Parallel Local Search

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Verhoeven, Marcus Gerardus Aldegonda
Parallel Local Search / Marcus Gerardus Aldegonda Verhoeven. -
Eindhoven: Eindhoven University of Technology
Thesis Technische Universiteit Eindhoven. -
With index, ref. - With summary in Dutch
ISBN 90-386-0247-2
Subject headings: parallelism, local search.

druk: Universitaire Drukkerij, Eindhoven
foto omslag: Theo Audenaerd

The work in this thesis has been carried out under the auspices of the research school IPA
(Institute for Programming research and Algorithmics).

©1996 by M.G.A. Verhoeven, Eindhoven, The Netherlands

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without prior permission of the author.

Parallel Local Search

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van
de Rector Magnificus, prof.dr. J.H. van Lint,
voor een commissie aangewezen door het College
van Dekanen in het openbaar te verdedigen op
donderdag 7 maart 1996 om 16.00 uur

door

Marcus Gerardus Aldegonda Verhoeven

geboren te Eindhoven

Dit proefschrift is goedgekeurd
door de promotoren:

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

para Maria de los Milagros

Contents

1

Introduction

A wide variety of problems in practical planning and design situations is concerned with the choice of the best solution from a finite, possibly large, number of alternatives. Many of these problems can be modelled as combinatorial optimization problems. Combinatorial optimization problems are often computationally intractable and, consequently, larger instances of such problems can typically be solved only to proximity.

Local search is a generally applicable approximation technique for combinatorial optimization problems that is able to find good quality solutions, albeit at the expense of substantial running times. It is often argued that parallelism can be used to reduce these running times, which makes it possible to handle larger instances in a given amount of running time, or to find better solutions for a given instance in an equal amount of running time. In this thesis we investigate the potentials of parallel processing for local search.

1.1 Combinatorial optimization

In this section we give a brief introduction to combinatorial optimization. More elaborate introductions can be found in [Papadimitriou & Steiglitz, 1982; Nemhauser & Wolsey, 1988]. A combinatorial optimization problem is either a minimization problem or a maximization problem consisting of a set of problem instances. Without loss of generality, we consider only minimization problems.

Definition 1.1. An *instance* of a combinatorial optimization problem is a pair $(\mathcal{S}, f)$, where $\mathcal{S}$ is a finite set of solutions, and $f : \mathcal{S} \rightarrow \mathbb{Z}$ is a function that gives the cost of a solution. The objective is to find a solution in $\mathcal{S}$ with minimal cost. □

The *size* of an instance I, denoted by $size(I)$, is defined as the number of symbols needed to encode I in a compact way. The finiteness of the set $\mathcal{S}$ suggests that an instance $(\mathcal{S}, f)$ can be solved by examining all solutions and selecting the one with minimal cost. Such an enumeration approach is only practical for very small instances if the size of $\mathcal{S}$, which is denoted by $|\mathcal{S}|$, grows superpolynomially with the instance size. The *time complexity function* $t_A : \mathbb{N} \rightarrow \mathbb{N}$ of an algorithm A gives for each instance size the largest running time needed by A for solving an instance of that size, where running time is measured in the number of elementary operations such as assignments, comparisons, etc. Hence, $t_A(size(I))$ is an upper bound on the running time needed to solve an instance I. To compare algorithms, one is often interested in the order of their complexity functions. A function f is $\mathcal{O}(f')$ with $f, f' : \mathbb{N} \rightarrow \mathbb{N}$ if there exist constants $c, m \in \mathbb{N}$ such that $f(n) \leq c \cdot f'(n)$ for all $n > m$. f is $\Omega(f')$ if there exist constants $c, m \in \mathbb{N}$ such that $f(n) \geq c \cdot f'(n)$ for all $n > m$, and f is $\Theta(f')$ if f is both $\Omega(f')$ and $\mathcal{O}(f')$. An algorithm A is a polynomial-time algorithm if $t_A = \mathcal{O}(f)$ for some polynomial function f. Otherwise A is a superpolynomial-time algorithm. Problems for which a polynomial-time algorithm is known, which implies that instances I can be solved in $size(I)^{\mathcal{O}(1)}$ time, are often called ‘easy’. For many other problems no such algorithms are known despite considerable effort to find them. The difference between these kinds of problems is formalized by the theory of NP-completeness. Garey & Johnson [1979] present an overview of the NP-completeness theory. This theory is based on *decision problems* in which one is asked to determine whether there exists a solution with cost of at most $k \in \mathbb{Z}$. A decision problem has only two possible solutions, either ‘yes’ or ‘no’. With each optimization problem a decision problem can be associated. Next, two classes of problems are introduced, called P and NP, with P $\subseteq$ NP. These classes are used to distinguish between easy and hard problems.

Definition 1.2. P is the class of decision problems for which each instance can be solved by a polynomial-time algorithm. □

A *concise certificate* for an instance I is an amount of data with size polynomial in $size(I)$. The class NP can now be defined as follows.

Definition 1.3. NP is the class of decision problems for which each instance I with answer ‘yes’ has a concise certificate c such that a ‘yes’ answer for I can be verified by a polynomial-time algorithm using c. □

The concept of reducibility has been proven useful for relating the computational complexity of two decision problems.

Definition 1.4. Problem π is *polynomially reducible* to problem π' if a polynomial-time algorithm exists that maps each instance I of π onto instance I' of π' such that I is a 'yes' instance of π if and only if I' is a 'yes' instance of π'. □

If π is polynomially reducible to π', then π' is at least as difficult as π. This leads to a class of problems in NP that can be considered as the most difficult ones in NP. These problems are called *NP-complete*.

