the x-coordinates plus the difference between the y-coordinates. As a result, distances are symmetric. We proposed the following vehicle routing problem.

Definition 4.2.1. *(Whizzkids '96). Four newspaper delivery boys deliver newspapers to 120 subscribers in Manhattan. The addresses are located at integer points in the plane. They start at the same time from a single depot. The problem is to cluster and route the addresses and produce a newspaper delivery plan. The primary objective is to deliver the last newspaper as early as possible. The secondary objective is to minimize the average delivery time of the newspapers.*

Formally, the Whizzkids '96 problem resembles a m-TSP with minmax objective and symmetric Manhattan distances. As the TSP is a subproblem, Whizzkids '96 is NP-hard. We use the secondary objective to distinguish between solutions that are submitted by the participants and have the same primary objective.

While in classical VRP's it is often required that all tours end at the depot, we deliberately chose to drop this requirement. As a result, the Whizzkids '96 problem is concerned with paths rather than tours. In the literature, relatively little attention has been paid to the min-max objective, although an adaptive memory programming heuristic for the minmax m-TSP can be found in a paper by Golden et al. [32].

4.2.1 Finding a hard problem instance

We generated a number of random problem instances with 120 addresses. The following ideas were used to generate hard instances:

- The addresses are clustered.
- The number of clusters and the number of newspaper delivery boys are relatively prime.
- The location of the depot is chosen in such a way that, in good solutions, each cluster is likely to be visited by at least two boys.

The locations of the addresses and the depot were generated at random in the plane with integral x and y coordinates between 0 and 500. First, we generated rectangular clusters. The clusters are mutually non-overlapping, and positioned at random locations. Next, the locations of the addresses were drawn uniformly at random from the clusters.

In order to get an idea of the hardness of each instance, we implemented a simulated annealing heuristic. Only the primary objective was taken into account. We experimented with various cooling schedules, resulting in running times ranging from a couple of minutes to several hours on a Sun SPARC-5. For each instance, we performed many short and longer runs, and analyzed the cost distribution of the solutions. We preferred cost distributions that are non-smooth and had a thin tail to the left, indicating that the best solutions are not found often, and a thick tail to the right, indicating that the chance of getting trapped in solutions of mediocre quality is high. To assess the quality of the upper bounds obtained by simulated annealing, we calculated a number of simple lower bounds for the problem instances, such as the maximum distance between the depot and any address and a quarter of the weight of a minimum-weight spanning tree. A large gap between these simple lower bounds and upper bounds also gave an idea of the hardness of the instance.

Assignment. We chose a problem instance consisting of three clusters. The problem instance is given in Figure 4.1. The large dot represents the depot, the other dots correspond to the 120 addresses. The coordinates can be obtained via our website,

`http://www.win.tue.nl/whizzkids/1996/index.html`.

The chosen instance was hard for our simulated annealing heuristic, and we expected that it would also be hard for other approximation methods.

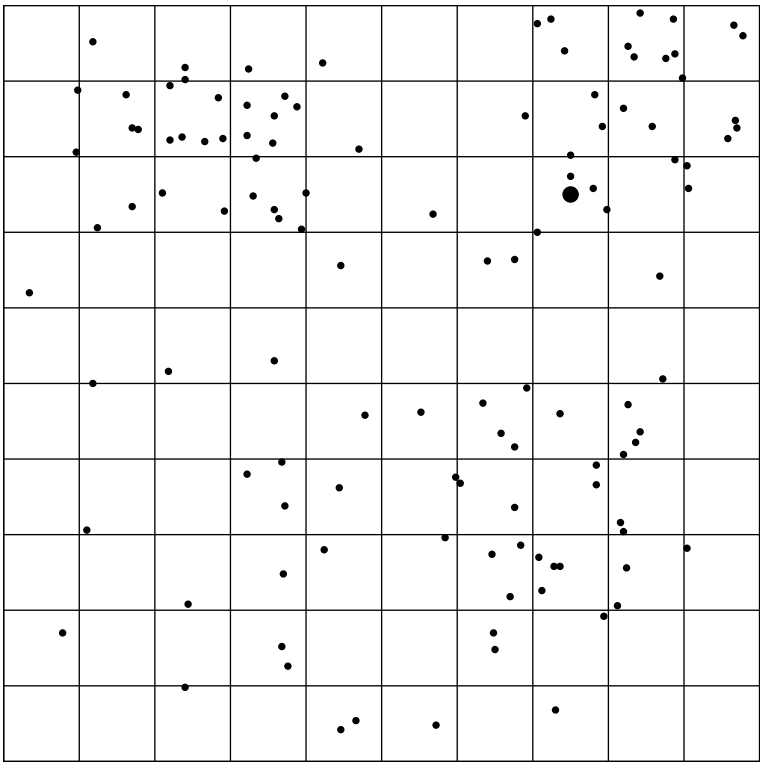

Figure 4.1: The Whizzkids '96 assignment. The large dot represents the depot, the other dots correspond to the 120 addresses.

4.3 Results

We discuss our upper bounds and lower bounds and the results of the participants. A brief description of the applied techniques is given.

4.3.1 Staff

First, we describe the simulated annealing heuristic that was developed to help identify hard problem instances. The initial solution is constructed by a best insertion heuristic that processes the addresses in a random order. The neighborhood is a mixture of insert moves and tail swaps such that at least one longest path is affected; 9 out of 10 moves are insert moves. Every now and then, a path improvement heuristic, based upon 2-opt, was applied to the current solution. Only the primary objective was taken into account. Using a geometric cooling schedule, our simulated annealing heuristic was

able to find solutions of cost 1,183, but it took an excessive amount of time on a Sun SPARC-5.

Bas Aarts and Pascal Coumans (Philips Research Laboratories) implemented a simulated annealing heuristic with balanced paths in the sense that all paths contain about the same number of addresses. After 531 runs of 5 hours each, the best known solution was found: a primary objective of 1,183 and a secondary objective of 558.82.

Trivial lower bounds include the maximum distance between the depot and any address (626) and a quarter of the weight of a minimum-weight spanning tree (1,007). Cor Hurkens and Arjen Vestjens used a number of more advanced techniques, including Lagrange relaxation and branch-and-cut, to obtain a lower bound of 1,160. In the mean time, André Rohe and Sanjeeb Dash of Rice University improved the lower bound to 1,170.

4.3.2 Participants

We received over 900 solutions in two categories, general and professional. There were 18 solutions in which the last newspaper was delivered at time 1,183. In order to distinguish between them, we calculated the average delivery time of the newspapers. There were two solutions with a minimum secondary objective of 558.82. The most successful approaches were based on local search heuristics, but paper-and-pencil and interactive approaches also proved to be competitive. We give a brief description of some heuristics that were employed to find approximate solutions.

- Éric Taillard adapted his tabu search heuristic for capacitated vehicle routing problems [67]. This heuristic operates on the set of partial solutions and has a penalty term in the objective. A partial solution consists of up to four paths and possibly a number of unvisited addresses. The penalty is the sum of the lengths of the edges from the depot to the unvisited addresses. The initial solution is obtained by a modified version of the *savings heuristic* due to Clarke & Wright [16] for VRP's. By memorizing and re-using a number of good paths and some user-interaction, Taillard found solutions of cost 1,183.

- A team of computer science students developed a graphical support tool. The four paths are displayed on the screen and the user can modify the paths by moving addresses between them. The length of each path is updated automatically.

- Surprisingly, by using a paper-and-pencil approach, a 14 year old high school student was able to beat most of the professional participants.

Other local search heuristics known from the literature that could be adapted to the Whizzkids '96 problem include Osman's simulated annealing algorithm [52], and Taburoute (Gendreau, Hertz & Laporte [25]).

4.4 Tabu search

We present a tabu search heuristic that is based upon techniques that are well known from the literature: Osman's λ-interchange neighborhood, the 2-opt heuristic, and the Lin-Kernighan heuristic. In this section, we study the finite-time behavior of our tabu search heuristic. In the next section, we introduce a commonality concept for Whizzkids '96, together with a commonality-preserving restart mechanism.

4.4.1 Neighborhood

The neighborhood that we employ is based upon Osman's λ-interchange neighborhood [52]. Osman interchanges two subsets of addresses of two different paths such that the cardinality of each subset is at most λ. We put $\lambda = 1$. As a result, we have two types of moves: insert moves and swap moves. An insert move consists of removing an address from its path and inserting it into the other path. A swap move exchanges two addresses that are in different paths. The insert neighborhood contains $\mathcal{O}(n^2)$ neighbors and is significantly smaller than the swap neighborhood, which contains $\mathcal{O}(n^4)$ neighbors, where n is the number of addresses.

4.4.2 Tabu list

In order to prevent cycling and to direct the search to unexplored regions, the neighborhoods are restricted by a tabu list τ. The tabu list determines whether or not we are allowed to insert an address into a path. After we remove an address from a path, we are not allowed to re-insert it for the next l iterations, where l denotes the tabu tenure. The tabu list is implemented by maintaining for each address a and newspaper boy b a time $t_{a,b}$, which denotes the iteration number at which address a was last removed from b's path. Initially, $t_{a,b} = -\infty$, for all addresses a and newspaper boys b. The iteration number is initially 0, and is increased by one after each effectuated move. When the current iteration number is smaller than $t_{a,b} + l$, it is not allowed to insert a into b's path, i.e, the move is tabu.

4.4.3 Neighborhood search strategy

Due to the min-max criterion, there exist many solutions with the same cost. This effect is amplified by using Manhattan distances: many addresses can be visited without any additional cost. For each pair of points (x_1, y_1) and (x_2, y_2), every third point (x_3, y_3) with $x_1 \leq x_3 \leq x_2$ and $y_1 \leq y_3 \leq y_2$ can be traversed with no additional cost. We adopt a greedy neighbor search strategy that takes this observation into account by defining the cost of a move as 1,000 times the cost difference between the longest paths plus the sum of the cost differences between the two affected paths. A negative cost implies an improvement, a positive cost implies a deterioration. Notice that we are only concerned with the primary objective.

Our neighborhood search strategy is centered around randomized minimum cost insertions, which are defined as follows. For a given address and a path, each possible insertion point is considered. The cost of inserting the address at a given position in the path is calculated in constant time. The address is inserted at a randomly selected insertion point of lowest cost.

In each iteration, either a best insert move or a best swap move is effectuated, depending on which one is better. A *best insert move* is determined in the following way. For each address, a minimum cost insertion point in another path is determined. By processing the addresses in a random order and keeping track of the best insert move found so far, a random best insert move is obtained. A *best swap move* is determined in a similar fashion. For each pair of addresses in different paths, a minimum cost insertion point in the other path is determined. Again, the addresses are processed in a random order, resulting in a random best swap move. When a move is tabu, but the resulting solution is better than the best so far, the tabu status is overruled. Otherwise, the cost of the move is defined as ∞.

We require that each move always involves two different paths. Moreover, by requiring that each move involves at least one longest path, the neighborhood is significantly reduced.

4.4.4 Path improvement heuristic

After every move, a path improvement heuristic (PIH) is applied to both affected paths. We either use the 2-opt heuristic or the Lin-Kernighan heuristic for that purpose.

In order to apply a path improvement heuristic, each path is transformed into a tour that begins and ends at the depot by adding a dummy address X with the property that the distance from the depot to X is 0, while the dis-

PIH	av	sd
none	1,621	87
2-opt	1,597	87
LK	1,566	78

Table 4.1: Cost of the initial solution obtained with randomized best insertion and a path improvement heuristic applied to each path.

tance from any address to X is M, for some large integer M. Consequently, each tour has at least one edge of length M. When X is not adjacent to the depot, the tour must contain two edges of length M. For each path, 2-opt is repeated until no more improvements are found. As the path is taken as the initial solution, 2-opt cannot produce a longer path.

Alternatively, we can use the Lin-Kernighan heuristic. We apply a single iteration of the Chained Lin-Kernighan (CLK) implementation of Applegate, Bixby, Chvátal & Cook, which is part of their Concorde project [3] to solve large TSP's. A single iteration of CLK corresponds to one invocation of the Lin-Kernighan heuristic. The initial solution is generated by the Quick-Borůvka constructive heuristic. As a result, CLK can produce a path which is worse than the original one. In that case, we keep the original path.

4.4.5 Results

We performed a number of computational experiments in which the initial solution is obtained by the following randomized best insertion heuristic. Initially, each paperboy path is empty. For each address, the cost of insertion at a best possible place in each path is computed. The address is inserted into a path that gives the lowest cost. The addresses are processed in a random order. When all addresses have been processed, a path improvement heuristic is invoked for each path.

We generated 250 initial solutions with and without a path improvement heuristic. The average (av) and standard deviation (sd) are listed in Table 4.1. We conclude that the Lin-Kernighan heuristic (LK) gives better results than the 2-opt heuristic, and the 2-opt heuristic gives better results than when no path improvement heuristic is employed. There are no significant differences in the running times.

For tabu search, we performed a number of computational experiments in which we put a maximum T on the number of successive non-improving iterations. A number of different values for T, values for the tabu tenure

l, and path improvement heuristics are considered. The experiments are repeated ten times. The average solutions quality (av) and the standard deviation of the solution quality (sd) are listed in Table 4.2. The experiments were carried out on a Pentium III/450 MHz PC. The running times range from 20 seconds for $T = 50$ to 27,000 seconds for $T = 80{,}000$. Interestingly, the running times are approximately the same for each of the three path improvement heuristics, implying that tabu search spends most of its time exploring the neighborhood. We conclude that Lin-Kernighan gives better results than 2-opt, and 2-opt gives better results than when no path improvement heuristic is employed. The best tabu tenure is $l = 20$, for each path improvement heuristic. Furthermore, it seems that, for each path improvement heuristic and l, the results improve monotonously for larger T.

From now on, we use Lin-Kernighan as the path improvement heuristic and use a tabu tenure l of 20. In the next section, we attempt to improve the performance of tabu search by restarting frequently in such a way that many commonalities are preserved.

4.5 Commonality-preserving restarts

Each solution to the Whizzkids '96 problem is a combination of assignment and sequencing decisions. The neighborhood of our tabu search heuristic is primarily concerned with changing the assignment of addresses to paperboys. A path improvement heuristic is in charge of the sequencing decisions. Therefore, it is natural to restrict our commonality concept to assignment decisions.

Definition 4.5.1. *Suppose that $\mathcal{L}$ is a multiset of solutions to the Whizzkids '96 problem. A pair (u, v) consisting of two distinct addresses u and v is called a* commonality of $\mathcal{L}$ *when in all $L_i \in \mathcal{L}$ address u and v are assigned to the same paperboy.*

Let $c(L_i)$ denote the cost of solution L_i, which is defined as the time at which the last newspaper is delivered. After an initial number of independent runs, a number of restarts are performed. The best solution found by every run is inserted into $\mathcal{L}$. After k runs, $\mathcal{L} = (L_1, \ldots, L_k)$ such that $c(L_i) \leq c(L_j)$ whenever $i < j$.

An initial solution L_0 for the next run is obtained by perturbing L_1, the best solution found so far. Initially, L_0 is equal to L_1. The four paths of L_0 are processed one after the other. Notice that the edges in a path capture the sequencing decisions as well as the assignment decisions. In order to

	l	10		20		30	
PIH	T	av	sd	av	sd	av	sd
	50	1,364	51	1,373	36	1,372	56
	100	1,331	39	1,336	31	1,334	30
	250	1,328	66	1,304	27	1,325	36
	500	1,304	60	1,284	31	1,295	48
none	1,000	1,263	40	1,278	29	1,266	26
	5,000	1,244	27	1,248	17	1,228	19
	10,000	1,248	42	1,245	19	1,230	16
	20,000	1,230	34	1,219	20	1,229	17
	40,000	1,220	17	1,214	16	1,226	19
	80,000	1,206	14	1,199	10	1,212	9
	50	1,332	51	1,304	43	1,336	39
	100	1,314	36	1,307	44	1,306	50
	250	1,273	38	1,268	38	1,278	35
	500	1,262	37	1,249	32	1,254	29
2-opt	1,000	1,246	22	1,254	24	1,246	17
	5,000	1,254	38	1,228	16	1,235	16
	10,000	1,213	17	1,215	15	1,224	17
	20,000	1,214	17	1,203	14	1,213	6
	40,000	1,205	14	1,203	12	1,204	9
	80,000	1,197	9	1,193	9	1,202	8
	50	1,281	43	1,269	30	1,300	44
	100	1,247	42	1,259	31	1,273	36
	250	1,243	27	1,242	36	1,256	21
	500	1,248	33	1,246	32	1,258	24
LK	1,000	1,223	16	1,227	17	1,239	31
	5,000	1,206	16	1,218	15	1,215	14
	10,000	1,204	17	1,206	11	1,206	7
	20,000	1,202	16	1,195	9	1,201	9
	40,000	1,194	6	1,192	5	1,196	5
	80,000	1,193	16	1,190	3	1,195	7

Table 4.2: Results for ten runs of the tabu search heuristic for various parameters.

preserve this sequencing information, our restart mechanism operates in terms of these edges. For each path of L_0, the edges are traversed in path order. The *commonality class* $\mathcal{C}(u, v)$ of a pair (u, v) of addresses is defined

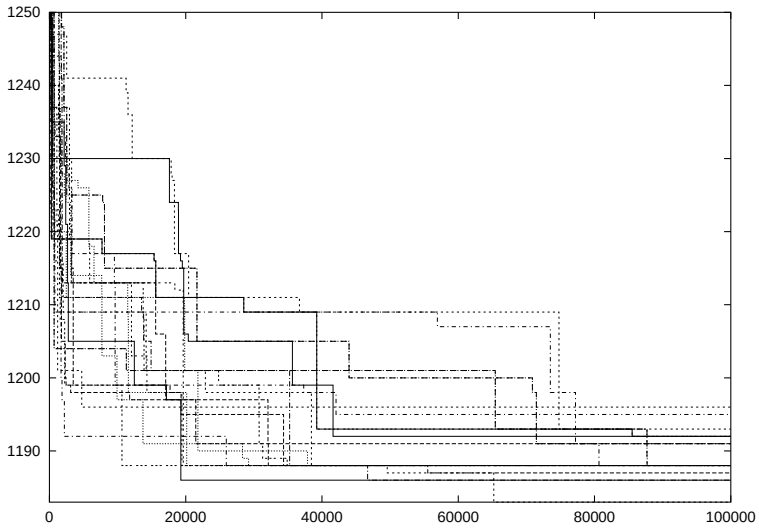

Figure 4.2: The trajectories of 30 independent runs of tabu search. The solution quality on the y-axis is given as a function of the iteration number on the x-axis.

as the number of solutions in $\mathcal{L}$ that have u and v assigned to the same path. For each edge $\{u, v\}$ in L_0, the acceptance probability is defined in terms of the fraction of solutions in $\mathcal{L}$ that have u and v assigned to the same path:

$$p_{\text{acc}}(u, v) := p_{\min} + (p_{\max} - p_{\min}) \frac{\mathcal{C}(u, v)}{k},$$

where $p_{\min}$ and $p_{\max}$ are given thresholds such that $0 \leq p_{\min} \leq p_{\max} \leq 1$. When $\{u, v\}$ is *accepted*, it stays in L_0. When $\{u, v\}$ is *rejected*, either address u or address v is relocated to another path. For both addresses, the best insertion point in a different path is determined. When more than one best insertion place exists, ties are broken at random. In case the resulting cost of relocating u is less than or equal to v's relocation cost, u is relocated; otherwise, v is relocated. Let $s(u)$ denote the successor of address u in a path. When u is removed, the next edge to consider is $\{v, s(v)\}$. When v is removed, the next edge to consider is $\{s(v), s(s(v))\}$. After applying the Lin-Kernighan heuristic to each path, the construction of the initial solution is complete.

strategy	av	sd	best
IR	1,187.15	0.88	1,186
CP/0.80	1,187.15	2.54	1,183
CP/0.85	1,186.55	1.82	1,183
CP/0.90	1,187.00	1.56	1,183
CP/0.95	1,186.45	1.79	1,183
LR	1,187.20	2.02	1,183

Table 4.3: The results of a single long run, twelve independent runs, and two independent runs followed by ten restarts of the commonality-preserving restart mechanism for various values of $p_{\max}$. The run length T is 20,000, Lin-Kernighan is used as path improvement heuristic, and the tabu tenure l is 20.

4.5.1 Results

In order to determine whether or not tabu search tends to get stuck, we perform 30 independent long runs with $l = 20$ and Lin-Kernighan as path improvement heuristic. The corresponding trajectories in cost space are given in Figure 4.2. As there are often substantial improvements after hours of computation, we conclude that tabu search does not get stuck early in the search process. Nevertheless, we are interested whether or not multiple shorter runs give better results than a single long run. Therefore, we must determine a proper run length. The total number of iterations at which the independent runs slow down significantly appears to be around 40,000. Furthermore, after 40,000 iterations, the majority of the independent runs have no improvement for the subsequent 15,000 to 20,000 iterations. As our stop criterion consists of putting a maximum T to the number of subsequent non-improving iterations, we will use $T = 20{,}000$ in our experiments with multiple shorter runs.

We carried out a number of computational experiments in which we compare two independent runs followed by ten commonality-preserving restarts (CP) to twelve independent runs (IR) and a single long run (LR). We experimented with a number of different values for $p_{\max}$. The experiments were repeated 20 times. For each strategy, the average (av), standard deviation (sd), and the best solution found by the 20 computational experiments are listed in Table 4.3. Each entry of the form CP/x corresponds to commonality-preserving restarts with $p_{\min} = 0$ and $p_{\max} = x$. A single long run is simulated by having a single independent run followed by

eleven restarts with $p_{\min} = p_{\max} = 1.0$, which corresponds to no perturbation. Although the average results of the commonality-preserving restarts are slightly lower than or equal to the average results of the independent runs and the long run, the standard deviation of the independent runs is significantly smaller than that of the other strategies. We conclude that commonality-preserving restarts and the single long run are less robust than the independent runs. Given that the averages are approximately the same for all strategies, the relatively high standard deviation of the commonality-preserving restarts and the single long run imply that the best solution found by the 20 experiments is likely to be better than that of the independent runs. The best solution found by the 20 experiments with a single long run and each of the four commonality-preserving strategies were able to find solutions of cost 1,183, while the independent runs could do no better than 1,186.

4.6 Conclusion

We described a tabu search heuristic that is based upon standard ingredients such as Osman's λ-interchange neighborhood and the Lin-Kernighan heuristic. The neighborhood of tabu search is mainly concerned with the assignment of addresses to paperboys. The Lin-Kernighan heuristic is employed to deal with the sequencing decisions for each paperboy. Frequently, solutions of cost 1,183, which is the best known upper bound, are found within hours.

In an attempt to improve the performance of the tabu search heuristic, we proposed a commonality-preserving restart mechanism. We compared two independent runs followed by ten commonality-preserving restarts to twelve independent runs and to a single long run. The average results of the commonality-preserving restarts are slightly lower than or equal to the average results of the independent runs and the long run. The independent runs are more robust than the commonality-preserving restarts and the single long run in the sense that the standard deviation is significantly smaller. On the other hand, the cost of the best solution found by 20 experiments with the commonality-preserving restart mechanism and the long run was 1,183, while the cost of the best solution found by 20 experiments with the independent runs was 1,186.

During the numerous computational experiments, a number of different solutions with cost 1,183 were found. In Figure 4.3, the union of these twelve solutions is given. It appears that only a small number of alternative edges

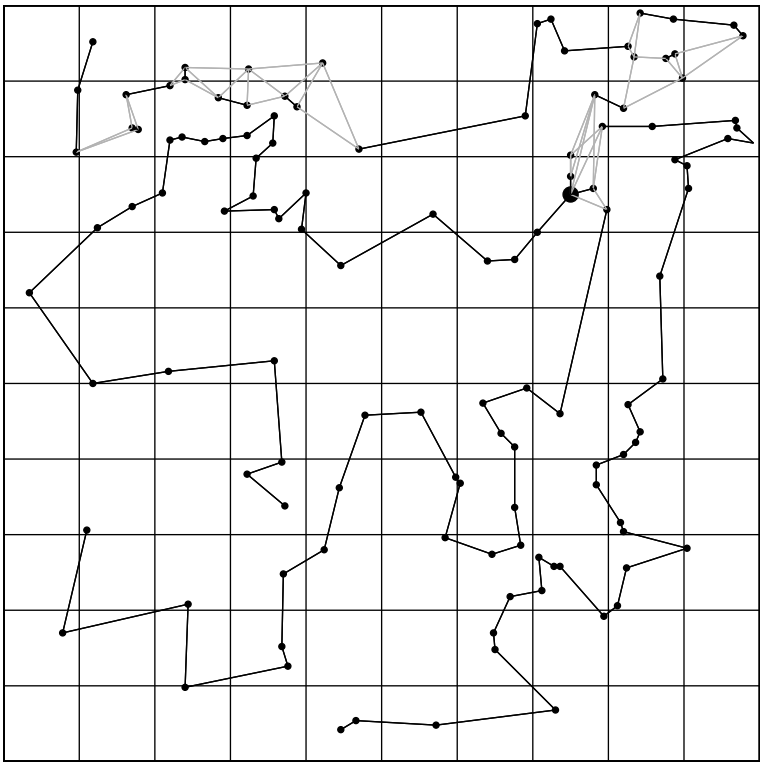

Figure 4.3: Twelve different solutions of cost 1,183. The black edges are common to all solutions, while the grey edges are not.

can be part of solutions of cost 1,183. Instead of commonality-preserving restarts, one could fix the commonalities and optimize over the other building elements. Given the abundance of commonalities, a straightforward explicit enumeration could do the job.

Chapter 5

Job shop scheduling

5.1 Introduction

Tabu search (Glover & Laguna [30]) has proven to be a very effective local search heuristic for the job shop scheduling problem (Nowicki & Smutnicki [51], Ten Eikelder, Aarts, Verhoeven & Aarts [19], Taillard [68]). It employs a greedy neighborhood search strategy and a mechanism to prevent getting trapped in local optima. Tabu search is, in general, not known to converge to a global optimum. In its elementary form, tabu search often seems to get stuck in some part of the search space, meaning that no further improvements will be found when it is given more time. When we have reasons to believe that an optimum solution has not yet been found, another mechanism is needed to direct the search to other regions. This mechanism is known as *diversification* (Glover & Laguna [30]).

One often resorts to multiple independent runs in order to ensure that a large portion of the search space is visited. This simple restart mechanism starts from a randomly generated solution and turns out to be very robust in the sense that the variance of the quality of the best solution found during a fixed number of independent runs is relatively small. Indeed, independent runs can be viewed as a simple diversification mechanism. Unfortunately, all efforts of previous runs are wasted, which results in a poor efficiency.

Another common way to restart tabu search is by means of a backtrack mechanism. During the search, a list of elite solutions is maintained. When tabu search terminates, it is restarted with a solution on the list. Appropriate measures are taken to ensure that the trajectory followed is different from the one before. Backtracking is a simple example of a so-called *intensification* mechanism (Glover & Laguna [30]). In this way, the surroundings of

good solutions are thoroughly explored. Good results can be obtained with this approach, see for instance Nowicki & Smutnicki [51] and Ten Eikelder, Aarts, Verhoeven & Aarts [19].

When good solutions that are found during independent runs are compared, it turns out that they have many attributes in common. When an attribute is common to many good solutions, it is interpreted as being "probably right". We present a probabilistic restart mechanism for tabu search that exploits this feature. During the execution of our algorithm, a list $\mathcal{L}$ of elite solutions is maintained. The main idea is that a new start solution is constructed by perturbing the best solution found so far. When an attribute of the best solution found so far is common to many good solutions in $\mathcal{L}$, it will become part of the initial solution of the next run with high probability. The amount of perturbation is a parameter of the restart mechanism. Perturbing all attributes corresponds to multiple independent runs and perturbing a small number of attributes resembles backtracking. In this way, a balance between diversification and intensification is obtained. We perform a number of computational experiments in which the running time is limited to 15 minutes on a Pentium II 333 MHz/Linux machine. We conclude that our restart mechanism is more robust than backtracking, more effective than independent runs, but slightly less effective than backtracking.

This chapter is organized as follows. The job shop scheduling problem is introduced in Section 5.2. Section 5.3 describes the underlying tabu search algorithm upon which the restart mechanism will be built. The restart mechanism is described in Section 5.4 and computational results can be found in Section 5.5.

5.2 Job shop scheduling

In the *job shop scheduling problem* we are given a set $\mathcal{J}$ of n jobs, a set $\mathcal{M}$ of m machines and a set $\mathcal{O}$ of N operations. For each operation $u \in \mathcal{O}$ there is a processing time $p_u \in \mathbf{Z}^+$, a unique machine $M_u \in \mathcal{M}$ on which it must be processed, and a unique job $J_u \in \mathcal{J}$ to which it belongs. Furthermore, a binary precedence relation $\prec$ is given that decomposes $\mathcal{O}$ into chains, one for each job. The problem is to find a start time s_u for every operation $u \in \mathcal{O}$ such that the *makespan*, defined as $C_{\max} = \max_{u \in \mathcal{O}}\{s_u + p_u\}$, is minimized subject to the following constraints:

$$
\begin{array}{ll}
s_u \geq 0 & \text{for all } u \in \mathcal{O}, \\
s_v \geq s_u + p_u & \text{for all } u, v \in \mathcal{O} \text{ with } u \prec v, \text{ and} \\
s_v \geq s_u + p_u \text{ or } s_u \geq s_v + p_v & \text{for all } u, v \in \mathcal{O} \text{ with } M_u = M_v.
\end{array}
$$

The first constraint implies that no machine is available before time 0, the second constraint accounts for the precedence relation $\prec$ such that no operation will start before its predecessor in the chain has finished, and the last constraint is the machine capacity constraint which stipulates that every machine can process at most one operation at a time.

5.2.1 Solution representation

An instance of the job shop scheduling problem can be represented by a *disjunctive graph* (Roy & Sussmann [62]). This is a vertex-weighted mixed graph $G = (\mathcal{O}, A, E)$ where the arc set A consists of *job arcs* connecting consecutive operations of the same job, and the edge set E consists of edges connecting distinct operations that must be executed on the same machine. Each vertex $u \in \mathcal{O}$ has a weight p_u.

Every solution σ to the job shop scheduling problem induces a unique acyclic digraph $G(\sigma)$ obtained from G by replacing every edge $\{u, v\} \in E$ by a *machine arc* (u, v) or (v, u). The earliest start time of operation u is equal to the length of a longest path in $G(\sigma)$ up to and excluding u. The cost of the solution, denoted by $c(\sigma)$, is equal to the length of a longest path in $G(\sigma)$. Our set of feasible solutions $\mathcal{S}$ is the set of acyclic digraphs that can be obtained from G. These acyclic digraphs correspond to the set of left-justified solutions.

From now on, no distinction will be made between a solution and its corresponding digraph. We introduce the following additional notation. In an acyclic digraph $G(\sigma)$, jp_u is the immediate job predecessor of an operation $u \in \mathcal{O}$, js_u is its immediate job successor, mp_u is its immediate machine predecessor, and ms_u is its immediate machine successor, whenever they exist.

5.3 Elementary tabu search

Tabu search is one of the most effective local search heuristics for the job shop scheduling problem. A detailed survey of local search heuristics applied to the job shop scheduling problem is given by Vaessens, Aarts & Lenstra [71]. We will describe an elementary tabu search heuristic that is based upon ideas from Nowicki & Smutnicki [51], Taillard [68] and Ten Eikelder, Aarts, Verhoeven & Aarts [19]. It does not incorporate an intensification or diversification mechanism, but it is designed to fit into our restart framework, which will be introduced later.

5.3.1 Neighborhood

Most neighborhoods for the job shop scheduling problem are based on reversing a machine arc on a longest path in the digraph. Reversing an arc on a longest path always results in a feasible solution, and reversing an arc that is not on a longest path will never improve the solution and can even lead to infeasibility (Van Laarhoven, Aarts & Lenstra [39]).

The neighborhood function we use is the one proposed by Nowicki & Smutnicki [51], which is based on the neighborhoods defined by Matsuo, Suh & Sullivan [44]. 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* when 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. Matsuo, Suh & Sullivan [44] noted that reversing an internal arc cannot decrease the makespan. Nowicki and Smutnicki observed that the same applies to machine arcs from the first operation of the first block to an internal operation and for machine arcs from an internal operation to the last operation of the last block. This leads to the following neighborhood function. For a given solution σ consider a single critical path $(B_1, \ldots, B_\lambda)$, where each B_i denotes a block. The neighborhood $\mathcal{N}(\sigma)$ of σ consists of all solutions that can be obtained by swapping the last two operations of B_1, the first two or the last two operations of $B_2, \ldots, B_{\lambda-1}$, or the first two operations of B_λ.

Each neighbor of σ can be obtained by reversing a machine arc $(u, v) \in \sigma$. Such a reversal is called a *move*. From now on no distinction will be made between a move (u, v) and the resulting neighbor.

5.3.2 Tabu list and aspiration levels

An important characteristic of tabu search is that the neighbor selection mechanism is influenced by a so-called *tabu list*. The purpose of a tabu list is to prevent cycling, i.e., moving to solutions that were visited before. This can be achieved by maintaining a list of the most recently visited solutions and testing for each neighbor if it is on the list or not.

Unfortunately, this involves comparing complete solutions, which is very time-consuming. Instead we maintain a tabu list τ consisting of machine arcs. After reversing a machine arc (u, v) all machine arcs originating from v, including (v, u), are appended to τ, which means that they cannot be reversed for some period of time. This period of time is called the *tabu tenure* and it is denoted by l. The tabu tenure must be large enough to

avoid cycling and small enough not to forbid too many moves.

It is known that randomizing the tabu tenure reduces the chance of getting trapped in a long cycle. In our algorithm we set $l = \max\{\bar{\mathcal{N}} + X, 1\}$, where $\bar{\mathcal{N}}$ is the average number of neighbors of the solutions visited so far and X is a random number between 0 and 5. The value of the tabu tenure is updated every $l + 2$ iterations.

For each operation $v \in \mathcal{O}$ let t_v denote the last iteration number at which a machine arc $(\cdot, v)$ was reversed to $(v, \cdot)$. Initially we put all $t_v = -\infty$. Each time that a machine arc $(\cdot, v)$ is reversed the value t_v is set to the current iteration number. This information is used to determine whether a move is allowed or not. Reversing a machine arc $(v, \cdot)$ is not allowed at iteration k when $k \leq t_v + l$. When this is the case the move is called *tabu*.

A drawback of the approach sketched above is that it can happen that a certain move is tabu although the resulting solution was never visited before. A mechanism for overruling the tabu status of moves is provided by *aspiration levels*. For every operation $v \in \mathcal{O}$ let α_v denote the threshold on the cost of any move $(v, \cdot)$ for overruling the tabu status. When the corresponding neighbor's cost are less than this threshold, the tabu status is overruled. A common approach is to set α_v equal to the makespan of the best solution found so far, for every operation $v \in \mathcal{O}$. This implies that the tabu status of any move $(v, \cdot)$ is overruled when the resulting solution's cost is less than the cost of the best solution found so far.

5.3.3 Neighborhood search strategy

A tabu search heuristic usually moves from a solution to a neighboring solution with minimum cost. Finding a minimum-cost neighbor of a solution σ is a time consuming process with complexity $\mathcal{O}(|\mathcal{N}(\sigma)|N)$, where $|\mathcal{N}(\sigma)|$ is the size of σ's neighborhood and N the number of operations. Instead of calculating the exact cost of a neighbor we use Taillard's $\mathcal{O}(1)$ time lower bound [68].

For every operation $u \in \mathcal{O}$, r_u denotes the *head* of operation u, which is defined as the length of a longest path up to and excluding u, and q_u denotes the *tail* of operation u, defined as the length of a longest path originating from and excluding u. These quantities determine which operations are on a longest path in the digraph: an operation u is on a longest path, i.e., *critical*, when $r_u + p_u + q_u = C_{\max}$. Taillard's lower bound on the makespan of the neighboring solution obtained by reversing a machine arc (u, v) on a longest path is calculated as follows. When an arc (u, v) on a longest path is reversed, the heads and tails of u and v can be updated in constant time

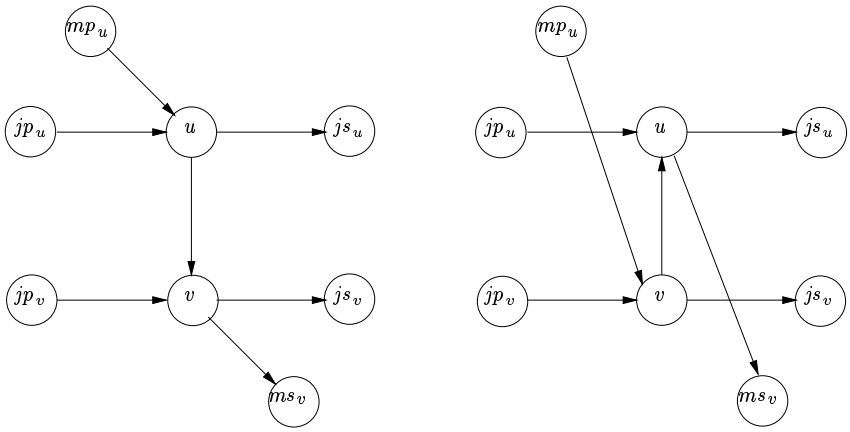

Figure 5.1: Before and after reversal of machine arc (u, v).

(see Figure 5.1):

$$\begin{aligned} r'_v &= \max\{r_{jp_v} + p_{jp_v}, r_{mp_u} + p_{mp_u}\}; \\ r'_u &= \max\{r_{jp_u} + p_{jp_u}, r'_v + p_v\}; \\ q'_u &= \max\{p_{js_u} + q_{js_u}, p_{ms_v} + q_{ms_v}\}; \\ q'_v &= \max\{p_{js_v} + q_{js_v}, p_u + q'_u\}. \end{aligned}$$

The value $\Lambda_{u,v} = \max\{r'_u + p_u + q'_u, r'_v + p_v + q'_v\}$ is a lower bound on the new makespan; when the new longest path passes through u or v, the new makespan is equal to $\Lambda_{u,v}$.

Using lower bounds instead of exact neighbor costs has consequences for the use of aspiration levels. Let UB denote the cost of the best solution found so far. When it must be decided whether or not to overrule the tabu status of a move it is not appropriate to test whether the lower bound on the neighbor cost is less than UB. Instead, for every operation v the threshold α_v is defined as the minimum of UB and the lowest lower bound that was calculated for any selected reversal $(\cdot, v)$. This value is interpreted as the so-called best promise that a move $(\cdot, v)$ ever made when evaluating the lower bound on its cost.

A drawback of using lower bounds is that the neighbor with the lowest lower bound is not necessarily a minimum-cost neighbor. The following neighborhood search strategy strikes a balance between accuracy and speed. For every neighbor (u, v) the lower bound $\Lambda_{u,v}$ is calculated. The cost of every tabu move (u, v) that is not overruled by the corresponding aspiration level, i.e., $\Lambda_{u,v} \geq \alpha_v$, is set to $B + t_v$, where $B \gg UB$. In this way, moves

that were recently made tabu are less attractive than old tabu moves and tabu moves are always less attractive than non-tabu moves. Neighbors that have their tabu status overruled maintain their lower bound cost. For every non-tabu move, including the overruled tabu moves, for which the lower bound is smaller than the cost of the current solution, the exact cost is calculated. The move with minimum cost is selected and in case of a tie between the minimum lower bound and the minimum exact cost we prefer the latter one. By convention the minimum over an empty set is ∞.

As the average solution quality improves, the number of neighbors with lower cost than the current solution decreases. Hence, the number of exact cost calculations decreases during the course of the algorithm, reducing the time required for an iteration.

When the selected move (u, v) is tabu, its tabu status is *relaxed* by putting $t_u = -\infty$. Furthermore, as the aspiration level α_v of an operation v is defined as the minimum of UB and the lowest lower bound that was calculated for any selected reversal $(\cdot, v)$, the aspiration level of v is updated only when the calculated cost of (u, v) is a lower bound, i.e., no exact cost calculation was performed.

After reversing a machine arc (u, v) the heads and tails must be updated. Profiling indicated that these updates consume 90% of the total computation time. Instead of recalculating every value we use the bow-tie algorithm proposed by Ten Eikelder, Aarts, Verhoeven & Aarts [19], which decreases the total update time by 35 to 40 percent. Suppose that (u, v) is the machine arc that will be reversed next. The head of v is determined by a longest path to v which contains either (u, v) or (jp_v, v). After reversing (u, v) the length of a longest path to v is equal to the maximum of $r_{mp_u} + p_{mp_u}$ and $r_{jp_v} + p_{jp_v}$. As $p_u \geq 0$ we conclude that r_v cannot increase (see Figure 5.1). Because the head of an operation is defined in terms of its job predecessor and machine predecessor, only the head of v and of all operations that can be reached from v require an update. This is accomplished by calculating the set of all operations that can be reached from v and applying Bellmann's labelling algorithm to the subgraph induced by these operations. The update of the tails is analogous.

5.4 Restart mechanism

In our experience, the elementary tabu search algorithm described above tends to get stuck in some part of the search space. There are several ways to direct the search to other parts of the search space.

One way to overcome this problem is by restarting the search with a new random initial solution. In this way, it is likely that a larger part of the search space will be explored. Its major drawback is that the search is restarted from scratch since the efforts of the preceding runs are ignored. As a diversification mechanism this works well, but it lacks intensification.

Another way to restart tabu search is by means of a backtrack mechanism. When tabu search gets stuck, it jumps back to a promising solution found in the past and avoids that the same search path is followed again. See for instance Nowicki & Smutnicki [51] and Ten Eikelder, Aarts, Verhoeven & Aarts [19]. A drawback of this approach is that the new start solution is close to the original search path, which can prevent escaping from a region in which it got stuck. In other words, the emphasis lies on intensifying the search while it incorporates almost no diversification.

Instead of restarting the search, one can also maintain a frequency based memory that depends on the number of times that a particular attribute has been added to or removed from a solution. When selecting a neighbor, this type of memory is used to discourage changes that have already occurred frequently, for instance by means of a penalty term in the objective function. Although this technique incorporates both intensification and diversification, it requires a lot of experience and tuning. See for instance Taillard [68] and Glover & Laguna [30].

This section describes a probabilistic restart mechanism based upon multiple independent runs that does not restart from scratch but uses information obtained from preceding runs to construct an initial solution.

5.4.1 Commonalities

For job shop scheduling problems it is often the case that certain scheduling decisions are more obvious than others. For instance, suppose that operation u is the first operation of its job, operation v is the last operation of another job and they require processing on the same machine. It is clear that good solutions will probably have (u, v) rather than (v, u). Hence, it is likely that good solutions have certain machine arcs in common and it is this kind of information that could be useful when constructing a new start solution.