Definition 1.5. A problem $\pi \in \mathrm{NP}$ is *NP-complete* if each problem in NP is polynomially reducible to π. The class of NP-complete problems is called NPC. □

Reducibility can be used to prove that a problem π is NP-complete. From Definitions 1.4 and 1.5 it follows that in order to prove that π is NP-complete it suffices to show that $\pi \in \mathrm{NP}$ and that a problem $\pi' \in \mathrm{NPC}$ is polynomially reducible to π. Note that, if one NP-complete problem can be solved in polynomial time, then all NP-complete problems can be solved in polynomial time. However, so far no polynomial-time algorithm has been designed for any NP-complete problem and it is widely believed that NP-complete problems are *intractable*—that is, any algorithm that solves each instance of an NP-complete problem requires superpolynomial running time.

Finally, a problem is *NP-hard* if it is as least as difficult as any problem in NP, i.e., any problem in NP is polynomially reducible to it. Hence, each NP-complete problem is NP-hard, and if the decision variant of a combinatorial optimization problem is NP-complete, then the optimization problem is NP-hard.

Handling NP-hard problems. Two approaches can be distinguished to handle NP-hard problems. Either one searches for optimal solutions, at the risk of very large, possibly excessive, running times, or one is satisfied with relatively quickly obtainable solutions at the risk of sub-optimality. The first approach finds optimal solutions and accepts the possibility of superpolynomial running times. It seeks to achieve as much improvement as possible over straightforward enumeration, because often many solutions can be discarded as non-optimal without enumerating them explicitly. Dynamic programming, branch-and-bound, and branch-and-cut algorithms are examples of such implicit enumeration techniques.

For many NP-hard optimization problems one has to resort to approximation algorithms since larger instances cannot be solved optimally in acceptable running times. Approximation algorithms aim to find good-quality suboptimal solutions in a moderate amount of time. Two classes of approximation algorithms can be distinguished, viz. *constructive* algorithms and *local search* algorithms.

Constructive algorithms build a solution by starting with an empty solution and consecutively adding elements to the partial solution until a complete solution is obtained. Constructive algorithms typically run in polynomial time as each step requires polynomial time and the total number of steps is also bounded polynomially, but they may find solutions of mediocre quality. Local search algorithms constitute a class of approximation algorithms that are based on repeatedly replacing a solution by a neighboring solution. They often find good-quality solutions but may require substantial running times. Local search is discussed in more detail in Chapter 2.

Analysis of algorithms. The quality of an approximation algorithm is judged by the quality of the solutions it produces, its *effectiveness*, and by the time it requires to obtain them, its *efficiency*. The quality of solutions is measured by their *excess ratio*, the relative deviation of the costs of solutions from optimal solutions. Both effectiveness and efficiency can be considered from a worst-case and an average-case point of view. Worst-case analysis gives upper bounds on the average-case performance. Two approaches can be used to study the average-case behavior, namely, probabilistic or empirical analysis. A probabilistic average-case analysis aims at determining the expected performance of an algorithm by presupposing a probability distribution over the set of problem instances. An empirical analysis is based on a large number of numerical experiments on a set of benchmark instances either originating from practice or randomly generated according to some probability distribution. If optimal solutions are not known, the excess ratio is usually computed using a lower bound for the optimal solution, which establishes an upper bound for the empirical average-case performance. For many approximation algorithms a worst-case or average-case analysis is quite complicated and, therefore, one often resorts to an empirical average-case analysis. Another option is to compare the performance of an algorithm with that of other algorithms in order to determine a relative ranking of algorithms. Here it is essential that a challenging set of test instances is available.

1.2 Parallel processing

In this section we discuss some preliminaries for parallel processing that are relevant for the work described in this thesis. A more elaborate introduction into parallel processing can be found in [Bertsekas & Tsitsiklis, 1989].

1.2.1 Parallel machines

We first discuss some issues related to parallel machine architectures. Our discussion is mainly based on the classification of parallel architectures presented by Bertsekas & Tsitsiklis [1989].

Over the years many parallel machine models have been proposed. Unfortunately, no theoretical model effectively describes the full spectrum of parallel machines. Some models are based on technological developments, while others are based on theoretical observations. The most widely used classification of parallel computers is based on the distinction between *single instruction, multiple data* (SIMD) and *multiple instruction, multiple data* (MIMD) machines. In SIMD machines one global instruction is performed at a time, possibly on different data elements. In MIMD machines different instructions on different data elements can be performed simultaneously. Most modern parallel computers belong to the class of MIMD machines.

Based on the timing of instructions on different processing units it is possible to distinguish between *synchronous* and *asynchronous* machines. A synchronous machine uses one global clocking scheme to synchronize instructions among processors. If several local clocking schemes exist in a parallel machine, typically one per processing unit, the system is called asynchronous. SIMD machines are synchronous by definition, while MIMD machines are mainly asynchronous.

The above model does not address the way information is exchanged between processors. There are two possibilities for exchanging information. In *shared-memory* systems processors may read and write in a common memory accessible by each processor. In *message-passing* systems processors exchange data through messages. Message-passing systems can be characterized by their interconnection network topology that describes how processors are connected. The most common topologies are the ring and the torus. In a ring each processor is connected with two other processors such that a cycle is formed. In a torus processors are positioned in a two-dimensional grid. Processors are connected to four other processors, and each row and column forms a ring. Other frequently used topologies are the tree, and the hypercube. Some parallel computing environments permit the usage of *virtual topologies* in which topologies are simulated on the machine at hand in a user transparent way. It is of course important for the system's performance that the virtual topology is mapped efficiently onto the physical topology.