Definition 5.4.1. *Let $\mathcal{L}$ denote a multiset of solutions. The* commonality set *of $\mathcal{L}$ is defined as the subset of machine arcs that are common to all solutions in $\mathcal{L}$. Each machine arc in the commonality set is called a* commonality *of $\mathcal{L}$.*

In order to get an idea of the abundance of commonalities, we carried

instance/T	0	10^0	10^1	10^2	10^3	10^4	10^5	10^6
ta21	35.7	36.2	63.2	70.6	72.3	77.7	78.3	82.2
ta22	36.4	37.1	64.6	65.7	74.7	77.2	78.7	80.1
ta23	35.8	38.4	62.3	69.9	73.9	77.5	79.8	83.9
ta24	36.4	37.3	64.3	68.4	73.1	76.7	83.0	88.9
ta25	35.9	37.2	64.3	72.1	73.1	73.0	75.2	79.4
ta26	35.9	38.2	63.9	70.3	71.4	75.7	80.2	80.2
ta27	36.2	38.2	61.9	70.3	74.1	79.0	81.3	83.9
ta28	36.7	37.8	63.0	67.7	76.6	79.5	80.5	82.0
ta29	36.3	38.2	66.7	71.9	78.8	83.6	85.2	86.1
ta30	35.4	36.8	62.2	69.7	74.1	73.6	77.9	83.5
yn1	37.1	39.8	64.9	73.2	79.1	81.1	82.0	85.5
yn2	37.1	38.0	65.6	70.5	74.7	78.7	80.8	82.6
yn3	36.3	36.6	61.1	70.7	76.0	80.2	85.5	87.8
yn4	32.8	34.4	57.2	70.2	72.0	78.4	80.6	81.8

Table 5.1: The average percentage of machine arcs in the best local optimum found that are common to the 25 best local optima found during 25 independent runs of the elementary tabu search algorithm.

out the following experiment. The 25 best local optima found during 25 independent runs of the elementary tabu search heuristic are stored in a list $\mathcal{L}$. For every machine arc (u, v) in the best local optimum L_1 in $\mathcal{L}$, we determine whether or not all other local optima in $\mathcal{L}$ have u scheduled before v. The percentage of machine arcs in L_1 for which this is the case is given in Table 5.1. The initial solutions are obtained from a randomized construction heuristic. Each independent run halts after T subsequent non-improving iterations. Hence $T = 0$ corresponds to taking a random initial solution and $T = 1$ to stopping at the first local optimum that is found. We repeat the experiment ten times and take the averages. Notice that for $T = 0$ and $T = 1$ there is exactly one local optimum in $\mathcal{L}$ for every independent run. For the other values of T this is not necessarily true. Although some runs contribute more to $\mathcal{L}$ than others, it is not the case that $\mathcal{L}$ consists entirely of local optima found during a relatively small number of runs. It is remarkable that the percentages of common machine arcs are very close for $T = 0$ and $T = 1$. On the other hand, the values for $T = 1$ and $T = 10$ differ considerably. For larger T, the percentage of common machine arcs increases to about 80%. Interestingly, the percentages are approximately the same for all problem instances that we considered. We conclude that

good local optima have many machine arcs in common.

Our restart mechanism is based upon perturbing the best solution found so far while preserving the commonalities of a number of so-called elite solutions. The machine arcs common to these elite solutions will form the basis of the new start solution. In this way, useful information gathered during previous runs is used by subsequent runs.

5.4.2 Perturbation

During the execution of tabu search a list $\mathcal{L} = (L_1, L_2, \ldots, L_k)$ of k elite solutions is maintained, for some value k to be chosen later. In order to keep the solutions in $\mathcal{L}$ sufficiently diverse, a solution is inserted only when it is a local optimum and it improves the best solution found so far. The worst solution in $\mathcal{L}$ is removed when $|\mathcal{L}| > k$.

We are interested in the machine arcs that are common to these local optima. This information is used to construct a new starting solution L_0 for tabu search. Machine arcs that are common to many solutions in $\mathcal{L}$ are thought of as being "probably right" and should be part of L_0.

Assume that $c(L_i) \leq c(L_{i+1})$ for $i = 1, 2, \ldots, k-1$. The *commonality class* $\mathcal{C}(u, v)$ of a machine arc (u, v) in L_1 is defined as the largest index i such that all successive local optima $L_1, L_2, \ldots, L_i$ in $\mathcal{L}$ have $u \prec v$. The construction of a new starting solution L_0 consists of two phases. In the first phase, the *selection phase*, each machine arc (u, v) in L_1 is either *accepted* with a probability $p_{\text{acc}}(u, v)$ or *rejected* with probability $1 - p_{\text{acc}}(u, v)$, where $p_{\text{acc}}(u, v)$ ranges between $p_{\min}$ and $p_{\max}$ and depends on the commonality class of (u, v) in $\mathcal{L}$:

$$p_{\text{acc}}(u, v) = \frac{\mathcal{C}(u, v) - 1}{k - 1} p_{\max} + \frac{k - \mathcal{C}(u, v)}{k - 1} p_{\min}.$$

When (u, v) is common to all $L_i \in \mathcal{L}$, the acceptance probability $p_{\text{acc}}(u, v)$ is set to a value $p_{\max}$ that is close to 1. We put $p_{\min} = \frac{1}{2}$, reflecting that machine arcs (u, v) that have $\mathcal{C}(u, v) = 1$ are not commonalities but rather peculiarities of L_1.

For example, the acceptance probabilities for a 20×20-problem instance with 3,800 machine arcs where $|\mathcal{L}| = 10$ are given in Table 5.2. The table is to be interpreted as follows. According to the sixth row, there are 17 machine arcs common to $L_1, \ldots, L_5$ that do not belong to L_6. Each of these machine arcs has an acceptance probability of 0.70, and 11 of them become part of the next start solution. In this experiment all machine arcs

$\mathcal{C}(u,v)$	$\#(u,v)$	p_{acc}	#accepted
10	3591	0.95	3424
9	113	0.90	102
8	6	0.85	4
7	38	0.80	28
6	11	0.75	8
5	17	0.70	11
4	2	0.65	2
3	2	0.60	1
2	16	0.55	8
1	4	0.50	1

Table 5.2: Acceptance probabilities for a sample 20×20 instance.

in the transitive closure of the precedence graph induced by each machine are processed. In Section 5.4.4 we give an alternative method.

In the second phase, the *augmentation phase*, L_0 is augmented to a complete solution. For every machine arc (u, v) in L_1 that was rejected a coin is flipped. Either (u, v) or its complement (v, u) is inserted into L_0, with equal probability, provided that it does not create a cycle. When it does, the inserted machine arc is reversed. The machine arcs in L_1 are not processed in a specific order. Some orders will probably do better than others, but we ignore this.

The solution L_0 obtained in this way is a perturbed version of the best local optimum found so far. It will serve as the initial solution of the next tabu search run. The amount of perturbation applied is determined by $p_{\max}$, $p_{\min}$ and $|\mathcal{L}|$, and typically ranges between 5 and 15 percent.

5.4.3 Restarting tabu search

The first tabu search run starts with a solution that is generated by a simple randomized construction heuristic, which operates as follows. A operation is called *enabled* when its job predecessor is already scheduled, otherwise it is called *disabled*. Initially, only the first operation of each job is enabled. A solution is constructed by repeatedly selecting at random an enabled operation to be scheduled next on its machine, disabling the operation and enabling its job successor.

Each time that a local optimum is found that improves the best solution found so far it is stored in the list $\mathcal{L}$. After T subsequent iterations without

improvement of the best solution tabu search halts. Each subsequent tabu search run starts with a solution obtained by perturbing L_1. Before each restart the tabu list and aspiration levels are cleared and during the search list $\mathcal{L}$ is updated.

Initially, when tabu search gets stuck, the next initial solution will be close to the best solution found so far. At this point, our restart mechanism resembles backtracking. After each restart in which L_1 is not improved, the value of $p_{\max}$ is decreased by multiplying it with a value ϵ, $0 < \epsilon < 1$, which is a parameter of the restart mechanism. This results in more perturbation of L_1 when constructing a new initial solution. Hence, after a large number of non-improving restarts, the initial solution is a strongly perturbed copy of the best solution found so far. It could as well be generated at random; in this case our restart mechanism resembles multiple independent runs. When a restart improves L_1, $p_{\max}$ is reset to its original value. Hence, the parameters $p_{\max}$ and ϵ determine the amount of perturbation having multiple independent runs and backtracking to the best solution found so far as special cases. Backtracking corresponds to $p_{\max} = p_{\min} = 1$ and multiple independent runs to $p_{\max} = p_{\min} = \frac{1}{2}$.

One could argue that in order to get a sufficiently diverse population the best local optima found during multiple independent runs should be combined. In this scenario after n_i independent runs a number of probabilistic restarts will be executed. Experiments showed that it is not necessary to have more than one independent run.

5.4.4 Implementation details

In the selection phase the commonality of every machine arc (u, v) in L_1 is determined. This involves testing whether $(u, v) \in L_i$ or not, for $i = 2, 3, \ldots, k$. According to the definition of digraphs in Section 5.2.1 either (u, v) or (v, u) exists in every L_i, hence the test takes constant time. Unfortunately, the number of machine arcs in a digraph is $\Theta(mn^2)$ for n jobs and m machines. This implies a total of $\Theta(kmn^2)$ machine arcs that must be stored in memory, where k is the number of local optima in $\mathcal{L}$, typically 25. As a result, the memory requirements for larger job shop instances become impractical.

Instead of storing every machine arc of a digraph, we only store the machine arcs in the transitive reduction of the precedence graph induced by each machine. These machine arcs are called *direct* machine arcs and the others are called *transitive* machine arcs. Notice that transitive machine arcs are never part of a longest path, provided that all operations have a

non-zero processing time. Operation u is scheduled before operation v when there exists a path $(u = u_0, u_1, \ldots, u_d = v)$ such that $M_{u_i} = \mu$ for $0 \leq i \leq d$. Testing whether u is scheduled before v in L_i now requires $\mathcal{O}(n)$ time.

The selection phase processes all direct and transitive machine arcs, either explicitly or implicitly. For instance, when the direct machine arcs (u, v) and (v, w) are selected, the transitive machine arc (u, w) is implicitly selected. On the other hand, when (u, v) is selected but (v, w) is not, an explicit selection trial is required for (u, w). In the selection phase L_0 is initially equal to L_1. Then, for every direct machine arc $(u, v) \in L_0$ the acceptance probability $p_{\text{acc}}(u, v)$ is determined and with probability $1 - p_{\text{acc}}(u, v)$ it is rejected, meaning that (u, v) is removed from L_0. When a direct machine arc (u, v) is removed, new direct machine arcs (mp_u, v) and (u, ms_v) are inserted, where mp_u denotes a direct machine predecessor of u and ms_v denotes a direct machine successor of v (see Figure 5.2). As a result of removing machine arcs, the operations on some machine are not completely ordered. Some operations have more that one direct machine predecessor or direct machine successor. These new direct machine arcs are subjected to the same acceptance trial described above. In the worst case all direct and transitive machine arcs are processed. It turns out that in practice about one and a half times the number of direct machine arcs are processed, which is far less than the number of machine arcs in the transitive closure.

During the augmentation phase every rejected machine arc is reversed with probability $\frac{1}{2}$ and inserted into L_0, provided that it does not create a cycle. Inserting a machine arc can make other machine arcs transitive. Since the number of rejected machine arcs is usually small and determining which machine arcs are transitive is quite expensive, transitive machine arcs are not removed. Instead, after the selection and augmentation phase we take the transitive reduction of the precedence graph induced by each machine in L_0.

Profiling indicates that the running time overhead of the restart mechanisms is less than 2 percent. This is due to the relatively small number of restarts, typically between 10 and 30 when the total running time is 15 minutes (see Section 5.5). Hence, the benefit of using advanced data structures for the restart mechanism is negligible.

5.5 Results

In many practical situations the amount of time available for finding an acceptable solution of a combinatorial optimization problem is limited to a

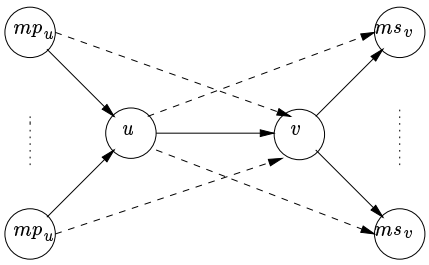

Figure 5.2: Rejection of (u, v).

couple of hours or perhaps a few days. Hence, our tuning objective is to find values for the parameters of tabu search and the restart mechanism such that the performance in a given amount of time is as good as possible. Obviously, the optimum value of these parameters depends on the class of problem instances to which our algorithm is to be applied. In particular the size of these instances plays an important role. In order to be able to compare our algorithm with other techniques our class of problem instances $\mathcal{I}$ is a set of 14 well known 20×20 benchmark instances proposed by Taillard [66], ta21 to ta30, and Yamada & Nakano [74], yn1 to yn4. These test instances are still open, although the gap between the best upper and lower bounds is relatively small. All tuning experiments were performed on a Pentium II 333 MHz/Linux machine and the running time was limited to 15 minutes.

5.5.1 Tuning

The restart mechanism described above has a number of parameters and we want to find good values for them. The most important parameters are $p_{\max}$ and ϵ, which determine the amount of perturbation that is applied to the best local optimum when creating a new starting solution. These parameters are closely related to the stop criterion T of tabu search, which is defined as the maximum number of subsequent iterations in which the best solution is not improved. Intuitively, the immediate surroundings of the best solution found so far are more thoroughly searched when T is big. As a result, more perturbation is needed to prevent that the new starting solution is too close to the best solution found so far. This is accomplished by choosing a smaller value for $p_{\max}$.

Because tuning requires a lot of computational effort, we chose to focus our attention to the two most important parameters, T and $p_{\max}$. During the tuning experiments we always have that ϵ is equal to $p_{\max}$. The

T	$p_{\max}$	av_z	sd_z	#restarts
25,000	0.95	0.0684	0.0008	74
25,000	0.97	0.0678	0.0006	77
25,000	0.99	0.0679	0.0003	83
50,000	0.95	0.0678	0.0006	40
50,000	0.97	0.0676	0.0008	42
50,000	0.99	0.0682	0.0011	45
100,000	0.95	0.0676	0.0006	21
100,000	0.97	0.0684	0.0008	21
100,000	0.99	0.0682	0.0008	22
250,000	0.95	0.0687	0.0010	7
250,000	0.97	0.0688	0.0007	7
250,000	0.99	0.0681	0.0008	7
500,000	0.95	0.0689	0.0008	3
500,000	0.97	0.0685	0.0006	3
500,000	0.99	0.0683	0.0009	3
1,000,000	0.95	0.0693	0.0006	1
1,000,000	0.97	0.0689	0.0004	1
1,000,000	0.99	0.0695	0.0009	1

Table 5.3: Tuning results for our probabilistic restart mechanism with $p_{\min} = \frac{1}{2}$, $\epsilon = p_{\max}$ and $k = 25$.

other parameters were fixed to appropriate values found during numerous preliminary experiments: $k = 25$ and $p_{\min} = 0.5$.

Our objective is to find a set of parameters that performs well on the set $\mathcal{I}$ of test instances. First, we need a measure of how good a given parameter set is. For each parameter set $(T, p_{\max})$ the algorithm is executed once for every test instance $I \in \mathcal{I}$. The average relative distance to the lower bound over the instances in $\mathcal{I}$ is given by

$$z(T, p_{\max}) = \frac{1}{|\mathcal{I}|} \sum_{I \in \mathcal{I}} \frac{UB_I - LB_I}{LB_I},$$

where UB_I is the cost of the best solution found and LB_I is the best known lower bound. We use this quantity to measure the performance of a parameter set. As the algorithm incorporates randomization, $z(T, p_{\max})$ is a random variable. Therefore, the experiment is repeated ten times for every parameter set and the average av_z and the standard deviation sd_z are calculated. The results are given in Table 5.3.

T	ΔT	$T_{\min}$	av_z	sd_z	#restarts
25,000	1,000	1,000	0.0679	0.0011	21
25,000	2,500	2,500	0.0681	0.0011	27
25,000	5,000	5,000	0.0677	0.0013	42
25,000	10,000	10,000	0.0678	0.0009	75
50,000	2,000	2,000	0.0679	0.0010	10
50,000	5,000	5,000	0.0676	0.0011	13
50,000	10,000	10,000	0.0681	0.0012	20
50,000	20,000	20,000	0.0670	0.0009	36
100,000	4,000	4,000	0.0676	0.0009	4
100,000	10,000	10,000	0.0673	0.0009	6
100,000	20,000	20,000	0.0666	0.0007	9
100,000	20,000	40,000	0.0675	0.0010	17
250,000	10,000	10,000	0.0687	0.0011	1
250,000	25,000	25,000	0.0679	0.0011	2
250,000	50,000	50,000	0.0676	0.0009	3
250,000	100,000	100,000	0.0678	0.0010	6
500,000	20,000	20,000	0.0687	0.0008	0
500,000	50,000	50,000	0.0686	0.0008	0
500,000	100,000	100,000	0.0683	0.0005	1
500,000	200,000	200,000	0.0678	0.0007	2

Table 5.4: Tuning results for backtracking.

The results in this table, and the tables to come, should not be taken too literally, since for a problem size of 20×20 far more runs are needed to determine an accurate confidence interval. Nevertheless, the trends in the tables provide sufficient information to distinguish good parameter sets from bad ones.

A quick glance at Table 5.3 suggests that, for our test instances, T should be smaller than 250,000. Moreover, we note that the preferred value for T decreases from 100,000 to 25,000 when $p_{\max}$ and ϵ are increased from 0.95 to 0.99. This is in accordance with our intuition that a stronger perturbation, i.e., small $p_{\max}$ and ϵ, of the best solution found so far is needed when its surroundings are more thoroughly searched, which corresponds to a high value of T. The best parameters with respect to av_z and sd_z are $T = 100{,}000$ and $p_{\max} = 0.95$. These parameters resulted in an average of 21 restarts in 15 minutes of computation time.

We applied the same tuning method to multiple independent runs and to

T	av_z	sd_z	#restarts
5,000	0.0751	0.0003	175
10,000	0.0729	0.0010	102
25,000	0.0710	0.0009	48
50,000	0.0706	0.0009	26
75,000	0.0701	0.0009	19
100,000	0.0700	0.0007	15
250,000	0.0699	0.0015	6
500,000	0.0695	0.0019	3

Table 5.5: Tuning multiple independent runs.

a backtrack mechanism inspired by Ten Eikelder, Aarts, Verhoeven & Aarts [19]. The backtrack mechanism under consideration maintains a list $\mathcal{L}$ of k elite solutions. Each time that a local optimum is found that improves the best solution, it is stored in the list together with the current tabu list, the aspiration levels and the list of the unvisited neighbors. After T subsequent iterations in which no improvement of the best solution is found, tabu search halts. The best unvisited neighbor of the best solution in $\mathcal{L}$ is the new start solution and T is decreased by ΔT, which is a parameter of the backtrack mechanism. When T is less than $T_{\min}$, another parameter of the backtrack mechanism, or the best local optimum in $\mathcal{L}$ has no unvisited neighbors, it is removed from $\mathcal{L}$ and T is reset to its original value. The tuning results are given in Table 5.4. In this table, the number of restarts denotes the number of different solutions on the elite list in which backtracking occurred. The best results are obtained for $T = 50{,}000$ and $T = 100{,}000$. In fact, the results are slightly better than the results of our restart mechanism. On the other hand, the standard deviation of our approach appears to be smaller. We always have $\Delta T = T_{\min}$ and high values of these parameters give better results than small values, meaning that jumping back to another solution in the elite list is to be preferred to exhausting all unvisited neighbors of the elite solution that is currently used for backtracking. Again we conclude that frequently restarting the search is to be preferred to longer runs.

The results for independent runs are given in Table 5.5. Clearly, the quality of the solutions found increases with T, at the expense of a considerable increase of the standard deviation. Independent runs are dominated with respect to solution quality and standard deviation by our restart mechanism and by backtracking.

instance	size	LB	old UB	new UB		Δ
ta16	20×15	1300	1368	1362		6
ta21	20×20	1539	1647	1645		2
ta25	20×20	1504	1598	1597		1
ta26	20×20	1539	1655	1651		4
ta27	20×20	1616	1689	1687		2
ta30	20×20	1473	1596	1585		11
ta31	30×15	1764	1766	1764	$*$	2
ta36	30×15	1819	1823	1819	$*$	4
ta38	30×15	1673	1681	1677		4
ta39	30×15	1795	1798	1795	$*$	3
ta44	30×20	1927	2003	1998		5
ta46	30×20	1940	2033	2029		4
ta47	30×20	1789	1920	1913		7
ta48	30×20	1912	1973	1971		2
ta49	30×20	1915	1991	1984		7
ta50	30×20	1807	1951	1937		14
ta62	50×20	2869	2895	2872		23
ta67	50×20	2825	2826	2825	$*$	1
swv04	20×10	1450	1483	1478		5
swv08	20×15	1640	1770	1759		11
swv10	20×15	1631	1773	1761		12
swv11	50×10	2983	3005	2989		16
swv12	50×10	2972	3038	3022		16
swv13	50×10	3104	3146	3104	$*$	42
swv15	50×10	2885	2940	2911		29
yn1	20×20	826	888	887		1
yn2	20×20	861	909	908		1
yn3	20×20	827	894	893		1
yn4	20×20	918	972	970		2

Table 5.6: New upper bounds. An $*$ means that an optimum was found.

5.5.2 Unlimited time

During the numerous computational experiments while testing and tuning our restart mechanism we solved five problem instances known from the literature that were not solved before. Moreover, we improved the upper bound of twenty other hard problem instances. The computation time ranged be-

tween a couple of hours and several days. These results are summarized in Table 5.6; an $*$ means that an optimum was found.

5.6 Conclusion

We presented a restart mechanism for tabu search that is based upon perturbing certain machine arcs in the best local optimum found so far. During the search process, a list $\mathcal{L}$ consisting of k elite solutions is maintained. Machine arcs that are common to many local optima in $\mathcal{L}$ have a small perturbation probability since they are "probably right". According to the tuning results we conclude that our restart mechanism is more robust than backtracking, approximately as robust as independent runs, more effective than independent runs, but slightly less effective than backtracking. When compared to backtracking our restart mechanism has the advantage that it is not necessary to store a list of tabu lists and aspiration levels. A drawback is that it is less trivial to implement.

When the running time was of no concern new upper bounds to twenty five hard benchmark instances were found and five of these were solved for the first time. It might be time for a benchmark with bigger problem instances.

Chapter 6

Whizzkids '97

6.1 Introduction and motivation

In 1997, a second Whizzkids contest was organized by the Department of Mathematics and Computing Science at the Technische Universiteit Eindhoven, CMG and De Telegraaf. This time, the participants had to construct a plan for a so-called parents' night at school where parents have the opportunity to have conversations with the teachers of their kids. Again, the principal reason was to increase interest in mathematics and computer science among high school students. We discuss a tabu search heuristic that is based upon the well known tabu search heuristic due to Nowicki and Smutnicki for the job shop scheduling problem. In addition, we propose a commonality-preserving restart mechanism, which slightly improves upon the finite-time behavior of the tabu search heuristic.

6.2 Assignment

The requirements for the contest were the same as in 1996: the assignment is easy to state but hard to solve, and it should be a non-standard problem that has not received much attention in the literature. Furthermore, the size of the problem instance should be too large for enumerative solution methods, such that participants are forced to use intuition and heuristics in which a lot of creativity can be deployed.

In 1997, the participants had to construct a plan for a parents' night at school, a problem that is likely to be close to the personal experience of the target group of participants. Each parent selects a number of teachers, specifies the duration of the conversations, and gives a partial order in which

the conversations are required to take place. The order is not necessarily linear. Each parent specifies which teachers belong to the same *group*. The order between these groups of teachers is fixed, but the order of the teachers within each group is irrelevant. For example, a parent can require that the math teacher comes first, then the teachers of the classical languages in any order, and finally the teachers of the natural sciences in any order.

Definition 6.2.1. *(Whizzkids '97). Given are 20 parents and 15 teachers. Each parent specifies a partial order of a special kind in which the teachers are to be visited and the corresponding conversation times. The order is not necessarily linear: each parent specifies which teachers belong to the same group; the order of the teachers within each group is irrelevant, while the order between the groups is fixed. It is not required that every parent meets every teacher. The problem is to construct a plan in which the last conversation is completed as early as possible. The secondary objective is to minimize the average amount of time that the parents spend at school.*

Whizzkids '97 is a mixture of a job shop and an open shop scheduling problem: in a job shop scheduling problem, the operations of each job are completely ordered, while in a open shop scheduling problem, there are no precedences between the operations that belong to the same job. In terms of the job shop scheduling problem, the parents correspond to jobs. As we must decide upon the relative order of the conversations for each teacher, the teachers correspond to machines. In addition, we must decide upon the relative order of the teachers within each group, implying that the groups also correspond to machines. Nevertheless, it is appropriate to differentiate between teachers and groups. Obviously, the conversations correspond to the operations of a job shop scheduling problem. In what follows, we use the terms parent and job, teacher and machine, and conversation and operation interchangeably. As the job shop scheduling problem is a special case of Whizzkids '97, the latter is NP-hard in the strong sense.

The primary objective corresponds to the makespan of a job shop scheduling problem. The secondary objective corresponds to the average flow time. According to Wennink [72], given a linear ordering of the conversations for each parent and each teacher, the average flow time can be minimized by solving a minimum cost flow problem. The secondary objective enables us to differentiate between solutions of the same length submitted by the participants.

6.2.1 Finding a hard problem instance

We generated a number of random problem instances with 20 parents and 15 teachers. We used the following ideas to generate hard instances:

- A small number of relatively long conversations are included, because it is likely that these conversations are hard to manipulate within a schedule.
- Teacher loads are balanced, implying that all teachers are equally busy and no teacher is easy to schedule.
- The parent orders are highly correlated such that, at any time, many parents have to compete for the same teacher.

All conversation times are in minutes. We assume that the parents' night starts at 16:00. In order to obtain some level of realism, we attempt to find a hard problem instance that can be finished just before midnight. As a result, the optimum makespan should be slightly less than 480 minutes.

The problem instances are generated as follows. First, we determine for each parent a subset of teachers. For each parent, the probability of not having a conversation with a certain teacher is 0.35. The total number of conversations is at most 300 (20 times 15).

Next, the conversation times are generated at random, according to the following probability distribution: with probability 0.85, a value is drawn uniformly at random from the interval $[5, 20]$, and with probability 0.15, a value is drawn uniformly at random from the interval $[40, 50]$. Subsequently, the machine with maximum load is determined, and all conversations on the other machines are multiplied with a machine-dependent factor in order to balance the machine loads.

For the first job, we determine a random complete ordering. The other jobs initially have the same order as the first job, but a small number of random pairs of operations are transposed. In this way, we obtain a high job order correlation. As a result, many machines are requested by multiple jobs at any time, giving many scheduling conflicts.

Finally, the groups are constructed as follows. The operations of each job are traversed in job order. As long as there are operations left, a random number G in $[1, 3]$ is drawn, indicating that the next G operations in the job belong to the same group.

In order to test the generated problem instances, we implemented a tabu search heuristic (see Section 6.3). For each instance, we performed 1,000 relatively short runs, starting from a random initial solution. We analyzed the

De opgave Print deze pagina zó uit dat hij op een liggend A-4 past WHIZZKIDS '97 prijsvraag

Ouders	Aanvraag							
1	(E16)	(B20)	(F12)	(G39, M13, N10)	(P13)			
2	(A22)	(G27)	(C20)	(D25, E32, F8)	(B14, H19, J9)	(K29, M29)	(N26, P5)	
3	(B9)	(D9, E15)	(G29, H52)	(M24, J29)	(N13, P19)	(R27)		
4	(K30, B52)	(C27)	(D19, E32)	(F16, G10, H8)	(J18, A12)	(L23)	(M24, N26)	(P41)
5	(A16)	(B18)	(C17, M33, E18)	(H56, L13)	(N22, P7)	(R13)		
6	(B12, D7)	(C27, F21)	(G12, H6)	(K73)	(L30)	(M22)	(N10, R32)	
7	(B6)	(C32)	(J18)	(H55, L9)	(N24)	(P48)	(R29)	
8	(A18)	(C27, D60)	(F21)	(J31)	(G22, K26)	(M13)	(P16, R26)	
9	(A34)	(B18)	(C9, K29)	(M20)	(P12)			
10	(A30)	(C15, D11)	(E15)	(G35, J36)	(L23)	(M17)	(P18, R10)	
11	(A30)	(D17)	(E8, F12)	(G12, H15, J34)	(L11, N13)	(R26)		
12	(A38, B14)	(G31)	(D11)	(F13, C19, J11)	(K12)	(L30, N12)	(R30)	
13	(A30, D55, E31)	(F18)	(K17, J25)	(H15)	(L33)	(P13, R8)		
14	(C15, B12, E26)	(F62, H14, J25)	(K29, L16)	(M13, N9)	(P15)	(R30)		
15	(A28, B45)	(C22, F10, D23)	(H16, J18)	(K12)	(L31, N24)			
16	(B23)	(C15)	(D21)	(F14, H21, K15)	(L16)	(N13, R23)		
17	(C10)	(F9)	(G37)	(H19, E10, P18)	(R10)			
18	(B16, L35)	(D14, E74, F21)	(G27)	(J13)	(K14)	(N17, P8)		
19	(B23, C23)	(D25)	(A36)	(G14, K11)	(M89, N71)	(P15, R15)		
20	(B17, C20)	(E18)	(J27, F59)	(L28)	(N8)	(P42, R18)		

Figure 6.1: The chosen problem instance. Each row corresponds to a parent and each conversation consists of a teacher and a duration in minutes.

cost distribution of the solutions that were found. Furthermore, we calculated a number of simple lower bounds for the problem instances: the maximum job length, the maximum machine load, and the maximum preemptive single-machine head-body-tail bound over all teachers and groups. Based upon the cost distributions and the lower bounds, we selected a problem instance for which there was a significant gap between the best upper and lower bounds, and which had a non-smooth cost distribution with only a small number of solutions of a very high quality.

Assignment. The chosen problem instance consists of 197 conversations, see Figure 6.1. For example, according to the third row, parent number 3 specifies that the first conversation takes place with teacher B for 9 minutes, followed by teacher D for 9 minutes and teacher E for 15 minutes in any order, etcetera. The data can be obtained via our website,

`http://www.win.tue.nl/whizzkids/1997/instance.txt`.

6.2.2 Results

For the chosen instance, we performed a number of very long runs of our tabu search heuristic. An upper bound of 469 was found. The details can be found in Section 6.3.

Simple lower bounds include the maximum job length (338) and the maximum machine load (299). The maximum preemptive single-machine head-body-tail bound over all groups and teachers is 416.

Cor Hurkens implemented a branch-and-bound algorithm based upon shaving, which is a lower bounding technique due to Martin and Shmoys for the job shop scheduling problem (see Chapter 7). For a given time T, shaving is a systematic way to reduce the start time intervals of the conversations in schedules of length at most T. Shaving gives a lower bound of 434, while double shave gives 446. Compared to the job shop scheduling problem, a different ascendant set condition was used, which was significantly faster to compute, quadratic in the number of operations on the resource instead of cubic, but slightly less powerful. Branching takes place by splitting the start time interval of a conversation in halves, producing two nodes. The first conversation to branch on was selected by hand. The tree is traversed in depth first-order; it reached a depth of 35, and a total of 19,500 nodes were generated. Branch-and-bound with double shaving proved that 469 is a lower bound. As the lower bound equals the upper bound, the instance is solved. The total computation time of the branch-and-bound algorithm was approximately 65 hours on a Pentium II/266 MHz.

We received 349 solutions in the general and professional category, eleven of which with a makespan of 469. The very best schedule had an average flow time of 262.05 minutes. It appeared that the contest was harder than the previous one.

6.3 Tabu search

We describe the tabu search heuristic that was implemented to help determine hard problem instances. The tabu search heuristic was derived from the tabu search heuristic for the job shop scheduling problem, as described in Chapter 5. Only the primary objective is taken into account.

6.3.1 Solution representation

The problem instance is represented by a vertex-weighted mixed graph. Each conversation corresponds to a vertex and the weight of a vertex is equal to the duration of the conversation. For each direct precedence between two teachers, as specified by a parent, there is a (directed) arc. There is an (undirected) edge between each pair of conversations of the same teacher and between each pair of conversations within the same group. An edge indicates a capacity constraint: a teacher can have at most one conversation at a time (machine edge) and a parent can have at most one conversation at a time (group edge). Hence, we must decide which of the two conversations comes first by orienting the edge, i.e., turning it into an arc. A solution

is obtained by orienting all edges in such a way that the resulting directed graph is acyclic. The *makespan* of the solution is equal to the length of a longest path in the graph, where the length of a path $(v_1, v_2, \ldots, v_z)$ is defined as the sum of the processing times $\sum_{i=1}^{z} p_{v_i}$ of the operations. By using this representation, we implicitly restrict our attention to left-justified schedules.

We introduce the following additional notation. In an acyclic digraph, jp_u is the immediate job predecessor of an operation $u \in \mathcal{O}$, js_u is its immediate job successor, mp_u is its immediate machine predecessor, and ms_u is its immediate machine successor, whenever they exist.

6.3.2 Neighborhood

Our neighborhood function is a generalization of the one proposed by Nowicki and Smutnicki [51] for job shop scheduling. A *block* is maximum sequence of operations processed by the same machine or in the same group and belonging to a longest path. Hence, a longest path is a sequence of blocks. An *internal arc* is an arc between two operations in a single block, provided that neither operation is the first or the last operation in the block. The neighborhood of a solution consists of all solutions that can be obtained by swapping the last two operations of the first block, the first two operations of the last block, or the first two or last two operations of an intermediate block.

6.3.3 Tabu list and aspiration levels

In order to prevent cycling, a tabu list τ is employed. After reversing a machine arc or group arc (u, v), it is forbidden to reverse any arc originating from v, for some period of time. This period of time is called the tabu tenure, and in our algorithm it is equal to $\max\{\bar{\mathcal{N}} + X, 1\}$, where $\bar{\mathcal{N}}$ is the average number of neighbors of the solutions visited so far and X is a random number between 0 and 5. With this tabu list, it is possible that a certain move is forbidden although the resulting solution was never seen before. Aspiration level criteria can overrule the tabu status of moves. The tabu list and aspiration level criteria are exactly the same as for the job shop scheduling problem. For details, the reader is referred to Chapter 5.

6.3.4 Neighborhood search strategy

We adopt a greedy neighborhood search strategy in which a minimum-cost neighbor is selected at each iteration. The neighborhood is explored by

traversing a longest path in the current solution; a tie between a group arc and a machine arc is broken at random.

A significant speed-up is obtained by calculating fast approximations of the cost of neighbors, rather than calculating the exact cost. For the job shop scheduling problem, Taillard [68] proposed an $\mathcal{O}(1)$ time cost approximation of neighbors. It is based on the heads and tails of the operations, and how they change after a reversal of a machine arc. The head r_u of an operation u is the length of a longest path up to and excluding u; the tail q_u of u is the length of a longest path originating from and excluding u. Given that the head of an operation can be expressed recursively in terms of the head of its job and machine predecessors, and that the tail of an operation can be expressed in terms of the tail of its job and machine successor, updating heads and tails after reversing a machine arc requires only constant time.

We generalize Taillard's cost approximation as follows. For machine arcs, the $\mathcal{O}(1)$ time cost approximation of Taillard for the job shop is used (see Chapter 5). The cost of the solution that is obtained after reversing a machine arc (u, v) is approximated by $\max\{r'_u + p_u + q'_u, r'_v + p_v + q'_v\}$, where a prime indicates an updated value. When, after the reversal, (v, u) is on a longest path, the approximation is exact. Otherwise, it is a lower bound on the new makespan.

For group arcs, a similar approximation is used. The heads and tails can be updated in constant time (see Figure 6.2):

$$
\begin{aligned}
r'_v &= \max\{r_{jp_u} + p_{jp_u}, r_{mp_v} + p_{mp_v}\};\\
r'_u &= \max\{r_{mp_u} + p_{mp_u}, r'_v + p_v\};\\
q'_u &= \max\{p_{ms_u} + q_{ms_u}, p_{js_v} + q_{js_v}\};\\
q'_v &= \max\{p_{ms_v} + q_{ms_v}, p_u + q'_u\}.
\end{aligned}
$$

Again, $\max\{r'_u + p_u + q'_u, r'_v + p_v + q'_v\}$ is a lower bound on the new makespan. When, after the reversal, (v, u) is on a longest path, the bound is exact.

As a result of using lower bounds on the cost of neighbors, the aspiration criterion is also expressed in terms of these lower bounds. The aspiration level α_v of an operation v is defined as the lowest lower bound that was ever evaluated for any move involving v. The tabu status of a move (u, v) is overruled when the lower bound is lower than α_u.

A neighbor is selected as follows. For each move, we calculate the lower bound. When the lower bound is lower than the cost of the current solution, the exact cost of the neighbor is calculated. Tabu moves are made less attractive by adding a large number B to their cost. A lowest cost neighbor

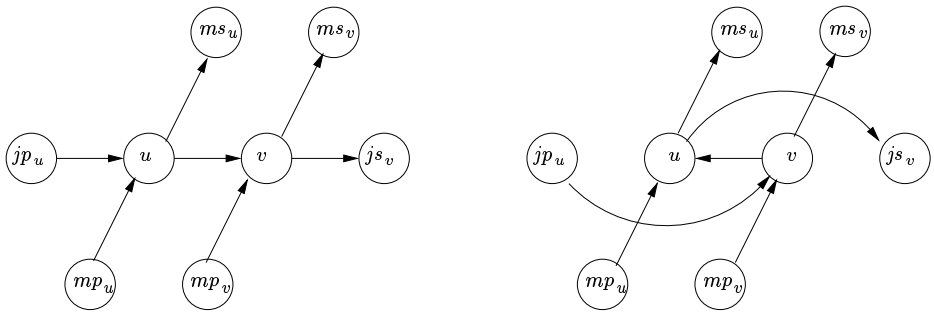

Figure 6.2: Before and after reversal of group arc (u, v).

is selected; in case of a tie between a minimum lower bound neighbor and a minimum exact cost neighbor, we obviously prefer the latter one.

After reversing a machine arc, we apply the bow-tie algorithm proposed by Ten Eikelder, Aarts, Verhoeven & Aarts [19] to update the heads and tails of affected operations, as described in Chapter 5. For group arcs, we follow a similar approach.

6.3.5 Results

We performed a number of computational experiments. The initial solution is constructed as follows. First, all conversations within each group are ordered at random. We build a graph with a vertex for each operation and an arc for each job precedence and each group precedence. Then, for each machine, the operations are ordered at random. One by one, the corresponding machine arcs are added to the graph. When a cycle is detected, the machine arc is reversed. Consequently, the initial solution is highly randomized. For a given integer T, tabu search stops after T subsequent non-improving iterations.

For various run lengths T, the average (av) and standard deviation (sd) are given for 25 independent runs. The running times range from a couple of seconds to a couple of hours on a Pentium II/266 MHz PC. The results are given in Table 6.1. We conclude that, in general, a high value of T gives better results than a low value of T. For the largest run lengths, the averages are within 2 percent of the optimum. The optimum of 469 was never found during these runs. After a number of very long runs, which took several weeks each on a Pentium II/266 MHz PC, solutions of cost 469 were eventually found. In the next section we consider multiple shorter runs

T	av	sd
1,000	548.80	23.00
5,000	506.08	10.87
10,000	506.84	15.12
50,000	489.08	5.63
100,000	485.84	3.70
500,000	481.52	2.97
1,000,000	480.04	2.21
2,000,000	479.88	1.96
4,000,000	477.04	2.05
6,000,000	477.60	1.28
8,000,000	476.67	1.95
10,000,000	476.70	1.62
12,000,000	476.60	1.40
14,000,000	476.14	1.68
16,000,000	476.40	1.63
18,000,000	476.52	1.53
20,000,000	476.12	1.51

Table 6.1: The results of 25 independent runs of the tabu search heuristic for various run lengths T. After T subsequent non-improving iterations, tabu search halts.

instead of a single long run.

6.4 Commonality-preserving restarts

As was the case for the job shop scheduling problem, good solutions to the Whizzkids '97 problem seem to have many machine arcs in common. In addition, these good solutions also seem to have many group arcs in common.

Definition 6.4.1. *Let $\mathcal{L}$ denote a multiset of solutions. The* commonality set *of $\mathcal{L}$ is defined as the subset of machine arcs and group arcs that are common to all solutions in $\mathcal{L}$. Each machine arc and group arc in the commonality set is called a* commonality *of $\mathcal{L}$.*

In this section, we exploit the existence of commonalities in the following way. Instead of starting from scratch, we construct a random initial solution

that preserves many commonalities of solutions that are found by previous runs.

We start with a single independent run. During the independent run, a list of solutions $\mathcal{L} = (L_1, L_2, \ldots, L_k)$ is maintained, for some k to be determined later. The best k local optima that are seen during the independent run are stored in $\mathcal{L}$.

Subsequently, a number of restarts are carried out. For each restart, the initial solution L_0 is a perturbed version of L_1, which is the best solution in $\mathcal{L}$. Initially, L_0 is equal to L_1. The perturbation of L_0 consists of two phases, a selection phase that determines which machine arcs and group arcs in L_0 are removed, and an augmentation phase in which the partial solution produced by the selection phase is augmented to a complete solution.

In the *selection phase*, each machine arc and group arc in L_0 is subjected to an acceptance trial. Intuitively, when many solutions in $\mathcal{L}$ have that u comes before v, the machine arc or group arc (u, v) should be accepted with a high probability. On the other hand, when (u, v) is only present in L_1, the acceptance probability should be low, i.e., close to $\frac{1}{2}$. The *commonality class* $\mathcal{C}(u, v)$ of a machine arc or group arc (u, v) in L_1 is defined as the largest index i such that all solutions $L_1, L_2, \ldots, L_i$ have that u comes before v. For a given $p_{\max} \leq 1$ and a $p_{\min} \geq \frac{1}{2}$, the acceptance probability of (u, v) is given by

$$p_{\text{acc}}(u, v) = \frac{\mathcal{C}(u, v) - 1}{k - 1} p_{\max} + \frac{k - \mathcal{C}(u, v)}{k - 1} p_{\min}.$$

When (u, v) is common to all $L_i \in \mathcal{L}$, $p_{\text{acc}}(u, v) = p_{\max}$; when (u, v) only appears in L_1, $p_{\text{acc}}(u, v) = p_{\min}$.

In the *augmentation phase*, a coin is flipped for every machine arc and group arc (u, v) that was not accepted. With probability $\frac{1}{2}$, the arc in inserted into L_0, and with probability $\frac{1}{2}$ its opposite, (v, u), is inserted. When a cycle is created, the arc is reversed.

During the restarts, we adopt a different maintenance policy for $\mathcal{L}$. Only locally optimal solutions that are better than the best solution in $\mathcal{L}$ are inserted into $\mathcal{L}$. In this way we hope to maintain a sufficiently diverse population $\mathcal{L}$.

After each subsequent run that failed to improve the best solution found so far, the value $p_{\max}$ is decreased by multiplying it by ϵ, which is another parameter of the restart mechanism. When a run improves the best solution found so far, $p_{\max}$ is reset to its original value. We assume that ϵ is always equal to the initial value of $p_{\max}$ and, unless stated otherwise, $p_{\min} = \frac{1}{2}$.