Other distinctions made between parallel machines are based on the amount of data each processor can handle and the number of processors. In *fine-grained* systems each processor can handle only a small amount of data, while in *coarse-grained* systems each processor deals with a large amount of data. *Massively parallel* systems consist of a large number of, usually relatively simple, processors. Massively parallel systems are typically fine-grained.

An important class of machine models in parallel complexity theory is the *parallel random access machine* (PRAM) model. A *concurrent read, concurrent* *write* PRAM is a shared memory machine in which all processors can simultaneously read from and write to the same memory locations. A *concurrent read, exclusive write* PRAM is a shared memory machine in which all processors can simultaneously read from the same memory location, but exclusive access to a memory location is required for writing. Although none of the currently available parallel machines are true PRAM's, this model supports the study of the intrinsic parallelism in problems and provides theoretical lower bounds for the time complexity of many parallel algorithms. The PRAM model can be adapted to capture features of more realistic parallel machines by considering memory partitioning, communication latency, and sparse network topologies, which all result in less powerful variants of the above PRAM model.

Features of a parallel machine at hand, besides the computational power of processors, can have a large influence on the performance of algorithms when executed on that machine. Important properties of a message-passing MIMD machine are the latency for starting up communication, and the throughput of communication between processors, which also depends on the interconnection network topology. For shared-memory MIMD machines the access time to various parts of memory plays an important role. Computational experiments are therefore important in the study of parallelism.

1.2.2 Parallel algorithms

Useful criteria to evaluate a parallel algorithm for a given problem are its *speed-up* and *efficiency*. The speed-up is the running time of the best known sequential algorithm for this problem divided by the parallel running time, and the efficiency is the speed-up divided by the number of processors [Barr & Hickman, 1993]. Another important issue is the *scalability* of an algorithm, which refers to its performance for increasing number of processors.

It is usually required that the output of parallel algorithms is independent of the number of employed processors. In our research, however, the number of employed processors may determine the part of the solution space that is explored, which may affect the output. Furthermore, we consider randomized algorithms for which different runs may require different running times. As the quality of the output of the local search algorithms we study is measured using the relative excess of final solutions over global minima, we define speed-up as follows.

Definition 1.6. Let $t_{A,\epsilon}(1)$ be the average time to find a solution with a relative excess ϵ over the global minimum using sequential algorithm A and let $t_{B,\epsilon}(P)$ be the average time for this using parallel algorithm B on machine M with P processors, then the *speed-up* of B over A on M is given by $t_{A,\epsilon}(1)/t_{B,\epsilon}(P)$. □

Other useful criteria, besides speed-up and efficiency, for judging the quality of the parallel algorithms we study in this thesis are the extend in which these parallel algorithms are able to find better quality solutions than sequential algorithms in a given amount of time. Also the instance sizes that can be handled by sequential and parallel algorithms in a given amount of time may be of interest. These criteria are of course closely related to the speed-up of algorithms as algorithms with a large speed-up can typically find better quality solutions in a given amount of time since more solutions can be explored. The reduction of running times for given instance sizes can also be employed to handle larger instances in a given amount of time.

Two issues are important in designing parallel algorithms with a good speed-up, viz., *communication overhead* and *load balancing*. On most parallel machines communication is costly compared to computation operations, so attention has to be paid to the communication behavior of parallel algorithms. Load balancing is important to achieve that no processor ever becomes idle while others are working.

1.2.3 Parallel complexity

It is often attempted to define a class of problems for which parallel processing can be profitable. A traditional example is the complexity class NC that consists of the problems that can be solved on a PRAM with a polynomial number of processors $n^{\mathcal{O}(1)}$ in polylogarithmic time $(\log n)^{\mathcal{O}(1)}$, i.e., a time bounded by a polynomial in the logarithm of the instance size n. Furthermore, a class of *P-complete* problems is identified that contains problems that are believed to be among the hardest problems in P. A problem is P-complete if it belongs to P and any other problem in P can be transformed to it using polylogarithmic time and polynomially many processors [Greenlaw, Hoover & Ruzzo, 1995]. It is conjectured that the class of P-complete problems is disjoint from the class NC, and consequently it is believed that P-complete problems cannot be solved in polylogarithmic time.

In practical situations one is often interested in parallel algorithms that solve an instance roughly P times faster when using P processors. Such algorithms may still exist for P-complete problems. So the distinction between P-complete problems and problems in NC is inadequate to capture the informal notion of problems that are amenable for parallel processing. Kruskal, Rudolph & Snir [1990] introduce a class EP of efficient parallel algorithms, in which they measure the performance of parallel algorithms relative to that of sequential algorithms. The usual yardstick is the best existing sequential algorithm for a given problem. EP is defined as follows.

Definition 1.7. Let the time complexity of a sequential algorithm and a parallel algorithm using $P(n)$ processors for instances of size n be given by $t(n)$ and

$T(n)$, respectively. An algorithm is polynomially fast and has constant efficiency, if $T(n) = \mathcal{O}(t(n)^{\epsilon})$ with $\epsilon < 1$ and $T(n) \cdot P(n) = \mathcal{O}(t(n))$. The class that contains these algorithms is called EP. $\square$

Although the class EP does not define strict complexity classes for problems since it depends on a particular sequential algorithm, it provides a practically relevant classification of parallel algorithms.

1.3 Thesis outline

The objective of the research described in this thesis is to design parallel local search algorithms that can handle large problem instances and that are competitive with the best known sequential algorithms both with respect to running time and quality of final solutions. For this, we study general approaches that can be applied to a wide variety of problems, as well as tailored approaches in which problem characteristics are exploited. Computational experiments on parallel machines are considered as an important aspect of this research.

The remainder of this thesis is organized as follows. Chapter 2 deals with local search. In Chapter 3 we introduce the concepts for designing parallel local search algorithms. This chapter is based on [Verhoeven & Aarts, 1995a]. Chapter 4 presents some results for the speed-up that can be obtained by performing several independent runs of a local search algorithm in parallel. These results are based on a probabilistic analysis of local search that can also be found in [Ten Eikelder, Verhoeven, Vossen & Aarts, 1996]. Chapters 5, 6, and 7 discuss tailored approaches to design parallel local search algorithms for some well-studied combinatorial optimization problems. In Chapter 5 we study the traveling salesman problem, in Chapter 6 the Steiner tree problem in graphs, and in Chapter 7 job shop scheduling. Chapter 7, furthermore, discusses sequential and parallel local search for resource-constrained project scheduling. These chapters are based on [Verhoeven, Aarts, Van de Sluis & Vaessens, 1992; Verhoeven & Aarts, 1994; Verhoeven, Aarts & Swinkels, 1995], [Verhoeven, Aarts & Severens, 1995], and [Verhoeven & Aarts, 1995b], respectively.

The main conclusion that can be drawn from the work described in this thesis is that local search algorithms for a wide variety of problems can be sped up significantly using parallelism based on hybrids of the approaches to parallel local search proposed in this thesis.

2

Local Search

Local search algorithms constitute a class of approximation algorithms for hard combinatorial optimization problems that are based on the exploration of neighborhoods of solutions. They have shown to be successfully applicable to a wide range of problems, giving good-quality solutions [Aarts & Lenstra, 1996]. The basic local search algorithm is the so-called *iterative improvement* algorithm. An iterative improvement algorithm starts off with an initial solution constructed by some heuristic. Next, the algorithm repeatedly tries to improve the current solution by replacing it with a lower-cost neighbor. If a solution has been reached that has no neighbors with lower cost, a local minimum has been found. An essential concept in local search algorithms is the notion of a neighborhood structure.

Definition 2.1. Let $(\mathcal{S}, f)$ be an instance of a combinatorial optimization problem. Then a *neighborhood structure* $\mathcal{N} : \mathcal{S} \to \mathcal{P}(\mathcal{S})$ assigns to each solution in $\mathcal{S}$ a set of solutions, called a *neighborhood*. A solution $s \in \mathcal{S}$ is a *local minimum* of $\mathcal{N}$ if $f(s') \geq f(s)$ for each $s' \in \mathcal{N}(s)$. □

The neighbors of a solution are not given explicitly but are to be constructed by a function. Most neighborhoods are based on the replacement of a few elements that constitute a solution. For this we introduce the following definition in which we assume that solutions are sets. The set of *building elements* that constitute solutions in $\mathcal{S}$ is then given by $\mathcal{E} = \{e \mid e \in s \wedge s \in \mathcal{S}\}$.

```
proc Iterative_Improvement ($s : \mathcal{S}$)
  var $w : \mathcal{P}(\mathcal{S})$;
  begin
    $w := \emptyset$ ;
    while $\mathcal{N}(s) \setminus w \neq \emptyset$ do
      $s' \in \mathcal{N}(s) \setminus w$ ;
      if $f(s') \geq f(s)$ then $w := w \cup \{s'\}$
      else $s := s'$; $w := \emptyset$ fi
    od $\{s$ is a local minimum of $\mathcal{N}\}$
  end
```

Figure 2.1: An iterative improvement algorithm.

Definition 2.2. Let the *exchange function* $\tau_r : \mathcal{S} \times \mathcal{E}^r \rightarrow \mathcal{S}$ be a partial function with $r \in \mathbb{N}$. Then, an r-exchange neighborhood structure $\mathcal{N}_r$ associated with τ_r is defined by $\mathcal{N}_r(s) = \{\tau_r(s, e_1, \ldots, e_r) \mid e_1, \ldots, e_r \in s\}$ for $s \in \mathcal{S}$. □

According to Definition 2.2 neighbors of a solution s can be constructed by applying the exchange function to s and a subset $e_1, \ldots, e_r$ of the building elements of s. Below, we give an example of an exchange function and a neighborhood for one of the best-known combinatorial optimization problems, viz. the traveling salesman problem (TSP). In the TSP a salesman wishes to visit all cities in a given set once and return to the starting point, in such a way that the total distance covered is as short as possible.

Example 2.1 (traveling salesman problem). Given is a complete weighted graph $(V, V \times V)$, where V is a set of N vertices and $d_{ij} \in \mathbb{N}$ gives the distance between each $i, j \in V$. A *tour t* is a set of N directed edges $\{e_1, \ldots, e_N\}$ that constitutes a *Hamiltonian cycle*, a cycle that visits each vertex in this graph precisely once. The solution space $\mathcal{S}$ of a TSP instance is the set of all tours. The cost function f is given by

$$f(t) = \sum_{(i,j) \in t} d_{ij}.$$

The problem is to find a tour $t \in \mathcal{S}$ for which $f(t)$ is minimal. The exchange function $\tau_2 : \mathcal{S} \times (V \times V)^2 \rightarrow \mathcal{S}$ gives for each tour a new tour obtained by replacing two edges with two other edges. The well-known 2-exchange neighborhood of Lin [1965] is defined by $\mathcal{N}_2(t) = \{\tau_2(t, e_i, e_j) \mid 1 \leq i < j \leq N\}$. □

Figure 2.1 presents a schematic description of an iterative improvement algorithm. A single iteration of an iterative improvement algorithm, which we call a *step*, consists of the following actions. First, a neighbor is constructed by applying the exchange function to the current solution. Subsequently, the cost of the neighbor is evaluated. If the neighbor has lower cost, it replaces the current solution by effectuating the proposed exchange. Otherwise the search is continued with the old current solution. A *pivoting rule* [Yannakakis, 1990] determines which neighbor will be the new current solution. Well-known pivoting rules are *first improvement*, which replaces the current solution with the first lower-cost neighbor that is found, and *best improvement*, which uses the solution that has lowest cost among all neighbors.