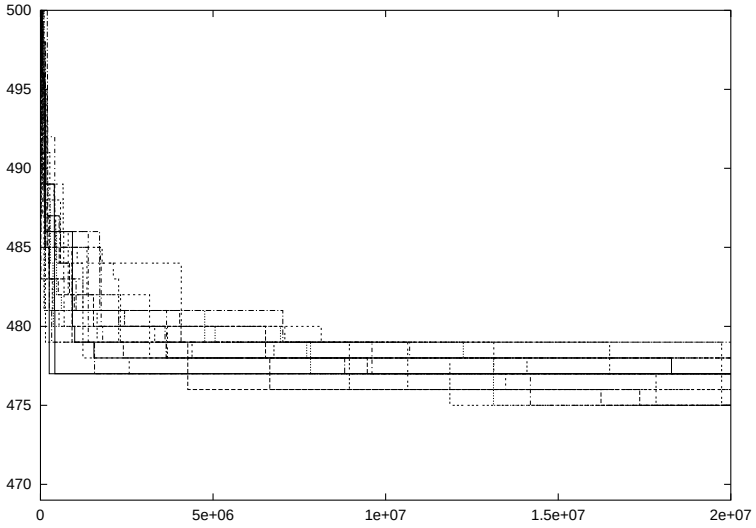

Figure 6.3: The trajectories of 30 runs of tabu search. The solution quality on the y-axis is given as a function of the iteration number on the x-axis.

6.4.1 Results

First, we determine when the tabu search heuristic tends to get stuck. In Figure 6.3, the cost trajectories of 30 independent long runs are given. We conclude that after a total of 5,000,000 iterations, the trajectories are slowed down significantly. Unfortunately, our stop criterion consists of putting a maximum T on the number of subsequent non-improving iterations. Consequently, it is not easy to derive from Figure 6.3 a value for T that corresponds to a total of 5,000,000 iterations. Therefore, we revert to Table 6.1. According to this table, both the average and the standard deviation appear to stagnate when the run length T exceeds 4,000,000 iterations. As a result, we will adopt $T = 4{,}000{,}000$ in our further experiments.

We perform a number of computational experiments in which we compare commonality-preserving restarts to independent runs and to a single long run. In our restart mechanism, the parameter k is always equal to the number of restarts plus 1. In Table 6.2 we compare the results of 26 independent runs, a single run followed by 25 restarts with various values of $p_{\max}$, and a single long run. A long run is simulated by having a single run followed by 25 restarts with $p_{\min} = p_{\max} = \epsilon = 1.0$. We conclude that the commonality-preserving restarts are slightly better than multiple independent runs and slightly better than a single long run; there is no sig-

strategy	av	sd
IR	474.40	1.13
CP/0.80	474.27	1.08
CP/0.85	474.20	1.19
CP/0.90	474.20	1.06
CP/0.95	474.17	1.29
CP/0.98	473.97	1.19
LR	474.23	1.28

Table 6.2: Best solution found by 26 independent runs, one independent run followed by 25 restarts for various values of $p_{\max}$, and a single long run. The run length is 4,000,000 iterations.

nificant difference in the standard deviation. The best results are obtained when the parameter $p_{\max}$ is close to 1. Although commonality-preserving restarts improve upon the finite-time behavior of tabu search, during these experiments, we did not find any solutions of cost 469.

6.5 Conclusion

We implemented a tabu search heuristic, which is based upon the algorithm of Nowicki and Smutnicki for the job shop scheduling problem. After many long runs which took several weeks each, solutions of cost 469 were found. Cor Hurkens proved that 469 is a lower bound, hence the Whizzkids '97 problem has been solved.

In an attempt to improve upon the finite-time behavior of tabu search, we proposed a commonality-preserving restart mechanism. We compared a short run followed by 25 restarts to 26 independent runs and to a single long run. Commonality-preserving restarts are slightly better than independent runs and slightly better than a single long run, in the sense that the average makespan of the solutions found is lower. The standard deviation is approximately the same for each of the three strategies.

Chapter 7

Heuristic shaving

7.1 Introduction

We are concerned with finding lower bounds to job shop scheduling problems. For completeness, we restate the definition of the job shop scheduling problem.

In the *job shop scheduling problem* we are given a set $\mathcal{J}$ of n jobs, a set $\mathcal{M}$ of m machines and a set $\mathcal{O}$ of operations. For each operation $i \in \mathcal{O}$ there is an integral processing time $p_i > 0$, a unique machine $M_i \in \mathcal{M}$ on which it must be processed, and a unique job $J_i \in \mathcal{J}$ to which it belongs. Furthermore, a binary precedence relation $\prec$ is given that decomposes $\mathcal{O}$ into chains, one for each job. The problem is to find a start time s_i for every operation $i \in \mathcal{O}$ such that the *makespan*, defined as $C_{\max} = \max_{i \in \mathcal{O}} s_i + p_i$, is minimized subject to the following constraints:

$$
\begin{array}{ll}
s_i \geq 0 & \text{for all } i \in \mathcal{O}, \\
s_j \geq s_i + p_i & \text{for all } i, j \in \mathcal{O} \text{ with } i \prec j, \text{ and} \\
s_j \geq s_i + p_i \text{ or } s_i \geq s_j + p_j & \text{for all } i, j \in \mathcal{O} \text{ with } M_i = M_j.
\end{array}
$$

The first constraint implies that no machine is available before time 0, the second constraint accounts for the precedence relation $\prec$ such that no operation will start before its predecessor in the chain has finished, and the last constraint is the machine capacity constraint which stipulates that every machine can process at most one operation at a time. For simplicity, we assume that every job visits a machine at most once.

As the job shop scheduling problem is NP-hard, one often resorts to approximation methods. Suppose that we have a heuristic to generate high quality solutions to the job shop scheduling problem. When, after numerous

runs, it seems that no better solution can be found, we might have found an optimum. We could try to prove that the solution is optimal by calculating lower bounds. The difference between the lower bound and the upper bound gives an indication of the relative quality of the solution.

The following lower bounds are trivial:

$$\max_{\mu \in \mathcal{M}} \sum_{i \in \mathcal{O}: M_i = \mu} p_i \quad \text{(maximum machine load), and}$$

$$\max_{\gamma \in \mathcal{J}} \sum_{i \in \mathcal{O}: J_i = \gamma} p_i \quad \text{(maximum job length).}$$

In general, better lower bounds can be obtained with Jackson's preemptive schedule for the single-machine relaxations [34]. Carlier and Pinson [12] proposed a method to strengthen Jackson's preemptive lower bound by means of ascendant set and descendant set updates. Another lower bound technique is the shaving algorithm of Martin and Shmoys [45, 49], which can be viewed as a depth-delimited branch-and-bound method, based upon the work of Carlier and Pinson. Shaving is known to produce excellent bounds, but at the expense of considerable running times. Other lower bounding techniques include the surrogate duality relaxation proposed by Fisher, Lageweg, Lenstra, and Rinnooy Kan [22], the cutting plane approach of Applegate and Cook [5], the geometric methods of Brucker and Jurisch [10], and the fractional packing approach of Martin and Shmoys [45].

The lower bound techniques discussed so far do not use any information about solutions, only about the problem instance. We propose a variant of the shaving algorithm that incorporates information about solutions to guide the process. Suppose that we have a list $\mathcal{L}$ of good solutions. We investigate whether or not information contained in these solutions can be employed to improve the running time of the shaving algorithm. In the original shaving algorithm of Martin and Shmoys, the order in which the operations are shaved solely depends on the job precedences, i.e., on the problem instance and not on solutions. In our approach, two types of heuristic information distilled from the solutions in $\mathcal{L}$ influence the order in which the operations are shaved: the pairwise machine precedences common to many solutions in $\mathcal{L}$, and the intervals of time spanned by the different start times of the operations in the solutions in $\mathcal{L}$. The heuristic information only determines the order in which the operations are shaved. As a result, the validity of the lower bounds is unaffected. Our heuristic shave algorithm derives the same lower bounds as Martin and Shmoys' algorithm, but it often significantly improves upon the running time. In addition, new lower bounds are derived for a number of open problem instances that were proposed by Taillard [66].

The chapter is organized as follows. In Section 7.2, the single-machine relaxation of a job shop scheduling problem is described together with Carlier and Pinson's update algorithm for heads and tails. The principles of shaving are discussed in Section 7.3. The shaving algorithm of Martin and Shmoys is described in Section 7.4. Section 7.5 is devoted to our heuristic shave method, and the results are given in Section 7.6.

7.2 Single-machine relaxation

In the *single-machine relaxation* of a job shop scheduling problem, all machine capacity constraints are relaxed, except for one machine, $\mu \in \mathcal{M}$. Let $\mathcal{O}_\mu$ denote the set of operations processed by machine μ. Relaxing the capacity constraints of a machine implies that more than one operation can be processed at the same time on that machine. Since the job predecessors of an operation $i \in \mathcal{O}_\mu$ do not require processing on machine μ, they can be started at time 0 and processed continuously until they complete at time $r_i := \sum_{j \in \mathcal{O}: j \prec i} p_j$. As a result, operation i can start at time r_i or later. The value r_i is called the *head* or *release time* of operation i. Analogously, when operation i completes, all job successors of i can be processed continuously until they are completed. The total processing time of the job successors of operation i is denoted by $q_i := \sum_{j \in \mathcal{O}: i \prec j} p_j$. The value q_i is called the *tail* or *delivery time* of operation i.

The processing time of an operation is often referred to as its *body*. The single-machine relaxation of a job shop scheduling problem leads to a single-machine *head-body-tail problem*.

Definition 7.2.1. *In the* single-machine head-body-tail problem *we are asked to find a schedule for the set $\mathcal{O}_\mu$ of operations on a machine μ, such that no operation $i \in \mathcal{O}_\mu$ is started before its release time r_i. Furthermore, each operation $i \in \mathcal{O}_\mu$ is processed continuously for p_i units of time and requires a delivery time q_i after its completion on the machine. The machine can process at most one operation at a time. The objective is to minimize the* makespan *which is defined as the maximum completion time plus delivery time over all operations.*

Obviously, the optimum makespan of the single-machine head-body-tail problem corresponding to each machine $\mu \in \mathcal{M}$ is a lower bound on the optimum makespan of the job shop scheduling problem at hand.

In the notation of Graham, Lawler, Lenstra and Rinnooy Kan [31], the single-machine head-body-tail problem is equivalent to the single-machine

i	r_i	p_i	q_i
1	4	6	20
2	0	8	25
3	9	4	30
4	15	6	9
5	20	8	14
6	21	8	16

Table 7.1: An instance of a single-machine head-body-tail problem with 6 operations.

scheduling problem $1|r_j, d_j \leq 0|L_{\max}$ with release times and due dates (by putting the due date d_i of operation i equal to $-q_i$). Unfortunately, the single-machine head-body-tail problem is strongly NP-hard. By allowing *preemption* we get a *preemptive single-machine head-body-tail problem*, which is equivalent to $1|r_j, d_j \leq 0, pmtn|L_{\max}$. In order to solve the problem, we can use Jackson's preemptive earliest due date rule [34] as follows. A schedule σ_{EDD} is constructed, starting at time 0. An operation i is called *available* at time t when $t \geq r_i$ and it has not been completed by time t. At any point in time, the available operation with maximum tail is scheduled, and ties are broken arbitrarily. The release times and completion times of the operations are taken as decision points. By sorting the operations in non-decreasing tail order, the preemptive schedule is computable in $\mathcal{O}(n \log n)$ time, where n is the number of operations on the machine. In practice, the preemptive bound is almost as good as the non-preemptive single-machine bound, but is far more efficient to calculate. Therefore, many optimization methods for the job shop scheduling problem rely on it. In fact, Jackson's preemptive schedule was used by Carlier and Pinson's algorithm [11] that solved the notorious 10×10 problem instance mt10 (Fisher and Thompson [20]) for the first time.

Example. In Table 7.1, the data of a head-body-tail problem with six operations are given. The resulting preemptive schedule is given in Figure 7.1. The makespan of the preemptive schedule in Figure 7.1 is 50, which is the sum of the completion time and the tail of operation 5.

For a given instance of a job shop scheduling problem, let LB^{μ}_{pr} denote the makespan of the preemptive schedule for machine μ. The maximum preemptive single-machine bound over all machines is denoted by LB_{pr}. The value LB_{pr} is a lower bound on the makespan of every solution, and it can be calculated in $\mathcal{O}(mn \log n)$ time, where n is the number of jobs and

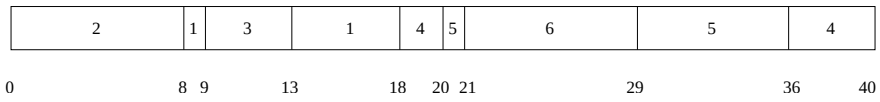

Figure 7.1: Preemptive schedule with a makespan of 50 constructed with Jackson's earliest due date rule.

m the number of machines.

7.2.1 Carlier-Pinson updates of heads and tails

In any solution to an instance of the job shop scheduling problem, the start time of an operation i is at least r_i and the amount of work after completing i is at least q_i. Carlier and Pinson proposed a method to increase heads and tails. They focus on the schedules of length at most T, for some time T, and try to prove that certain operations must be processed before others, thereby updating heads and tails. Obviously, these updated heads and tails are only valid for schedules of length at most T. The idea is that when, as a result of a head or tail update, a single-machine bound exceeds T, there can be no schedules of length T, hence $T + 1$ is a lower bound.

Carlier and Pinson's method is based upon the notion of ascendant sets and descendant sets, for which we introduce the following notation. For a set of operations U on a machine μ, we define

$$\begin{aligned} r(U) &:= \min_{i \in U} r_i, \\ p(U) &:= \sum_{i \in U} p_i, \text{ and} \\ q(U) &:= \min_{i \in U} q_i. \end{aligned}$$

Definition 7.2.2. *For a given machine μ, a time T, and an operation $i \in \mathcal{O}_\mu$, a set $U \subset \mathcal{O}_\mu \setminus \{i\}$ is called an* ascendant set *of operation i, when*

$$r(U \cup \{i\}) + p(U \cup \{i\}) + q(U) > T. \tag{7.1}$$

The earliest possible time at which $U \cup \{i\}$ can start is $r(U \cup \{i\})$. The total processing time of $U \cup \{i\}$ is $p(U \cup \{i\})$, and when i is not completed after all operations in U, $U \cup \{i\}$ is followed by a tail of length at least $q(U)$. So, when equation (7.1) holds, we conclude that in any schedule that completes by time T operation i must be scheduled after all operations in

U. As a consequence, the head of i must be at least the earliest possible completion time of U. Such an update of r_i is called an *ascendant set update*.

Descendant sets are defined similarly. Due to the symmetric role of heads and tails, interchanging the heads and tails just changes the direction in time through which we are scheduling. By starting at time T and scheduling backward, we can restrict ourselves to ascendant sets and head updates.

An efficient algorithm to find the largest possible ascendent set update for every operation on a machine μ is the algorithm of Carlier and Pinson [12], which is outlined below.

Definition 7.2.3. *For a given machine μ and a preemptive schedule σ_{EDD}, let $p_i(t)$ denote the amount of processing that remains for operation i in schedule σ_{EDD} at time t. Furthermore, let $p(t,U) = \sum_{i \in U} p_i(t)$, for any $U \subset \mathcal{O}_\mu$.*

Carlier and Pinson's algorithm repeatedly transforms a preemptive schedule into another preemptive schedule. Given is a preemptive schedule σ_{EDD} for machine μ. For each operation i, a set U_i is constructed that initially consists of all operations that have a longer tail than operation i, and which have not been completely processed in σ_{EDD} at time r_i. The key to Carlier and Pinson's algorithm is that, when

$$r_i + p_i + p(r_i, U_i) + q(U_i) > T, \tag{7.2}$$

the head of operation i must be at least as large as the completion time in σ_{EDD} of the last operation in U_i. The proof is based upon the construction of a set V of operations such that $U_i \cup V$ is an ascendant set of operation i. The set V is constructed as follows. Let $t \leq r_i$ denote the earliest time such that between t and r_i in σ_{EDD} the machine is continuously busy processing operations with tails at least as large as $q(U_i)$. If V denotes the set of operations that are at least partially processed between t and r_i, then it can be shown that $U_i \cup V$ is an ascendant set of operation i. The details of the proof can be found in Martin [45].

Suppose that U_i does not satisfy the condition above. Notice that there are two terms in equation (7.2) that are in a sense complementary. When we remove an operation from U_i, the term $p(r_i, U_i)$ cannot increase while the term $q(U_i)$ cannot decrease. If the operation that is removed from U_i does not have the smallest tail, the term $q(U_i)$ stays the same, so the left-hand site of equation (7.2) will not increase. Consequently, only the removal of an operation in U_i with minimum tail has a chance of increasing the left-hand site of equation (7.2), and can lead to an ascendant set update for operation i.

Carlier and Pinson's algorithm takes a machine μ and a time T as arguments. The algorithm is outlined below.

Carlier-Pinson Head Update:

1. Calculate a preemptive schedule σ_{EDD} for machine μ.
2. For every operation i in $\mathcal{O}_\mu$, do the following steps.
   (a) Calculate $U_i := \{j \in \mathcal{O}_\mu | p_j(r_i) > 0, q_j > q_i\}$.
   (b) If $U_i = \emptyset$, proceed with the next operation.
   (c) When $r_i + p_i + p(r_i, U_i) + q(U_i) > T$:
   - Let $j \in U_i$ denote the last operation to complete in schedule σ_{EDD}. Update i's head: $r_i := \max\{r_i, C_j\}$, where C_j denotes the completion time of operation j in σ_{EDD}.
   - Proceed with the next operation.

   Otherwise,
   - Remove an operation $j \in U_i$ with minimum tail from set U_i.
   - Go to Step 2b.

The complexity of the algorithm is $\mathcal{O}(n^2)$, where n is the number of operations on machine μ, which is equal to the number of jobs. In 1994, Carlier and Pinson [13] presented an algorithm that runs in $\mathcal{O}(n \log n)$ time. We refer to this algorithm as *Fast Carlier-Pinson Head Update*. The total complexity for updating all machines is $\mathcal{O}(mn^2)$ ($\mathcal{O}(mn \log n)$), where m denotes the number of machines.

Example (continued). Suppose that we have a job shop schedule with a makespan of 52. Put $T = 52$ and consider operation 4. All other operations have a longer tail than operation 4, but operations 2 and 3 have already been completed by time $r_4 = 15$. Hence, $U_4 = \{1, 5, 6\}$ and $p(r_4, U_4) = 19$. Since U_4 satisfies equation (7.2), we conclude that the head of operation 4 must be at least the maximum completion time of U_4, which is 36 (see Figure 7.1). Because $t = 0$ and the corresponding $V = \{1, 2, 3\}$, it is easily checked that $U_4 \cup V$ is an ascendant set of operation 4. The new preemptive schedule is given in Figure 7.2. Notice that the makespan of the preemptive schedule is increased by one.

Descendant sets are found in a similar fashion. By interchanging the heads and the tails prior to the invocation of procedure *Carlier-Pinson Head Update*, the largest possible descendent set update is determined for each operation.

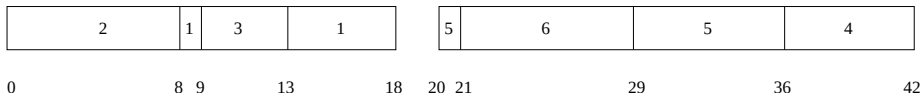

Figure 7.2: The new preemptive schedule with a makespan of 51.

In general, updating one operation can have consequences for the updates of other operations . In order to see this, observe that for operation i, the set U_i is defined in terms of σ_{EDD}, which in turn depends on the heads and tails of all operations. Suppose that operation i is processed before operation j in Step 2 of *Carlier-Pinson Head Update*. When the head or tail of operation j is increased and we would recalculate σ_{EDD}, we might obtain a different preemptive schedule than we had before j's update. Consequently, the set U_i could be different, and new ascendant sets could be found for operation i. Therefore, in order to find all updates we must repeatedly invoke *Carlier-Pinson Head Update* for ascendant sets and descendant sets until no more updates are found. This is accomplished by the procedure *Carlier-Pinson Update*, which has two parameters: a machine μ and a time T. The procedure is outlined below.

Carlier-Pinson Update:

1. In order to find ascendant sets and head updates, invoke *Carlier-Pinson Head Update* for machine μ and time T.
2. Interchange the heads and the tails of the operations in $\mathcal{O}_\mu$.
3. In order to find descendant sets and tail updates, invoke *Carlier-Pinson Head Update* for machine μ and time T.
4. Interchange the heads and the tails of the operations in $\mathcal{O}_\mu$ again.
5. If any head or tail was updated in Step 1 or 3, go to Step 1. Otherwise, halt.

As a result of updating the heads and tails for every machine, it is possible that the preemptive lower bound is increased. The new preemptive lower bound is denoted by $LB_{\mathrm{pr}}(T)$, and is valid for all job shop schedules that are no longer than T, for any T. In practice, the number of rounds in *Carlier-Pinson Update* is small. In 65% of the invocations, one round was sufficient; in 30% of the invocations, two rounds were sufficient; only 5% of the invocations required three or more rounds.

Choosing T

Let $\mathcal{S}$ denote the set of all schedules. Let $\mathcal{S}_T \subset \mathcal{S}$ denote the set of job shop schedules of length at most T. The heads and tails of the operations upon termination of *Carlier-Pinson Update* depend upon $\mathcal{S}_T$: every solution in $\mathcal{S}_T$ has the property that, for every operation i, the start time of i is at least r_i and the amount of work after the completion of i is at least q_i. Hence, the preemptive lower bound $LB_{\rm pr}(T)$ is a lower bound on the length of each schedule in $\mathcal{S}_T$. Obviously, $LB_{\rm pr}(T) \geq LB_{\rm pr}$, for all T. When $LB_{\rm pr}(T) \leq T$, $LB_{\rm pr}$ is also a lower bound on $\mathcal{S}$ (because T is a lower bound on $\mathcal{S} \setminus \mathcal{S}_T$). On the other hand, when $LB_{\rm pr}(T) > T$, we conclude that schedules of length no longer than T cannot exist, hence $T+1$ is a lower bound on $\mathcal{S}$.

If we have a solution of cost UB, we can put $T = UB$ and invoke *Carlier-Pinson Update* to obtain the lower bound $LB_{\rm pr}(UB)$ on $\mathcal{S}_{UB}$, with the property that $LB_{\rm pr} \leq LB_{\rm pr}(UB) \leq UB$. Therefore, $LB_{\rm pr}(UB)$ is also a lower bound on $\mathcal{S}$. Obviously, the higher we put T, the less likely it is that ascendant set and descendant set updates are found. Indeed, $\lim_{T\to\infty} LB_{\rm pr}(T) \downarrow LB_{\rm pr}$.

Instead of using an upper bound, we can also bluff by choosing a value for T for which we have no evidence that there actually exists a schedule that completes by time T. Suppose that we invoke *Carlier-Pinson Update* with such a value of T. When $LB_{\rm pr}(T) > T$, infeasibility is derived, and we conclude that $\mathcal{S}_T = \emptyset$. As a result, schedules of length at most T do not exist, hence $T+1$ is a lower bound on $\mathcal{S}$. Given an initial lower bound and an upper bound, we can use *bisection search* over T between the initial lower bound and upper bound to determine the largest value of T for which $\mathcal{S}_T = \emptyset$.

7.2.2 Propagation of updates

According to the previous section, an update of an operation can lead to an update of another operation on the same machine. Therefore, we must iterate *Carlier-Pinson Update* until no more updates are found on the machine. In this section, we illustrate that an update of an operation on a machine can affect other machines in the sense that new ascendant set or descendant set updates could be found on them. Therefore, we should iterate *Carlier-Pinson Update* until no more updates are found on any affected machine.

For a given job shop scheduling problem instance and an operation i, let jp_i denote the direct job predecessor of i and let js_i denote the direct job

successor of i. The head (tail) of an operation is recursively related to the head (tail) of its job predecessor (successor):

$$\begin{aligned} r_i &\geq r_{jp_i} + p_{jp_i}, \text{ and} \\ q_i &\geq p_{js_i} + q_{js_i}. \end{aligned}$$

So, if we increase the head (tail) of an operation, we can *propagate* the update to its job successors (predecessors). For instance, when r_i has just been increased, the release time of its job successor js_i is affected when $r_{js_i} < r_i + p_i$. In that case, we update r_{js_i} and proceed with the job successor of js_i. Otherwise, the propagation comes to an end.

Procedure *Head Propagate* takes an operation i, a set $\mathcal{K}$ of affected machines, and a time T as arguments. It assumes that i's head has just been increased. The new value of the head of operation i is propagated to its job successors. When the head of a job successor j of i has been updated, we check if $r_j + p_j + q_j > T$. If so, infeasibility is derived, implying that either i's head update cannot lead to schedules of length at most T, or $\mathcal{S}_T = \emptyset$. The former case can only occur as a result of shaving, which is discussed in the next section. When infeasibility is not derived, the machine corresponding to operation j is added to $\mathcal{K}$, indicating that the machine is possibly affected by i's head update and new ascendant sets could be found on it. Upon completion, all machines different from $\mathcal{M}_i$ that are affected by i's update are contained in $\mathcal{K}$. In some cases, $\mathcal{M}_i$ is inserted into $\mathcal{K}$ prior to the invocation of the procedure. Procedure *Head Propagate* is outlined below; procedure *Tail Propagate* works in a similar fashion.

Head Propagate:

1. When i has no job successor, halt (no further propagation possible).
2. Put $x := i$ and $y := js_i$.
3. When $r_y < r_x + p_x$, operation y is affected by the head update of operation x and we put $r_y := r_x + p_x$ and $\mathcal{K} := \mathcal{K} \cup \{M_y\}$. Otherwise, halt (no further propagation possible).
4. In case $r_y + p_y + q_y > T$, halt (infeasible).
5. When y has a job successor, put $x := y$ and $y := js_y$ and go to Step 3.

As discussed above, a head or tail update of an operation on a machine can have consequences for the operations on a different machine μ' such that

new ascendant sets and descendant sets can be found on μ'. For a given set $\mathcal{K}$ of affected machines and a time T, the following algorithm repeatedly invokes *Carlier-Pinson Update* for some machine $\mu \in \mathcal{K}$. Each update is propagated and the affected machines are added to $\mathcal{K}$. The algorithm continues until $\mathcal{K}$ becomes empty or infeasibility is derived.

Iterated Carlier-Pinson Update:

1. If $\mathcal{K}$ is empty, halt.
2. Select some machine $\mu \in \mathcal{K}$.
3. Invoke *Carlier-Pinson Update* to find ascendant set updates and descendant set updates on machine μ with time T.
4. When $LB^{\mu}_{\mathrm{pr}} > T$, halt (infeasible).
5. For all operations i on machine μ, whose head was updated in Step 3, invoke *Head Propagate* with machine set $\mathcal{K}$ and time T. When *Head Propagate* derived infeasibility, halt (infeasible).
6. For all operations i on machine μ, whose tail was updated in Step 3, invoke *Tail Propagate* with machine set $\mathcal{K}$ and time T. When *Tail Propagate* derived infeasibility, halt (infeasible).
7. Put $\mathcal{K} := \mathcal{K} \setminus \{\mu\}$.
8. Go to Step 1.

For a given T, suppose that *Iterated Carlier-Pinson Update* is invoked with $\mathcal{K} = \mathcal{M}$. Upon termination, there are no more head or tail updates possible on any machine. As a result of the updates, the preemptive lower bound is generally improved. When infeasibility is derived, we conclude that $\mathcal{S}_T = \emptyset$, hence $T + 1$ is a lower bound on $\mathcal{S}$. We use bisection search over T to find the largest value of T for which *Iterated Carlier-Pinson Update* derives that $\mathcal{S}_T = \emptyset$.

In the next section, we discuss shaving, which is interpreted as a form of branch-and-bound. In the branching step, the head or tail of some operation i is tentatively increased and the update is propagated. As a result, one or more machines are affected. Subsequently, *Iterated Carlier-Pinson Update* is invoked with $\mathcal{K}$ containing only the affected machines. When infeasibility is derived, we conclude that the branch cannot lead to a schedule of length at most T, hence the branch is discarded.

7.3 Shaving

Given are a job shop scheduling instance and a time T. Consider the heads and tails of the operations upon termination of *Iterated Carlier-Pinson Update* with $\mathcal{K} = \mathcal{M}$. Assume that infeasibility has not been derived. *Shaving* (Martin and Shmoys [49]) is a systematic way to further increase the heads and tails.

Shaving is centered around the concept of processing windows. For a given time T, the *processing window* of an operation i is defined as the interval of time in which the operation is required to start processing in order to let the schedule complete by time T. The processing window of operation i is denoted by $[u_i, v_i]$, where u_i is initially equal to r_i and is increased during the shaving process, and v_i is initially equal to $T - p_i - q_i$ and is decreased during the shaving process. In any schedule that completes by time T, the start time of operation i must lie in the interval $[u_i, v_i]$, see Figure 7.3. For a given time T, the notion of processing windows is just another way to interpret heads and tails and is therefore equivalent. In the remainder of the chapter, r_i and u_i are used interchangeably.

Figure 7.3: The processing window $[u_i, v_i]$ of operation i for a given time T.

Shaving attempts to reduce the processing windows of operations in the following way. If we prove that starting operation i at the left endpoint u_i of its processing window cannot lead to a schedule of length at most T, we can reduce the processing window to $[u_i + 1, v_i]$. More generally, if we can prove that i cannot start in $[u_i, w]$, for some $u_i \leq w \leq v_i$, we can reduce i's processing window to $[w+1, v_i]$. This process is called *head shaving*, because pruning the left end of a processing window corresponds to increasing the head. Similarly, *tail shaving* tries to reduce the right end of a processing window: by proving that i cannot start in $[w, v_i]$, for some $u_i \leq w \leq v_i$, we can reduce i's processing window to $[u_i, w - 1]$. Pruning the right end of a processing window corresponds to increasing the tail.

Obviously, when the processing window of an operation becomes empty, we conclude that $\mathcal{S}_T = \emptyset$, hence $T+1$ is a lower bound on $\mathcal{S}$. Furthermore, as a result of increasing heads and tails, the preemptive lower bound $LB_{\rm pr}(T)$ is generally improved. When, during the shaving process, $LB_{\rm pr}(T)$ becomes

larger than T, we also conclude that $\mathcal{S}_T = \emptyset$ and $T + 1$ is a lower bound on $\mathcal{S}$.

Shaving can be interpreted as a depth-delimited type of branch-and-bound. A branching step consists of temporarily reducing the processing window of an operation. The bounding step tries to derive infeasibility for the chosen branch. When infeasibility has been derived, the branch can be discarded, and the processing window is reduced.

7.3.1 Branching

Head shaving and tail shaving are branching strategies that split a processing window in two parts: a left branch and a right branch.

Head shaving attempts to increase the head of an operation i by temporarily reducing i's processing window to some prefix $[u_i, w]$, for some $u_i \leq w < v_i$. Because in any schedule of length at most T operation i either starts in $[u_i, w]$ or in $[w + 1, v_i]$, the temporary reduction corresponds to taking the left branch. When it can be shown that the left branch cannot lead to a schedule of length at most T, the processing window of i is reduced to $[w + 1, v_i]$ by increasing r_i.

Similarly, tail shaving attempts to increase the tail of an operation i by temporarily reducing i's processing window to some suffix $[w, v_i]$, for some $u_i < w \leq v_i$. In any schedule of length at most T, operation i either starts in $[u_i, w - 1]$ or in $[w, v_i]$. Therefore, the temporary reduction corresponds to taking the right branch. When it is proved that the right branch cannot lead to a schedule of length at most T, i's processing window is reduced to $[u_i, w - 1]$ by increasing q_i.

The way in which the operation i and the value w are chosen is described later.

7.3.2 Bounding

Suppose that, as a result of branching, i's processing window is temporarily restricted to $[u_i, w]$. In the bounding step, we try to derive infeasibility for the selected branch. When infeasibility is derived, we conclude that when i starts in $[u_i, w]$, we cannot obtain a schedule of length at most T. Consequently, we can reduce i's processing window to $[w + 1, v_i]$.

Definition 7.3.1. *For a given set of operations with heads and tails, corresponding to one or more single-machine head-body-tail problems and a time T, a* bounding algorithm *tries to prove that schedules of length at most T cannot exist.*

For a given T, a *bounding algorithm* tries to derive infeasibility for the chosen branch. For example, all variants of Carlier and Pinson's update algorithm are bounding algorithms. They try to derive infeasibility for a given T by updating heads and tails. Infeasibility is derived when the preemptive lower bound of one of the machines exceeds T, or when a processing window of an operation becomes empty. *Iterated Carlier-Pinson Update* is the most powerful bounding algorithm discussed so far, but it is also the most time-consuming one. The choice of the bounding algorithm has a great influence on the lower bound that can be derived by shaving, but also on the running time.

7.3.3 *Remove Prefix* and *Remove Suffix*

Recall that head shaving takes the left branch by temporarily reducing the processing window of an operation i to some prefix, while tail shaving takes the right branch by temporarily reducing the processing window of an operation i to some suffix. Subsequently, a bounding algorithm is invoked that tries to derive infeasibility for the chosen branch. When infeasibility is derived, the branch is discarded. Head shaving and tail shaving rely on the following two procedures, which combine a branch and a bound step.

Procedure *Remove Prefix* has three parameters: an operation i, a value w such that $u_i < w \leq v_i$, and a time T. First, the right branch is taken by increasing i's head to w, thereby reducing i's processing window to $[w, v_i]$. The update is propagated to the job successors and the set $\mathcal{K}$ of all affected machines is determined. Next, the bounding algorithm *Iterated Carlier-Pinson Update* is invoked with machine set $\mathcal{K}$ to see if infeasibility can be derived for the right branch. The procedure is outlined below.

Remove Prefix:

1. Remove the prefix $[u_i, w-1]$ from i's processing window by increasing the head of i, $r_i := w$.

2. If $r_i + p_i + q_i > T$, halt (infeasible).

3. Mark the machine on which i is processed as affected by putting $\mathcal{K} := \{M_i\}$.

4. Invoke *Head Propagate* for operation i, with machine set $\mathcal{K}$ and time T. When infeasibility is derived, halt (infeasible).

5. Invoke *Iterated Carlier-Pinson Update* with the set $\mathcal{K}$ of machines that are affected by i's update, and time T. When infeasibility is derived, halt (infeasible).

Similarly, procedure *Remove Suffix* has three parameters: an operation i, a value w such that $u_i \leq w < v_i$, and a time T. First, the suffix $[w+1, v_i]$ is removed from i's processing window by increasing q_i by $v_i - w$, which corresponds to taking the left branch. The update is propagated to the job predecessors and the set $\mathcal{K}$ of affected machines is determined. Next, the bounding algorithm *Iterated Carlier-Pinson Update* is invoked with machine set $\mathcal{K}$ to see if infeasibility can be derived for the left branch.

7.4 Martin and Shmoys

For a given time T, head shaving and tail shaving attempt to reduce the processing windows as much as possible. As these reductions correspond to increasing heads and tails, the preemptive lower bound $LB_{\text{pr}}(T)$ is generally improved. When $LB_{\text{pr}}(T) > T$, or the processing window of an operation becomes empty, we conclude that schedules of length at most T cannot exist, hence $T+1$ is a lower bound on $\mathcal{S}$.

The order in which the operations are shaved and the way in which w is chosen can vary from one shaving strategy to another. In this section, we focus on the shaving strategy due to Martin and Shmoys [45, 49]. Procedure *MS Head Shave* considers all operations one by one in an attempt to reduce the prefix of their processing windows as much as possible. Similarly, procedure *MS Tail Shave* attempts to reduce the suffix of their processing windows as much as possible.

7.4.1 *MS Head Shave* and *MS Tail Shave*

MS Head Shave processes all operations one by one. For each operation i, the longest prefix that can be removed from its processing window is determined in the following way. Let δ denote the step size, which is initially equal to 1. First, we try to shave off one unit of the front of the processing window $[u_i, v_i]$ by invoking *Remove Suffix* with $w = u_i + \delta - 1$. When *Remove Suffix* derives infeasibility, we conclude that when i starts in $[u_i, w]$, we cannot obtain a schedule of length at most T. Therefore, we can discard that part of the processing window. This is accomplished by invoking *Remove Prefix* with $w = u_i + \delta$. When *Remove Prefix* derives infeasibility, we conclude that schedules of length at most T cannot exist, hence $T+1$ is a lower bound.

Otherwise, the step size δ is doubled and the process is repeated. Let δ denote the step size for which *Remove Suffix* could not derive infeasibility. We proceed with bisection search between the current left endpoint u_i and $M = u_i + \delta$, starting with a step size of $\lfloor \delta/2 \rfloor$. We continue as long as the step size is nonzero and infeasibility is derived by *Remove Suffix*. Upon termination, we have either maximally reduced the left end of the processing window of operation i, or proved that schedules of length at most T cannot exist.

In order to profit from the propagation of updates, for head shave, the operations are sorted by increasing release time. When two or more operations have the same release time, the relative order in which they are processed is arbitrary. The order in which the operations are processed is fixed in advance.

For a given time T, *MS Head Shave* is outlined below. When an interval becomes empty or a preemptive lower bounds exceeds T, infeasibility is derived. In that case, $T + 1$ is a lower bound.

MS Head Shave:
For all operations i in increasing u_i order, do:

1. $\delta := 1$, bisection := false, $M := 0$
2. If $u_i > v_i$, halt (infeasible: schedules of length at most T cannot exist).
3. Temporarily reduce i's processing window to the prefix $[u_i, u_i + \delta - 1]$ by invoking *Remove Suffix* for operation i, with value $w = u_i + \delta - 1$, and time T.
4. When infeasibility is derived in Step 3, we can discard the prefix $[u_i, u_i + \delta - 1]$:

   - Undo all updates accomplished in Step 3.
   - Invoke *Remove Prefix* for operation i with value $w = u_i + \delta$, and time T. When infeasibility is derived, halt (infeasible: schedules of length at most T cannot exist).
   - If bisection = true, $\delta := \lfloor \frac{M - u_i}{2} \rfloor$. Otherwise, $\delta := 2\delta$.
   - When $\delta > 0$, go to Step 2. Otherwise, proceed with the next operation.

   Otherwise,

   - Undo all updates accomplished in Step 3.

- bisection := true, $M := u_i + \delta$, $\delta := \lfloor \delta/2 \rfloor$
- When $\delta > 0$, go to Step 2. Otherwise, proceed with the next operation.

Suppose that the complexity of each procedure that is directly or indirectly invoked by *MS Head Shave* is polynomial in the number of operations of the job shop instance. In the worst case, all values in $[u_i, v_i]$ are shaved off one by one. As a result, the complexity of *MS Head Shave* is at least pseudo-polynomial in the length of the job shop instance.

MS Tail Shave works similarly. In order to profit from the propagation of updates, the operations are shaved in decreasing head order.

In the Section 7.5, we propose an alternative shave order that incorporates information derived from good solutions.

7.4.2 *MS Shave*

Head and tail updates that are found by *MS Head Shave* and *MS Tail Shave* can have consequences for other operations. Therefore, shaving consists of a number of passes. In each pass, all operations are head shaved and tail shaved. When no updates are found during a pass, shaving terminates. In Martin and Shmoys' algorithm first all heads are shaved and then all tails. For a given time T, *MS Shave* is outlined below.

MS Shave:

1. Invoke *MS Head Shave* with time T. When infeasibility is derived, halt (infeasible: schedules of length at most T cannot exist).

2. Invoke *MS Tail Shave* with time T. When infeasibility is derived, halt (infeasible: schedules of length at most T cannot exist).

3. If any head or tail was updated during Step 1 or 2, go to Step 1.

Shaving is useful in finding lower bounds in the following two ways. First of all, the updated heads and tails can lead to a sharper preemptive single-machine lower bound $LB_{\rm pr}(T)$, for a given T. On the other hand, when a processing window of an operation becomes empty, or the preemptive single-machine bound of some machine exceeds T, we conclude that schedules of length at most T cannot exist. Hence, $T + 1$ is a lower bound on $\mathcal{S}$. In the latter case, bisection search over T is employed to find the largest value of T for which $\mathcal{S}_T = \emptyset$. The time T ranges between an initial lower bound and an

initial upper bound. Obviously, the initial lower bound LB can be obtained with *Iterated Carlier-Pinson Update*.

The complexity of *MS Shave* depends upon the complexity of *MS Head Shave* and *MS Tail Shave*. In addition, it is proportional to $\log T$, due to the bisection over T.

7.4.3 *MS Double Shave*

According to Definition 7.3.1, *MS Shave* itself is a bounding algorithm. Instead of only using *Remove Prefix* and *Remove Suffix* to derive infeasibility for a chosen branch, we can also use *MS Shave* for that purpose. Recall that head shaving invokes *Remove Suffix* in an attempt to derive infeasibility for the left branch. When *Remove Suffix* cannot derive infeasibility, we invoke the more powerful *MS Shave* on the chosen branch. When *MS Shave* derives infeasibility, the branch is discarded. For tail shaving, we proceed in a similar fashion.

The resulting algorithm is called *MS Double Shave*. In terms of branch-and-bound, the maximum branch depth of *MS Shave* is one, while the maximum branch depth of *MS Double Shave* is two.

7.4.4 Results

For a number of test instances, the best possible lower bounds that can be derived by *MS Shave* and *MS Double Shave* are given in Table 7.2. All algorithms discussed so far maintain the total amount by which the heads and tails are increased, or equivalently, the total amount by which the processing windows are reduced. This quantity can be found in Table 7.2 in the result column. A result of -1 indicates that infeasibility was derived.

We conclude that the lower bounds found by *MS Double Shave* are much better that those found by *MS Shave*. On the other hand, the computation times of *MS Double Shave* are often much longer. For many instances, *MS Double Shave* is able to show that the optimum value minus 1 is infeasible.

Our implementation is not as efficient as the implementation of Martin and Shmoys because we use the $\mathcal{O}(n^2)$ update algorithm of Carlier and Pinson instead of the $\mathcal{O}(n \log n)$ algorithm. Furthermore, we did not use k-tailed ascendant sets (see Martin [45]). All our experiments are carried out on a 333 MHz Pentium II PC, while Martin and Shmoys used a 90 MHz Pentium PC. The running times of our implementation of *MS Shave* and *MS Double Shave* is up to three times smaller than the running times reported by

		MS Shave			*MS Double Shave*		
instance	optimum	T	result	time	T	result	time
mt10	930	918	−1	20	929	−1	158
		919	8,395	14	930	38,734	815
abz5	1,234	1,201	−1	7	1,233	−1	278
		1,202	4,272	7	1,234	40,912	532
la19	842	829	−1	9	841	−1	225
		830	5,811	10	842	27,325	494
la24	935	917	−1	40	934	−1	5,304
		918	15,045	44	935	58,216	7,144
la25	977	957	−1	22	976	−1	6,315
		958	6,772	20	977	53,444	14,987
la36	1,268	1,266	−1	26	1,267	−1	1,399
		1,267	15,051	27	1,268	93,569	9,156

Table 7.2: The running times of *MS Shave* and *MS Double Shave* in seconds.

Martin and Shmoys. We conclude that the difference in efficiency between their implementation and ours is not very large.