For practical reasons it is often required that each local search step can be done in polynomial time. For this it is necessary that neighbors can be selected in polynomial time, which requires that the exchange function associated with a neighborhood structure has polynomial time complexity, and that the cost difference $f(s') - f(s)$ of two neighboring solutions s and s' can be computed in polynomial time.

2.1 Computational complexity of local search

Unless a neighborhood is *exact*, i.e., each local minimum is a global minimum, it is generally not possible to give a fixed upper bound on the relative excess of local minima. Moreover, for the iterative improvement algorithm of Figure 2.1 it is in general not possible to give a non-trivial upper bound on the number of steps needed to find a local minimum. Examples are known of problem instances for which the number of steps cannot be bounded by a polynomial in the size of the instance. Therefore, the question has been raised whether this is a general property of local search algorithms. Johnson, Papadimitriou & Yannakakis [1988] have formalized the question of the worst-case behavior of local search algorithms by introducing a new complexity class PLS. A *local search problem* Π is a set of problem instances of which each instance is characterized by a triple $(\mathcal{S}, f, \mathcal{N})$, where $\mathcal{S}$ is the set of solutions, f the cost function, and $\mathcal{N}$ the neighborhood structure. The question is to find a local optimum of $\mathcal{N}$. The class PLS is then defined as follows.

Definition 2.3. A local search problem Π is in PLS if polynomially computable functions g and g' exist such that for an instance I of Π, $g(I)$ returns a start solution, $g'(I, s)$ returns a solution $s' \in \mathcal{N}(s)$ with $f(s') < f(s)$ for $s \in \mathcal{S}$ and if no such solution exists, g' returns s, which is then a local minimum of $\mathcal{N}$. □

Informally stated, PLS is the class of local search problems for which each local search step requires polynomial time. The notion of PLS-reducibility has been introduced in order to define PLS-complete problems.

Definition 2.4. A local search problem Π is *PLS-reducible* to a local search problem Π' if there exists a polynomially computable function h that maps instances I of Π onto instances I' of Π' and a polynomial function h' that maps pairs of the form (solution of $h(I)$, I) onto solutions of I such that, if s is a local optimum for $h(I)$, then $h'(s, I)$ is locally optimal for I. □

Definition 2.5. A local search problem $\Pi \in$ PLS is *PLS-complete* if each problem in PLS is PLS-reducible to Π. □

To prove that a local search problem Π is PLS-complete it suffices to show that Π is included in PLS and that there exists a known PLS-complete problem Π' that can be PLS-reduced to Π. The generic PLS-complete problem is FLIP. The class of PLS-complete problems contains the hardest problems in PLS, and if one of these problems can be solved in polynomial time, then all others can. It is, however, conjectured that $\mathrm{P_s}$ is a strict subset of PLS, where $\mathrm{P_s}$ is the class of search problems that can be solved in polynomial time [Yannakakis, 1996]. Consequently, if this conjecture holds, superpolynomial running times might be required by any algorithm to find a local optimum for the PLS-complete problems.

2.2 Local search variants

A neighborhood structure imposes a directed graph on the solution space. The vertices of this *neighborhood graph* are the solutions, and there is an edge from a vertex to another vertex if this vertex is a neighbor of the first vertex. The course of a local search algorithm is characterized by a walk in this graph. Starting in some vertex, the costs of neighbors are evaluated, and a step to some neighbor is made. This process is repeated until some stop criterion is met. The effectiveness of local search depends on both the structure of the neighborhood graph, which is determined by the neighborhood structure, and the way the neighborhood graph is traversed, which is determined by the type of local search that is used. A neighborhood is *sufficiently connected* if there is a path in the neighborhood graph to an optimal solution from any solution, and it is *completely connected* if there is a path to each solution from any other solution.

A drawback of iterative improvement is that it may get stuck in poor-quality local minima. To overcome this drawback, one can either modify the neighborhood structure, or change the way the neighborhood graph is traversed. The first approach is problem specific, whereas the latter one is less problem dependent.

Most neighborhood structures are based on exchange functions that modify only a few elements, which results in small neighborhoods. In order to avoid the risk of getting stuck in a poor-quality local minimum, one may increase the size of the neighborhood by exchanging more elements. Another option is to exploit characteristics of the problem at hand to tailor the neighborhood to this problem. Examples of this approach are *variable-depth* neighborhood structures [Kernighan & Lin, 1970; Lin & Kernighan, 1973] and the quite similar *ejection chain* approach in [Reeves, 1993]. Variable-depth neighborhoods are based on a simple exchange function. A neighbor is obtained by constructing a sequence of exchanges, instead of a single exchange. An algorithm that uses a variable-depth neighborhood is often no longer regarded as a basic iterative improvement algorithm since it actually generates a sequence of solutions, which may contain solutions with higher cost than a preceding solution. The solution that eventually replaces the current solution is the solution in this sequence with minimal cost. The algorithm stops if this neighbor has higher cost than the current solution.

Another approach to escape from local minima is to accept neighbors with cost higher than that of the current solution. We mention the following local search variants, sometimes called *meta heuristics*, which all traverse the neighborhood graph in a different way to escape from local minima.