7.5 Heuristic shaving

Given is a list $\mathcal{L}$ of good schedules. We will use the information contained in them in an attempt to improve upon the efficiency of the shaving strategy of Martin and Shmoys. The running time of our heuristic shave strategy does not include the time that is needed to generate the schedules in $\mathcal{L}$. We assume that we already have these schedules, and investigate whether or not we can use them to guide the shave process. Although we use heuristic information derived from the schedules in $\mathcal{L}$, the correctness of the obtained lower bounds is not affected. When the heuristic information is misleading, it only can have consequences for the running time.

Recall that every pass of *MS Head Shave* (*MS Tail Shave*) starts with sorting all operations in increasing (decreasing) head order. We propose an alternative order in which the operations are processed. A *precedence graph* for the operations with job arcs and the machine arcs common to all schedules in $\mathcal{L}$ is employed to heuristically optimize the propagation of updates. In addition, the time intervals in which the operations start in the schedules in $\mathcal{L}$ help us to further refine the heuristic shave order.

7.5.1 Commonalities

A schedule is often represented by a vertex-weighted acyclic digraph $G = (\mathcal{O}, A)$. The arc set A consists of *job arcs* connecting consecutive operations of the same job, and *machine arcs* connecting consecutive operations that must be processed on the same machine. Each vertex $i \in \mathcal{O}$ has a weight p_i. The makespan of the schedule is equal to the length of a longest path in G. The job arcs are fixed, but the machine arcs correspond to basic scheduling decisions. A job arc (i, j) specifies that operation i is processed before operation j in the same job. A machine arc (i, j) stipulates that operation i is processed before operation j on the same machine.

For a schedule L, let $c(L)$ denote the makespan. Suppose that $\mathcal{L} = (L_1, L_2, \dots)$ is a list of good schedules, such that $c(L_i) \leq c(L_j)$ whenever $i < j$. Let UB denote the length of the best schedule in $\mathcal{L}$.

Definition 7.5.1. *For a given $k_{\text{arc}} \geq 1$, the machine arcs that are common to the schedules $L \in \{L_1, \dots, L_{k_{\text{arc}}}\}$ are called the* commonalities *of $\{L_1, \dots, L_{k_{\text{arc}}}\}$.*

The commonalities form a partial order on each machine. Commonalities are thought of as being "probably right" in the sense that it is likely that they are part of many good schedules. Notice that when k_{arc} is increased, the set of commonalities becomes more reliable but contains fewer machine arcs, and therefore less information.

While commonalities are parts of existing solutions of cost at least UB, shaving is concerned with proving that solutions of cost $T < UB$ cannot exist. In order to apply commonalities to shaving, we use the following proposition.

Proposition 7.5.1. *For T not much smaller than UB, many machine arcs that are common to the schedules in $\mathcal{L}$ are also common to the schedules in $\mathcal{S}_T$, whenever $\mathcal{S}_T$ is nonempty.*

We employ the commonalities of $\mathcal{L}$ to determine a heuristic shave order in the following way. Instead of sorting the operations in advance before every pass in increasing (decreasing) head order, we construct a *precedence graph* $\mathcal{G}$ to determine the order in which the operations are head (tail) shaved. Each operation corresponds to a node in $\mathcal{G}$ and for every job arc there is an arc in $\mathcal{G}$. Furthermore, we add the machine arcs that are common to the best k_{arc} solutions in $\mathcal{L}$, for some $k_{\text{arc}} \geq 1$. The common machine arcs are determined as follows. For every pair $\{i, j\}$ of operations that are processed on the same machine, we check whether or not i is scheduled before j in

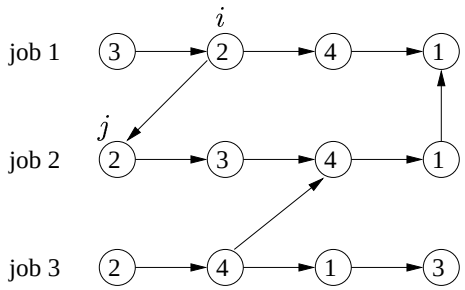

Figure 7.4: A precedence graph $\mathcal{G}$ with three machine arcs for a problem instance with three jobs and four machines. Each operation corresponds to a vertex. The label of a vertex is the machine on which the operation is to be processed.

every solution $L \in \{L_1, \ldots, L_{k_{\text{arc}}}\}$. When this is the case, the machine arc (i, j) is added to the precedence graph $\mathcal{G}$.

In order to clarify the intuition behind the precedence graph thus constructed, consider the one given in Figure 7.4. Let i denote the second operation of job 1 and let j denote the first operation of job 2. Both operations are processed on machine 2. Notice that operation j is released before operation i. Nevertheless, as operation i is to be processed before operation j in many good solutions, intuitively, it is more likely that operation i is in an ascendant set of operation j than vice versa. Hence, j's head could depend on i's head: when i is the last operation to complete in an ascendant set U_j of operation j, then j's head is greater than or equal to the completion time of i. Therefore, an update of i's head could lead to an update of j's head. Suppose that i's head is updated. As a result, machine 2 is affected, so *Carlier-Pinson Update* is invoked. *Carlier-Pinson Update* finds all ascendant set updates and descendant set updates for the operations on machine 2, including operation j. As these updates are, more or less, for free, shaving j before i could be a waste of time.

A shave iteration is quite expensive, so every update should be propagated as much as possible. Propagation of updates is relatively cheap, and it often has a multiplicative effect on the update. Our shave order heuristically optimizes the propagation of updates.

For head shaving, the precedence graph determines the shave order in the following way. An operation is called *available* when all its job and machine predecessors in $\mathcal{G}$ have already been shaved, but the operation itself has not been shaved in the current pass. Hence, only the first operation of each

job is initially available. The set of available operations shifts to the right during head shaving. Similarly, for tail shaving, only the last operation of each job is initially available. The set of available operations shifts to the left during tail shaving.

7.5.2 Heuristic start intervals

During the shave process, it is possible that two or more operations are available at the same time. In Martin and Shmoys' algorithm, the operations are sorted before every pass. For operations with the same release time, ties are broken arbitrarily.

When more than one operation is available, we use priorities to select the next one to shave. The priorities are based upon heuristic information that is distilled from a number of good schedules. Given a list $\mathcal{L} = (L_1, L_2, \dots)$ of good schedules, such that $c(L_i) \leq c(L_j)$ whenever $i < j$. Let UB denote the makespan of the best schedule in $\mathcal{L}$.

Definition 7.5.2. *Let $s_i(L)$ denote the start time of operation i in schedule L. The* heuristic start interval *of operation i, determined by $\{L_1, \dots, L_{k_{\text{int}}}\}$ is the interval $[\alpha_i, \beta_i]$ where*

$$\begin{aligned} \alpha_i &:= \min_{L \in \{L_1, \dots, L_{k_{\text{int}}}\}} s_i(L), \quad \textit{and} \\ \beta_i &:= \max_{L \in \{L_1, \dots, L_{k_{\text{int}}}\}} s_i(L), \end{aligned}$$

for some $k_{\text{int}} \geq 1$.

The heuristic start intervals $[\alpha_i, \beta_i]$ give an indication of the range of the start time of each operation in good schedules. Intuitively, it is unlikely that a good schedule has an operation i that does not start in $[\alpha_i, \beta_i]$. When $k_{\text{int}} \uparrow \infty$, the heuristic start intervals become more reliable.

While heuristic start intervals are derived from existing solutions of cost at least UB, shaving is concerned with proving that solutions of cost $T < UB$ cannot exist. In order to apply heuristic start intervals to shaving, we use the following proposition.

Proposition 7.5.2. *For T not much smaller than UB, the heuristic start intervals of the operations determined by $\mathcal{L}$ approximate the possible start times of the operations in the schedules in $\mathcal{S}_T$, whenever $\mathcal{S}_T$ is nonempty.*

Given the sharp lower bounds that are derived by double shave, see Table 7.2, the values of T with which we shave are not much smaller than UB. Consequently, the heuristic start intervals determined by $\mathcal{L}$ approximate the

possible start times of the operations in the schedules of length at most T (if any). We exploit this feature in the following way.

For a given time T and an operation $i \in \mathcal{O}$, let $u_i = r_i$ and $v_i = T - q_i - p_i$. As some operations have a more flexible start time in good solutions than others, a large gap between u_i and v_i does not necessarily mean that a lot of reductions can be expected. The heuristic start intervals $[\alpha_i, \beta_i]$ indicate which operations have a flexible start time in good schedules and which operations are pretty much fixed to a specific point in time. When $T = UB$, we always have that $\alpha_i \geq u_i$ and $\beta_i \leq v_i = UB - q_i - p_i$. According to Proposition 7.5.2, for T not much smaller than UB, these relations are approximately true. As a result, it is likely that $[\alpha_i, \beta_i] \subset [u_i, v_i]$, and therefore it is unlikely that we can shave within $[\alpha_i, \beta_i]$. For our heuristic shave algorithm, we adopt a greedy priority rule: the available operation with the largest gap between u_i and α_i or between $UB - q_i - p_i$ and β_i is shaved first. Intuitively, it is more likely that we can shave off a large part of a processing window in case of a large gap than in case of a small gap.

7.5.3 *Heuristic Head Shave* and *Heuristic Tail Shave*

For a given list $\mathcal{L}$ of good schedules, a value k_{arc}, a value k_{int}, and a time T, procedure *Heuristic Head Shave* is outlined below. We put $UB = c(L_1)$.

Heuristic Head Shave:

1. Initialize the precedence graph $\mathcal{G}$ with all job arcs and the machine arcs that are common to the best k_{arc} schedules in $\mathcal{L}$.
2. Determine for every operation i the heuristic start interval $[\alpha_i, \beta_i]$ using the best k_{int} schedules in $\mathcal{L}$.
3. Insert all available operations into a priority queue Q. The priority of operation i is $\alpha_i - u_i$.
4. While Q not empty
   (a) Select an available operation $i \in Q$ with maximum priority.
   (b) $\delta := 1$, bisection := false, $M := 0$.
   (c) If $u_i > v_i$, halt (infeasible: schedules of length at most T cannot exist).
   (d) Temporarily reduce i's processing window to the prefix $[u_i, u_i + \delta - 1]$ by invoking *Remove Suffix* for operation i, with value $w = u_i + \delta - 1$, and time T.

(e) When infeasibility is derived in Step 4d, we can discard the prefix $[u_i, u_i + \delta - 1]$:

- Undo all updates accomplished in Step 4d.
- Invoke *Remove Prefix* for operation i with value $w = u_i + \delta$, and time T. When infeasibility is derived, halt (infeasible: schedules of length at most T cannot exist).
- If bisection = true, $\delta := \lfloor \frac{M-u_i}{2} \rfloor$. Otherwise, $\delta := 2\delta$.
- When $\delta > 0$, go to Step 4c. Otherwise, mark operation i to reflect that it has been shaved. Each unmarked successor j of i in $\mathcal{G}$ is added to Q, provided that all its predecessors in $\mathcal{G}$ are marked. The corresponding priority is $\alpha_j - u_j$. Go to Step 4.

Otherwise,

- Undo all updates accomplished in Step 4d.
- bisection := true, $M := u_i + \delta$, $\delta := \lfloor \delta/2 \rfloor$.
- When $\delta > 0$, go to Step 4c. Otherwise, mark operation i to reflect that it has been shaved. Each unmarked successor j of i in $\mathcal{G}$ is added to Q, provided that all its predecessors in $\mathcal{G}$ are marked. The corresponding priority is $\alpha_j - u_j$. Go to Step 4.

For *Heuristic Tail Shave*, the available operation with the largest gap between $UB - q_i - p_i$ and β_i is shaved first. Initially, only the last operation of each job is available.

7.5.4 *Heuristic Shave* and *Heuristic Double Shave*

Heuristic Shave is the same as *MS Shave*, except that *MS Head Shave* is replaced by *Heuristic Head Shave* and *MS Tail Shave* is replaced by *Heuristic Tail Shave*.

Heuristic Double Shave is completely analogous to *MS Double Shave*.

7.6 Results

We restrict our attention to double shaving. Compared to *MS Double Shave*, our *Heuristic Double Shave* has the following additional parameters: a list $\mathcal{L}$ of good schedules, a value $k_{\text{arc}} \geq 1$, and a value $k_{\text{int}} \geq 1$. The best k_{arc} schedules in $\mathcal{L}$ determine which heuristic machine arcs are added to $\mathcal{G}$, and the best k_{int} schedules in $\mathcal{L}$ determine the heuristic start intervals.

In order to find reasonable parameter settings, we will use the set of problem instances in Table 7.2 in a number of computational experiments. For each of these test instances the optimum makespan is known. Our goal is to find appropriate parameters for these test instances first. In this way, we hope to find general settings that work well on other, open, instances. Then, in Section 7.6.3, we apply *Heuristic Double Shave* with the selected parameters to a number of open problems instances.

For each of the test instances, we generated a number of good schedules with multiple runs of the tabu search heuristic due to Nowicki and Smutnicki [51]. The running times varied from a couple of minutes to several hours. These running times are not included in our results.

We are interested in the effect of the size of k_{arc}, the size of k_{int}, and the quality of the solutions in $\mathcal{L}$ on the running time of *Heuristic Double Shave*. We consider two cases: $\mathcal{L}$ contains only optima, and $\mathcal{L}$ contains no optima.

7.6.1 $\mathcal{L}$ contains only optima

For each test instance we have a list $\mathcal{L} = (L_1, L_2, \dots)$ of different optima with makespan $C_{\max}$. The first k_{arc} solutions in $\mathcal{L}$ determine which machine arcs are added to the precedence graph $\mathcal{G}$. When $k_{\text{arc}} = 0$, $\mathcal{G}$ contains only job arcs. The first k_{int} solutions in $\mathcal{L}$ determine the heuristic start intervals. When $k_{\text{int}} \geq 1$, the priority of an available operation i is determined by the gap between $[u_i, v_i]$ and the heuristic start interval $[\alpha_i, \beta_i]$; when $k_{\text{int}} = 0$, the priority is determined by the release time (just like Martin and Shmoys did). Even when we do not employ common machine arcs and heuristic start intervals, i.e., $k_{\text{arc}} = 0$ and $k_{\text{int}} = 0$, there is a difference between the shave order of Martin and Shmoys and our shave order. The shave order of Martin and Shmoys is completely determined beforehand, while our shave order is dynamic in the sense that priorities can change during shaving as a result of updating heads and tails.

We carried out some computational experiments for *Heuristic Double Shave* with different values for k_{arc} and k_{int}. We put $UB = c(L_1) = C_{\max}$ and the parameter T is equal to the optimum minus one. The results are given in Table 7.3. We conclude that in most cases where $k_{\text{int}} = 0$, the running times are approximately the same as for *MS Double Shave* (see Table 7.2). Furthermore, most cases where $k_{\text{arc}} = 0$ and $k_{\text{int}} \geq 2$ give very bad results. We explain this as follows. Consider the different schedules in $\mathcal{L}$. When an operation i is early in its job (relatively small u_i), but always late on its machine (relatively large α_i), its priority is very high, due to the large gap between $[u_i, v_i]$ and the heuristic start interval $[\alpha_i, \beta_i]$. As a result,

instance/T	$k_{\text{arc}}/k_{\text{int}}$	0	2	4	8	16
mt10/929	0	189	158	260	261	260
	2	158	149	163	163	163
	4	157	150	150	150	150
	8	159	151	150	151	151
	16	158	150	150	150	150
abz5/1,233	0	284	727	727	729	730
	2	170	147	147	147	147
	4	165	143	143	143	143
	8	167	144	144	145	145
	16	164	141	141	142	142
la19/841	0	220	260	259	259	259
	2	216	121	122	121	122
	4	215	121	121	121	121
	8	214	121	122	121	121
	16	216	122	122	122	122
la24/934	0	4,957	7,827	7,749	7,624	7,614
	2	4,931	5,453	5,441	5,489	5,489
	4	4,999	5,166	5,137	5,173	5,178
	8	5,098	5,225	5,185	5,224	5,224
	16	5,182	5,316	5,322	5,317	5,311
la25/976	0	5,078	10,506	10,508	7,397	7,415
	2	6,933	6,292	6,299	6,060	6,062
	4	6,912	6,300	6,313	6,042	6,042
	8	7,162	7,230	7,244	7,331	7,389
	16	7,134	7,223	7,245	7,288	7,265
la36/1,267	0	1,423	3,813	2,336	3,296	3,312
	2	1,345	490	459	492	490
	4	1,332	533	466	528	528
	8	1,320	579	914	578	577
	16	1,311	619	952	616	615

Table 7.3: The running time in seconds of *Heuristic Double Shave* for a number of problem instances and parameters. The list $\mathcal{L}$ contains only optima.

Heuristic Double Shave is too greedy. This is prevented by adding common machine arcs to $\mathcal{G}$, which reflect that operation i is relatively late on its ma-

chine. Their presence in $\mathcal{G}$ cause the operation to be delayed in the shaving process by making it available later. According to the results, it appears that both common machine arcs and heuristic start intervals are necessary to improve upon the running times of Martin and Shmoys, i.e., $k_{\text{arc}} \geq 2$ and $k_{\text{int}} \geq 2$. Although larger values of k_{arc} and k_{int} sometimes improve and sometimes deteriorate the running time, we conclude that there is no reason to use more than two solutions. Apparently, two distinct solutions contain approximately the same heuristic information as three or more solutions. When $k_{\text{arc}} = 2$ and $k_{\text{int}} = 2$, the running time of *Heuristic Double Shave* on abz5 and la19 is about half the running time of *MS Double Shave*. On la36, *Heuristic Double Shave* is approximately three times faster than *MS Double Shave*. For mt10, la24 and la25, the running times are roughly the same as for *MS Double Shave*.

For each problem instance, we determine the average running time of *Heuristic Double Shave* over a number of different pairs of optima $\mathcal{L} = (L_1, L_2)$. The results are given in Table 7.4. Apparently, the running time can be quite dependent on the pair of optima that is used. Nevertheless, we conclude that the results are quite stable for the different pairs of optima.

instance	mt10	abz5	la19	la24	la25	la36
run 1	147	157	122	5,176	5,897	447
run 2	126	159	219	5,114	6,693	433
run 3	148	157	120	5,104	6,970	881
run 4	145	157	116	5,099	7,111	904
run 5	126	157	220	5,100	5,821	496
run 6	148	157	122	5,188	5,833	430
run 7	147	157	120	5,084	6,032	529
run 8	146	159	119	5,326	6,208	494
run 9	149	167	116	5,135	6,312	490
run 10	126	142	219	5,124	7,092	488
average	141	157	149	5,145	6,397	559

Table 7.4: The running time of *Heuristic Double Shave* using different pairs of optima.

7.6.2 $\mathcal{L}$ contains no optima

In this section, we investigate the influence of the quality of the solutions in $\mathcal{L}$ on the running time of *Heuristic Double Shave*. For each test instance,

let $C_{\max}$ denote the makespan of an optimum solution. We consider the following cases: the best solution in $\mathcal{L}$ has a makespan of $C_{\max}+1$ and the best solution in $\mathcal{L}$ has a makespan of approximately $C_{\max}+5$. As usual, the value of UB is equal to the makespan of the best solution in $\mathcal{L}$, although we know the optimum. Parameter T is equal to UB minus one. We carried out a number of computational experiments for $k_{\text{arc}}=2$ and $k_{\text{int}}=2$.

Unfortunately, for only two problem instances we had a number of solutions with a makespan of $C_{\max}+1$. For these instances, the results are given in Table 7.5. Compared to Table 7.4, the average result for la19 is slightly worse, while the average result of la25 is slightly better. We conclude that there is no evidence that sub-optimality of the solutions harms the usefulness of the heuristic information contained in $\mathcal{L}$.

instance	la19	la25
UB	843	978
run 1	128	5,559
run 2	229	5,657
run 3	129	6,763
run 4	129	6,669
run 5	124	5,940
run 6	122	6,569
run 7	232	6,004
run 8	129	6,736
run 9	127	6,222
run 10	232	5,450
average	158	6,157

Table 7.5: The running time of *Heuristic Double Shave* using different pairs of solutions of cost $UB = C_{\max}+1$.

In Table 7.6, we consider the case where solutions of cost approximately $C_{\max}+5$ are employed. Compared to Table 7.4, we conclude that for most instances *Heuristic Double Shave* performs significantly worse. Only for la24 and la25 it perform slightly better, implying that luck also plays a role. Therefore, we conclude that it is likely that the quality of the solutions in $\mathcal{L}$ does affect the quality of the heuristic information contained in them. Nevertheless, the results are in most cases still better than the corresponding results of *MS Double Shave*.

instance	mt10	abz5	la19	la24	la25	la36
UB	937	1,238	847	940	982	1,271
run 1	218	179	247	5,097	6,056	329
run 2	130	183	259	4,830	6,158	521
run 3	130	333	258	3,583	6,372	610
run 4	130	302	247	4,792	4,844	503
run 5	133	302	259	4,814	6,359	494
run 6	132	170	261	3,577	6,092	598
run 7	130	381	249	4,392	7,259	934
run 8	221	302	259	5,350	6,330	491
run 9	130	302	248	5,281	6,680	2,028
run 10	161	301	247	4,716	5,985	777
average	152	276	253	4,643	6,214	729

Table 7.6: The running time of *Heuristic Double Shave* using different pairs of solutions of cost UB, which is approximately equal to $C_{\max} + 5$.

7.6.3 New lower bounds

We consider a number of open problem instances proposed by Taillard [66]. We concentrate on Taillard's benchmark instances ta11 to ta20, having 20 jobs and 15 machines. At the time of writing, only ta14 had been solved.