Iterated local search [Johnson, 1990; Martin, Otto & Felten, 1991] uses two neighborhoods $\mathcal{N}$ and $\mathcal{N}'$. The search starts with the first neighborhood $\mathcal{N}$ until a local minimum of $\mathcal{N}$ is found. If this local minimum has lower cost than the best solution found so far, this local minimum replaces it. Then, a neighbor of the best solution found so far is chosen from a second neighborhood $\mathcal{N}'$. Subsequently, the search is restarted from this neighbor using the first neighborhood. In this way local search is performed at two levels. At the lower level regular local search steps are made in the neighborhood graph induced by $\mathcal{N}$, but at the higher level large steps between local minima of $\mathcal{N}$ are made.

Tabu search [Glover, 1989; Glover, Taillard & De Werra, 1993] tries to direct the search into unexplored areas of the solution space once a local minimum is found, by memorizing the course of the search. To this end, a finite list of the most recently visited solutions, the *tabu list*, is constructed, whose length is given by the *tabu tenure*. In each step of a tabu search algorithm a neighbor s' is chosen from the neighborhood $\mathcal{N}(s)$ of the current solution s such that s' is not included in the tabu list. Subsequently, s is added to the tabu list. Hence, the set from which a neighbor s' of s is chosen depends on how s is reached, and no s' is chosen that has been visited in the k most recent steps, if the tabu tenure is k. By choosing solutions not on the trajectory to recently visited local minima, tabu search can be directed into other areas of the solution space.

Rather than storing complete solutions that have been visited in the tabu list, building elements associated with exchanges that have led to these solutions are included in the tabu list. If a step is made from a solution s to s' with $s' = \tau(s, e)$, where τ is an exchange function and e a building element of a solution, then an element e' for which $s = \tau(s', e')$ is added to the tabu list. If the tabu list is full, its oldest element is removed. Neighbors are obtained by performing exchanges that act on building elements that do not occur in the tabu list. An exchange is not permissible, or *tabu*, if it occurs on this list. In this way, the implicitly defined set of tabu solutions includes the k most recently visited solutions.

Most tabu search algorithms select the lowest-cost non-tabu neighbor in the neighborhood as the new current solution. If all neighbors are tabu, then the oldest element on the tabu list is removed, until a permissible neighbor can be selected. Since such a pivoting rule can be computationally expensive *candidate sets* can be used. Candidate sets are subsets of neighborhoods in which it is attempted to exclude high-cost neighbors from a neighborhood without exploring them explicitly. *Aspiration criteria* can be used to overrule the tabu status of an exchange when it seems promising in some sense, e.g., if it leads to a solution with lower cost than the best solution found so far.

The effectiveness of many tabu search algorithms is significantly increased by incorporating *intensification* and *diversification* of the search. Intensification of the search around good-quality local minima is used to search promising areas of the solution space more thoroughly. This can be implemented by keeping track of the best solutions found in the course of the search. If no improvement of the best solution is achieved for a number of steps, the search is restarted with one of these best solutions, where the search is forced to follow a different trajectory from this solution by associating a tabu status with each attempted exchange for this solution. This requires storage of tabu lists associated with the best solutions found in the course of the search.

Search diversification is used to ensure that other regions of the solution space are searched as well. Diversification of the search can be achieved by adding penalties to the cost of solutions if some of its elements occur frequently in the current solution or are included in the current solution for a large number of local search steps.

Simulated annealing [Kirkpatrick, Gelatt & Vecchi, 1983; Aarts & Korst, 1989] probabilistically accepts neighbors with higher costs, in addition to lower-cost neighbors, which are always accepted. To this end, an *acceptance probability* is used that is reversely related to the size of the cost increase. The acceptance probability of cost-increasing exchanges is given by

$$\exp(-\frac{(f(s') - f(s))}{c})$$

for solutions s and s' with $f(s') > f(s)$. The *control parameter* c is used to lower the acceptance probability in the course of the search.

A simulated annealing algorithm can be outlined as follows. First, an initial value for the control parameter c is chosen. For each value of c a number of randomly chosen exchanges is attempted, where proposed exchanges are accepted according to the above acceptance probability, after which the control parameter is lowered. If the control parameter c drops below a given value the search is ended. A *cooling schedule* determines how the control parameter c is decremented. In many practical simulated annealing algorithms a *geometric* cooling schedule is used in which c is decremented by multiplication with a constant factor α, with $\alpha < 1$.

Using finite Markov chain theory, it has been proven that simulated annealing converges to a global minimum if the acceptance probability is lowered slowly enough, provided that the employed neighborhood is at least sufficiently connected. This connectivity condition is not hard to meet for many problems.

Genetic local search [Mühlenbein, Gorges-Schleuter & Krämer, 1988; Aarts & Verhoeven, 1996] incorporates local search into genetic algorithms. In genetic local search, a *population* of solutions is explored, instead of only a single solution. A genetic local search algorithm starts with a population of p initial solutions. This population is augmented with q solutions obtained as follows. Two *parent* solutions are selected from the population. Using these two *parent* solutions a new solution is constructed by applying *cross-over*. In a cross-over operation building elements of two solutions are combined to one solution. This newly constructed solution is then improved using a local search algorithm and added to the population. The augmented population is subsequently reduced to its original size using concepts from biological evolution strategies.

Evidently, the cross-over operation is important, since here one must try to take advantage of the availability of more than one local minimum by exploiting their structure. *Mutation* in a traditional genetic algorithms corresponds with the part of the above algorithmic concept in which local search is applied to the solution resulting from the cross-over operation.

Widely recognized advantages of local search are its general applicability, its flexibility, and its ease of implementation. Only a specification of solutions, a cost function, and a neighborhood structure are required, which can easily be defined for many problems. Furthermore, practical experience in scheduling, layouting, and routing applications has revealed that the average-case performance of local search is quite good in the sense that high-quality solutions can be found within acceptable running times. This has led to the conviction that local search is a powerful technique to handle complex real-world combinatorial optimization problems that arise in management science and engineering. For more extensive overviews on local search in which the above variants are discussed in more detail we refer to [Yannakakis, 1990; Reeves, 1993; Aarts & Lenstra, 1996].

3

Concepts of Parallel Local Search

Although local search algorithms may find good-quality solutions, they require substantial amounts of running time for the larger instances of some combinatorial optimization problems; see for instance [Sechen, 1988], who reports running times of 24 hours for his simulated annealing based cell-placement algorithm when applied to real-life problem instances. To cope with this drawback, several approaches have been proposed in the literature to implement these algorithms on parallel machines. Other goals of these parallel approaches to local search are to find better quality solutions in a given amount of time or to enable handling of larger problem instances. In this chapter we try to disclose the underlying concepts on which these approaches are based. In this discussion we try to abstract from implementation issues that depend on the machine model at hand. Furthermore, we study some complexity issues of parallel local search.

Much effort in the field of parallel local search has concentrated on the design of parallel simulated annealing algorithms [Aarts & Korst, 1989; Azencott, 1992]. For an overview of parallel simulated annealing, we refer to [Greening, 1990]. More recently, also parallel approaches to other local search algorithms, such as tabu search and genetic local search, have been reported. Voss [1993] presents a classification of parallel tabu search algorithms based on the use of different search strategies and starting solutions. Crainic, Toulouse & Gendreau [1993b] extended this classification with dimensions based on communication organization and information handling.

We distinguish between *tailored* and *general* approaches. A tailored approach can be applied only to a specific problem, whereas a general approach can be applied to a wide variety of problems. Furthermore, we distinguish between single-walk and multiple-walk parallelism and between asynchronous and synchronous algorithms. In the class of single-walk algorithms, we distinguish between multiple-step and single-step parallelism.

Tailored approaches. A tailored approach to incorporate parallelism in local search requires parallel execution of a problem specific part of the algorithm. Cost computation is a typical problem dependent part of a local search algorithm. Also operations on the data structures used to represent a solution are problem specific. If one of these problem dependent parts is very time consuming, the running time of the local search algorithm can be decreased substantially by efficient parallel execution of this part.

Kravitz & Rutenbar [1987] apply a tailored approach to their local search algorithm for the standard cell placement problem. They compute the cost of a neighbor in parallel. Another example of parallel cost computation is given by Taillard [1994], who uses a parallel longest path algorithm to compute the length of a schedule in the job shop scheduling problem.

Tailored approaches have limited applicability because they strongly depend on the problem at hand. Therefore, we discard these approaches here and concentrate on techniques that can be applied to a wide variety of local search problems.

General approaches. A general approach can be applied to a broad class of local search problems. We can distinguish between two approaches: *single-walk* and *multiple-walk* parallel local search. In a single-walk algorithm only a single walk in the neighborhood graph is carried out, whereas in a multiple-walk algorithm several walks are performed simultaneously.

Within the class of multiple-walk algorithms we can distinguish between algorithms that perform interacting walks and algorithms that perform multiple independent walks. The latter can be considered as the most straightforward approach to introduce parallelism in local search.

In single-walk parallel local search a single walk is carried out in parallel. Typically, this requires some distribution of a solution or elements of a solution over a number of processors. Furthermore, we distinguish between *single-step* and *multiple-step* parallelism. The idea of single-step parallelism is to evaluate neighbors simultaneously and subsequently make a single step. To this end, the neighborhood of a solution is partitioned into subsets that are searched in parallel. In an algorithm with multiple-step parallelism several consecutive steps through the neighborhood graph are made simultaneously. Figure 3.1 shows neighborhood graphs that illustrate the idea of single-step and multiple-step parallelism.

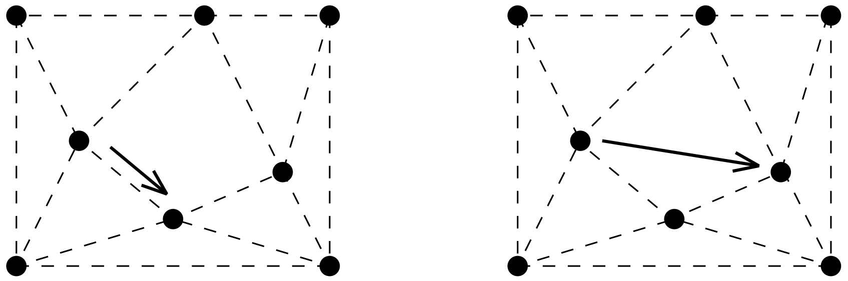

Figure 3.1: Local search with single-step parallelism (left) and multiple-step parallelism (right) in a neighborhood graph. Dotted edges connect neighbors and a solid arc represents a step made by a local search algorithm.

Finally, in both single-walk and multiple-walk parallel local search we distinguish between *synchronous* and *asynchronous* algorithms. In a synchronous algorithm one or more steps of the algorithm are performed simultaneously by all processors. Synchronous parallelism requires a global clocking scheme or token-passing mechanism that guarantees that communication occurs at given points in time. In asynchronous parallelism no such global clocking mechanism exists.

A multiple-walk algorithm performs multiple single walks. Hence, it is possible to combine multiple-walk and single-walk parallelism in the design of parallel local search algorithms. Also tailored approaches can be incorporated into general approaches resulting in all kinds of hybrid approaches, but the items in our classification are the basic concepts of such algorithms.