For some instances we do not have solutions with the best known upper bound. In that case, we use a number of solutions that are slightly worse. As usual, the value UB is set to the best solution in $\mathcal{L}$, and not to the best known upper bound. Notice that the solutions in $\mathcal{L}$ do not all have the same cost. In Table 7.7 new lower bounds are given, which are derived by *Heuristic Double Shave* with $k_{\text{arc}} = 2$ and $k_{\text{int}} = 2$. The running times are given in seconds on a Pentium II/333 MHz PC. All solutions are obtained with the tabu search algorithm due to Nowicki and Smutnicki [51]. The running times of tabu search, which ranged between a couple of hours and several days, are not included in the table.

For ta17, *Heuristic Double Shave* proved that solutions of cost at most 1,461 cannot exist, hence 1,462 is a lower bound. Furthermore, it could not derive infeasibility for $T = 1{,}462$; it took 114,583 seconds to derive that solutions of cost 1,462 could exist. For ta18, our heuristic double shaver could not derive infeasibility for $T = 1{,}369$, the best known lower bound from the literature. It took 237,540 seconds to derive that solutions of cost 1,369 could exist.

instance	old LB	UB	$c(L_1)$	$c(L_2)$	new LB	time
ta11	1,321	1,364	1,367	1,373	1,323	371,160
ta12	1,321	1,367	1,377	1,377	1,351	145,513
ta13	1,271	1,350	1,350	1,350	1,282	67,181
ta15	1,293	1,342	1,342	1,342	1,304	30,886
ta16	1,300	1,362	1,368	1,368	1,302	69,338
ta17	1,458	1,464	1,476	1,476	1,462	14,295
ta19	1,276	1,341	1,341	1,344	1,297	31,054
ta20	1,316	1,353	1,359	1,359	1,318	282,814

Table 7.7: New lower bounds on some Taillard instances with 20 jobs and 15 machines using two good solutions.

Obviously, the same lower bounds could be derived by *MS Double Shave*, but it could take considerably longer.

7.7 Conclusion

Given is a list $\mathcal{L}$ of good solutions to an instance of a job shop scheduling problem. We distill heuristic information from $\mathcal{L}$, which is used in an attempt to improve upon the running time of the shaving method proposed by Martin and Shmoys. The heuristic information is employed to determine the order in which the operations are shaved. The job arcs and the machine arcs that are common to the solutions in $\mathcal{L}$ determine a partial order on the operations. Ties between available operations are broken by considering the heuristic start intervals of the operations in the solutions in $\mathcal{L}$.

Experiments indicate that both common machine arcs and heuristic start intervals are required to improve upon the running time of Martin and Shmoys. It is sufficient to have two good solutions. Although the running time generally increases as the quality of the solutions in $\mathcal{L}$ deteriorates, the results are quite stable for different pairs of good solutions. For three out of six problem instances, the heuristic shave algorithm is two to three times faster than the shaving algorithm due to Martin and Shmoys. Furthermore, with the heuristic shave algorithm, we found new lower bounds to a number of open problem instances.

Chapter 8

Conclusion

The research reported in this thesis is motivated by the common defect of many local search heuristics that they tend to get stuck. When a local search heuristics *gets stuck*, a lot of time is wasted in an unfruitful part of the search space.

In an attempt to overcome this weakness, we experimented with multiple independent runs. Each run was started in a different solution, which was constructed from scratch. It turned out that good solutions to a combinatorial optimization problem have many building elements in common, frequently up to 80 or 90 percent. These so-called *commonalities* are interpreted as being "probably right" in the sense that it is expected that they are contained in almost every good solution. The commonalities of a number of good solutions form the basis of a restart mechanism for local search heuristics that takes advantage of the past by using information obtained by previous runs. The idea is to perform multiple shorter runs instead of a single long run. During the search, a list of good solutions is maintained. The initial solution of each subsequent run is obtained by perturbing the best solution found so far in such a way that many commonalities of the solutions in the list are preserved. In this way, the search is immediately directed to an interesting part of the search space. The run length and the amount of perturbation are parameters of this *commonality-preserving restart mechanism.*

For a number of combinatorial optimization problems and local search heuristics we implemented this commonality-preserving restart mechanism. The commonality-preserving restart mechanism was compared to a number of other strategies that are based upon multiple shorter runs and to a single long run. We compared the averages of the quality of the solutions that

are found and the associated standard deviation. The average quality is an indication of the effectiveness, while the standard deviation is a measure of the robustness. All computational experiments were carried out on a time-equivalent basis. The results are summarized below.

Traveling salesman. The Chained Lin-Kernighan heuristic is one of the most successful heuristics for symmetric traveling salesman problems. Experiments clearly indicate that Chained Lin-Kernighan tends to get stuck. With respect to the average result, our commonality-preserving restart mechanism is better than a single long run, better than multiple independent runs, and better than a restart mechanism that perturbs the best solution found so far at random. The standard deviation of each of the three strategies that are based upon multiple runs was approximately the same and often considerably smaller than that of a single long run.

Whizzkids '96. We implemented a tabu search heuristic that employs Chained Lin-Kernighan as a path improvement heuristic. According to some computational experiments, the tabu search heuristic did not seem to get stuck. Nevertheless, our commonality-preserving restart mechanism appeared to be more effective than multiple independent runs and more effective than a single long run. The independent runs proved to be more robust than the other strategies. As each of the strategies frequently produced solutions with the best known upper bound, it seems that there is not much room for further improvement.

Job shop scheduling. The tabu search heuristic of Nowicki and Smutnicki is one of the most effective heuristics for job shop scheduling problems. We compared a commonality-preserving restart mechanism to multiple independent runs and to a backtrack mechanism. Our commonality-preserving restarts are more effective than independent runs, approximately as robust as independent runs, more robust than backtracking, and slightly less effective than backtracking. During the experiments, new upper bounds were found for a large number of benchmark instances.

Whizzkids '97. We implemented a tabu search heuristic based upon the tabu search heuristic of Nowicki and Smutnicki for the job shop scheduling problem. In our computational experiments, it was clear that the tabu search heuristic tends to get stuck. Our commonality-preserving restart mechanism is more effective than a single long run and more effective than multiple independent runs. The standard deviation is approximately the same for each of the strategies that we considered. As the problem is solved, the high quality of the solutions that are obtained by each of the strategies implies that there is not much room for further improvement.

Although the details of the commonality-preserving restart mechanisms

vary from one case to the other, we conclude that they are quite successful.

Multiple independent runs and backtracking are extreme cases of a commonality-preserving restart mechanism: perturbing all building elements of the best solution found so far corresponds to multiple independent runs, while perturbing none of the building elements corresponds to a simple form of backtracking. Performing multiple independent runs in which each run is started in a different solution is a simple *diversification mechanism*, while backtracking is an *intensification mechanism*. Therefore, by varying the amount of perturbation, a balance between intensification and diversification is obtained. Similarly, the amount of perturbation determines the balance between the effectiveness of backtracking and the robustness of multiple independent runs.

Commonalities are also useful in finding lower bounds, which was demonstrated in the context of the shaving algorithm, a lower bounding method for job shop scheduling due to Martin and Shmoys. The commonalities of a small number of good solutions were employed, together with some other heuristic information derived from the solutions, to determine an efficient order in which the operations are processed by the algorithm. For three out of six problem instances, our heuristic shave algorithm was two to three times faster than the shaving algorithm due to Martin and Shmoys. In some cases, it was slightly slower. Furthermore, with the heuristic shave algorithm, we found new lower bounds to a number of open problem instances.

Given the results of the commonality-preserving restart mechanisms and the results of the heuristic shaving algorithm, we conclude that commonalities have useful applications in combinatorial optimization.

Bibliography

[1] E.H.L. Aarts, P.J.M. van Laarhoven (1985). Statistical cooling: a general approach to combinatorial optimization problems. *Philips Journal of Research* 40, pp. 193-226.

[2] E.H.L. Aarts, J.K. Lenstra (eds.) (1997). *Local Search in Combinatorial Optimization*, Wiley, Chichester.

[3] D. Applegate, R.E. Bixby, V. Chvátal, W. Cook (1999). `http://www.caam.rice.edu/~keck/concorde.html`.

[4] D. Applegate, R.E. Bixby, V. Chvátal, W. Cook (1999). *Finding tours in the TSP.* Unpublished manuscript.

[5] D. Applegate, W. Cook (1991). A computational study of the job shop scheduling problem. *ORSA Journal on Computing* 3, pp. 149-156.

[6] D. Applegate, W. Cook, A. Rohe (1999). *Chained Lin-Kernighan for large traveling salesman problems.* Unpublished manuscript.

[7] J.R. Araque, G. Kudva, T.L. Morin, J.F. Pekny (1994). A branch-and-cut algorithm for the vehicle routing problem. *Annals of Operations Research* 50, pp. 37-59.

[8] F. Bock (1958). *An algorithm for solving "traveling salesman" and related network optimization problems.* Internal report, Armour Research Foundation, Chicago, Illinois.

[9] O. Borůvka (1926). On a certain minimal problem. *Práce Moravské Přírodovědecké Společnosti* 3, pp. 37-58 (in Czech).

[10] P. Brucker, B. Jurisch (1993). A new lower bound for the job-shop scheduling problem. *European Journal of Operational Research* 64, pp. 156-167.

[11] J. Carlier, E. Pinson (1989). An algorithm for solving the job-shop problem. *Management Science* 35, pp. 164-176.

[12] J. Carlier, E. Pinson (1990). A practical use of Jackson's preemptive schedule for solving the job shop problem. *Annals of Operations Research* 26, pp. 269-287.

[13] J. Carlier, E. Pinson (1994). Adjustment of heads and tails for the job-shop problem. *European Journal of Operational Research* 78, pp. 146-161.

[14] V. Černý (1985). Thermodynamical approach to the traveling salesman problem: an efficient simulation algorithm. *Journal of Optimization Theory and Applications* 45, pp. 41-51.

[15] N. Christofides, E. Hadjiconstantinou, A. Mingozzi (1993). *A new exact algorithm for the vehicle routing problem based on q-path and k-shortest path relaxations*, Working paper, Management School, Imperial College, London.

[16] G. Clarke, J.W. Wright (1964). Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research* 12, pp. 568-581.

[17] G. Cornuéjols, F. Harche (1993). Polyhedral study of the capacitated vehicle routing problem. *Mathematical Programming* 60, pp. 21-52.

[18] G.A. Croes (1958). A method for solving traveling salesman problems. *Operations Research* 6, pp. 791-812.

[19] H.M.M. ten Eikelder, B.J.M. Aarts, M.G.A. Verhoeven, E.H.L. Aarts (1999). Sequential and parallel local search algorithms for job shop scheduling. S. Voss, S. Martello, I.H. Osman, C. Roucairol (eds.). *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*, Kluwer Academic Publishers, Norwell, Massachusetts, pp. 359-371.

[20] H. Fisher, G.L. Thompson (1963). Probabilistic learning combinations of local job-shop scheduling rules. J.F. Muth, G.L. Thompson (eds.). *Industrial Scheduling*, Prentice Hall, Englewood Cliffs, New Jersey, pp. 225-251.

[21] M.L. Fisher (1995). Vehicle routing. M.O. Ball, T.L. Magnanti, C.L. Monma, G.L. Nemhauser (eds.). *Network Routing*, Handbooks in Operations Research and Management Science, Volume 8, North-Holland, Amsterdam, pp. 1-33.

[22] M.L. Fisher, B.J. Lageweg, J.K. Lenstra, A.H.G. Rinnooy Kan (1983). Surrogate duality relaxation for job shop scheduling. *Discrete Applied Mathematics* 5, pp. 65-75.

[23] M.R. Garey, R.L. Graham, D.S. Johnson (1977). The complexity of computing Steiner minimal trees. *SIAM Journal on Applied Mathematics* 32, pp. 835-859.

[24] M.R. Garey, D.S. Johnson (1979). *Computers and Intractability: a Guide to the Theory of NP-completeness*. Freeman, San Francisco, California.

[25] M. Gendreau, A. Hertz, G. Laporte (1994). A tabu search heuristic for the vehicle routing problem. *Management Science* 40, pp. 1276-1290.

[26] M. Gendreau, G. Laporte, J.-Y. Potvin (1997). Vehicle routing: modern heuristics. E.H.L. Aarts, J.K. Lenstra (eds.). *Local Search in Combinatorial Optimization*, Wiley, Chichester, pp. 311-336.

[27] F. Glover (1986). Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research* 13, pp. 533-549.

[28] F. Glover (1989). Tabu search: part I. *ORSA Journal on Computing* 1, pp. 190-206.

[29] F. Glover (1990). Tabu search: part II. *ORSA Journal on Computing* 2, pp. 4-32.

[30] F. Glover, M. Laguna (1997). *Tabu Search*. Kluwer Academic Publishers, Norwell, Massachusetts.

[31] R.L. Graham, E.L. Lawler, J.K. Lenstra, A.H.G Rinnooy Kan (1979). Optimization and approximation in deterministic sequencing and scheduling. *Annals of Discrete Mathematics* 5, pp. 287-326.

[32] B.L. Golden, G. Laporte, É.D. Taillard (1997). An adaptive memory heuristic for a class of vehicle routing problems with minmax objective. *Computers & Operations Research* 24, pp. 445-452.

[33] J.H. Holland (1975). *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, Michigan.

[34] J.R. Jackson (1955). *Scheduling a production line to minimize maximum tardiness.* Research Report No. 43, Management Science Research Project, University of California, Los Angeles.

[35] D.S. Johnson, L.A. McGeoch (1997). The traveling salesman problem: a case study. E.H.L. Aarts, J.K. Lenstra (eds.). *Local Search in Combinatorial Optimization*, Wiley, Chichester, pp. 215-310.

[36] M. Jünger, G. Reinelt, G.Rinaldi (1995). The traveling salesman problem. M.O. Ball, T.L. Magnanti, C.L. Monma, G.L. Nemhauser (eds.). *Network Models*, Handbooks in Operations Research and Management Science, Volume 7, North-Holland, Amsterdam, pp. 225-330.

[37] B.W. Kernighan, S. Lin (1970). A efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal* 49, pp. 291-307.

[38] S. Kirkpatrick, C.D. Gelatt Jr., M.P. Vecchi (1983). Optimization by simulated annealing. *Science* 220, pp. 671-680.

[39] P.J.M. van Laarhoven, E.H.L. Aarts, J.K. Lenstra (1992). Job shop scheduling by simulated annealing. *Operations Research* 40, pp. 113-125.

[40] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, D.B. Shmoys (eds.) (1985). *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, Wiley, Chichester.

[41] S. Lin (1965). Computer solutions of the traveling salesman problem. *Bell System Technical Journal* 44, pp. 2245-2269.

[42] S. Lin, B.W. Kernighan (1973). An effective heuristic algorithm for the traveling-salesman problem. *Operations Research* 21, pp. 498-516.

[43] K.-T. Mak, A.J. Morton (1993). A modified Lin-Kernighan traveling-salesman heuristic. *Operations Research Letters* 13, pp. 127-132.

[44] H. C. Matsuo, J. Suh, R. S. Sullivan (1988). *A controlled search simulated annealing method for the general jobshop scheduling problem.* Working Paper 03-04-88, Department of Management, The University of Texas at Austin, Austin, Texas.

[45] P. MARTIN (1996). *A time-oriented approach to computing optimal schedules for the job-shop scheduling problem.* Ph.D. Thesis, Cornell University, Ithaca, NY, USA.

[46] O. MARTIN, S.W. OTTO, E.W. FELTEN (1991). Large-step Markov chains for the traveling salesman problem. *Complex Systems* 5, pp. 299-326.

[47] O. MARTIN, S.W. OTTO, E.W. FELTEN (1992). Large-step Markov chains for the TSP incorporating local search heuristics. *Operations Research Letters* 11, pp. 219-224.

[48] O.C. MARTIN, S.W. OTTO (1996). Combining simulated annealing with local search heuristics. *Annals of Operations Research* 63, pp. 57-75.

[49] P. MARTIN, D.B. SHMOYS (1996). A new approach to computing optimal schedules for the job-shop scheduling problem. W.H. CURNIGHAM, S.T. MCCORMICK, M. QUEYRANNE (EDS.). *Lecture Notes in Computer Science; Integer Programming and Combinatorial Optimization.* Proceedings fifth international IPCO conference, Vancouver, Canada.

[50] T.A.J. NICHOLSON (1965). A sequential method for discrete optimization problems and its application to the assignment, travelling salesman and three scheduling problems. *Journal of the Institute of Mathematics and its Applications* 13, pp. 362-375.

[51] E. NOWICKI, C. SMUTNICKI (1996). A fast taboo search algorithm for the job shop problem. *Management Science* 42, pp. 797-813.

[52] I.H. OSMAN (1993). Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem. *Annals of Operations Research* 41, pp. 421-451.

[53] E.S. PAGE (1965). On Monte Carlo methods in congestion problems: I, searching for an optimum in discrete situations. *Operations Research* 13, pp. 291-299.

[54] C.H. PAPADIMITRIOU, K. STEIGLITZ (1982). *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, New York.

[55] M. PINEDO (1995). *Scheduling: Theory, Algorithms and Systems*, Prentice Hall, Englewood Cliffs, New Jersey.

[56] G. Reinelt (1991). TSPLIB: a traveling salesman problem library. *ORSA Journal on Computing* 3, pp. 376-384.

[57] G. Reinelt (1994). *The Traveling Salesman: Computational Solutions for TSP Applications.* Springer-Verlag, Berlin.

[58] G. Reinelt (1995). *TSPLIB 95.* Research Report, Institut für Angewandte Mathematik, Universität Heidelberg.

[59] G. Reinelt (1999). TSPLIB upper bounds. `http://www.iwr.uni-heidelberg.de/iwr/comopt/software/TSPLIB95/STSP.html`.

[60] Y. Rochat, É.D. Taillard (1995). Probabilistic diversification and intensification in local search for vehicle routing. *Journal of Heuristics* 1, pp. 130-147.

[61] C. Rosing, C. ReVelle (1997). Heuristic concentration: two stage solution construction. *European Journal of Operational Research* 97, pp. 75-86.

[62] B. Roy, B. Sussmann (1964). *Les problèmes d'ordonnancement avec contraintes disjonctives.* Note DS No. 9 bis, SEMA, Paris (in French).

[63] R.M.F. Schilham, H.M.M. ten Eikelder (1999). Job shop scheduling by tabu search: a commonality-based restart mechanism. Submitted for publication.

[64] R.M.F. Schilham, H.M.M. ten Eikelder (2000). Commonality-preserving restarts of Chained Lin-Kernighan. Submitted for publication.

[65] R.M.F. Schilham, H.M.M. ten Eikelder (2000). Heuristic shaving for the job shop scheduling problem. Submitted for publication.

[66] É.D. Taillard (1993). Benchmarks for basic scheduling problems. *European Journal of Operational Research* 64, pp. 278-285.

[67] É.D. Taillard (1993). Parallel iterative search methods for vehicle routing problems. *Networks* 23, pp. 254-265.

[68] É.D. Taillard (1994). Parallel taboo search techniques for the job shop scheduling problem. *ORSA Journal on Computing* 6, pp. 108-117.

[69] É.D. TAILLARD (1999). Job shop upper bounds and lower bounds. `http://www.eivd.ch/ina/Collaborateurs/etd/problemes.dir/ordonnancement.dir/jobshop.dir/best_lb_up.txt`.

[70] É.D. TAILLARD, L.-M. GAMBARDELLA, M. GENDREAU, J.-Y. POTVIN (1998). *Adaptive memory programming: a unified view of metaheuristics.* To appear in *European Journal of Operational Research.*

[71] R.J.M. VAESSENS, E.H.L. AARTS, J.K. LENSTRA (1996). Job shop scheduling by local search. *INFORMS Journal on Computing* 8, pp. 302-317.

[72] M. WENNINK (1995). *Algorithmic Support for Automated Planning Boards.* Ph.D. Thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology.

[73] D. DE WERRA, A. HERTZ (1989). Tabu search techniques: a tutorial and an application to neural networks. *OR Spektrum* 11, pp. 131-141.

[74] T. YAMADA, R. NAKANO (1992). A genetic algorithm applicable to large-scale job shop instances. R. MANNER, B. MANDERICK (EDS.). *Parallel Problem Solving from Nature* 2, North-Holland, Amsterdam.

Acknowledgments

First of all, I thank Huub ten Eikelder for his involvement in my work. His enthusiasm and interest were a major motivation for completing the thesis. It was a pleasure to work with him.

Then I would like to express my gratitude to Jan Karel Lenstra for his supervision and detailed proofreading.

I am indebted to Emile Aarts for his encouragement and interesting points of view with regard to my work.

The contribution of David Applegate to the choice of several parameters of the Chained Lin-Kernighan code is greatly acknowledged. His help in making available the source code of the Concorde project and supplying two unpublished manuscripts contributed a lot to this thesis.

Finally, I would like to thank the people at the Max-Planck-Institut für Informatik for making available their valuable LEDA library. This Library of Efficient Data structures and Algorithms has been used extensively in each of the implemented heuristics described in this thesis.

Robin Schilham

Utrecht, December 2000

Curriculum vitae

Robin Schilham was born on June 21, 1970, in Amersfoort, the Netherlands. In 1988 he received his atheneum diploma from the Corderius College in Amersfoort. He started his study computer science at Utrecht University in the same year, where he specialized in statistics and algorithms. In 1994 he graduated after completing his Master's thesis on "Markov chains and the cover time of graphs", under the supervision of prof.dr. Jan van Leeuwen. Subsequently, as he was liable to military service, he spent a year in the army. In 1996 he started as a PhD student at Eindhoven University of Technology under the supervision of dr.ir. Huub ten Eikelder, prof.dr. Jan Karel Lenstra, and prof.dr. Emile Aarts. The results are presented in this thesis.