The remainder of this chapter is organized as follows. In Section 3.1 we discuss multiple-walk parallelism. Section 3.2.1 discusses single-step parallelism, and Section 3.2.2 discusses multiple-step parallelism in more detail. Finally, Section 3.3 discusses some complexity issues of parallel local search.

3.1 Multiple-walk parallelism

Multiple-walk algorithms have inherent parallelism, which can be exploited by distributing the walks over a number of processors and performing them simultaneously. We distinguish between algorithms that perform multiple independent walks and algorithms that perform multiple interacting walks. The first class of algorithms is the simplest one, since no complex parallelization scheme is required. In order to describe algorithms with multiple interacting walks more precisely, we extend the notion of neighborhood structures to hyper neighborhood structures. Neighborhood structures define neighbors for a single solution; hyper neighborhood structures define neighbors for a set of solutions [Vaessens, 1995].

```
proc Multiple_Walks ($s_1, \dots, s_P \in \mathcal{S}$)
  begin
    stop_criterion := false;
    while ¬ stop_criterion do
      par $p \in \Omega$ do
        Iterative_Improvement($s_p$) ;
        $s_p \in \mathcal{H}(s_{h_p(1)}, \dots, s_{h_p(M)})$
      rap
    od
  end
```

Figure 3.2: A multiple-walk parallel local search algorithm.

Definition 3.1. Let $M \geq 1$. Then, a *hyper neighborhood structure* $\mathcal{H} : \mathcal{S}^M \to \mathcal{P}(\mathcal{S})$ assigns to a set of M solutions in $\mathcal{S}$, a subset of solutions in $\mathcal{S}$. □

Let $\Omega = \{1, \dots, P\}$ be a set of P processors, and let $h_p : \{1, \dots, M\} \to \Omega$ be a function for each $p \in \Omega$, where $h_p(i)$ denotes the i-th processor with which p communicates. h_p gives, for each processor p, M other processors with which interaction takes place in order to construct a hyper neighbor.

Using the concept of hyper neighborhood structures, we can formulate the template presented in Figure 3.2 that describes algorithms with interacting multiple-walk parallelism. The set of current solutions $s_1, \dots, s_P$ is often called the *population*. A current solution s_p is assigned to processor p. All processors perform simultaneously a walk in the neighborhood graph by executing an iterative improvement algorithm, or some other local search algorithm. After a number of local search steps, a hyper neighbor is constructed, which requires interaction with M other processors $h_p(1), \dots, h_p(M)$. This process is continued until some stop criterion is met.

3.1.1 Multiple independent walks

A trivial approach to parallel local search is to perform a number of independent walks in parallel, since this requires no communication between different runs. The speed-up one hopes to achieve by using such an approach is based on the following property for random walks in a neighborhood graph.

Theorem 3.1. *Let $Q_p(t)$ be the probability of not having found a solution with relative excess ϵ in t time units with p independent walks, and let $Q_1(t) = e^{-\lambda t}$ with $\lambda \in \mathbb{R}^+$, i.e., Q_1 is distributed exponentially. Then, $Q_p(t) = Q_1(pt)$.* □

Hence, it is possible to achieve linear speed-up with multiple independent walks if the probability to find an optimal (or suboptimal) solution within a given amount of time units is distributed exponentially. An appropriate time measure for local search algorithms is the number of solutions of which the cost has been evaluated. Theorem 3.1 states that the probability to find a (sub)optimal solution in t time units and P processors is equal to the probability to find a (sub)optimal solution in Pt time with a single processor, if this probability is distributed exponentially. So in that case, a linear speed-up and an efficiency equal to one is reached with multiple independent walks.

Several authors have studied the probabilistic behavior of local search algorithms for various problems to investigate whether the condition of Theorem 3.1 holds in practice. Battiti & Tecchiolli [1992] empirically investigate the behavior of tabu search for randomly generated instances of the quadratic assignment problem. They observe that the probability of finding a (sub)optimal solution with a tabu search algorithm is indeed distributed exponentially, provided that the search is started with a local minimum. This indicates that good efficiencies can be obtained with multiple independent parallel walks in tabu search, if the time needed to find the first local minimum is relatively small compared to the time spent in the remainder of the search process. Taillard [1991] also shows that the probability of finding an optimal solution for quadratic assignment problems with a tabu search algorithm fits well with an exponential distribution. Dodd [1990] empirically shows that a similar conclusion also holds for simulated annealing. For the problem and parameter setting that he used an efficiency of one could be obtained with a maximum of 16 processors. Osborne & Gillett [1991] investigate the empirical behavior of simulated annealing for the Steiner tree problem in graphs. They also observe that the probability of finding a near-optimal solution is distributed exponentially. Chapter 4 presents some results for the average-case behavior of iterated local search algorithms for the traveling salesman problem. It is shown that the probability of finding a solution with a small excess over the global minimum using an iterated local search algorithm can be described by a geometrical distribution, which is the discrete counterpart of an exponential distribution, somewhat translated to compensate for the time needed to find the first local minimum. Good efficiencies can be obtained with multiple-walk parallelism but the amount of speed-up depends on the time needed to find the first local minimum, the time needed to find subsequent local minima, and the desired solution quality.

Shonkwiler & Van Vleck [1994] present a theoretical analysis of the speed-up with multiple independent stochastic walks using Markov-chain analysis. They show that the speed-up is given by $\frac{1-\lambda^p}{1-\lambda}s^{p-1}$ where p is the number of walks and s and λ are parameters that are related to the search process at hand. They claim that nearly always $\lambda \approx 1$ so then the speed-up is equal to ps^{p-1}. They present some artificial problems for which they are able to determine the parameters s and λ, and for some of these problems even superlinear speed-ups are achieved with multiple independent stochastic walks. Ferreira & Zerovnik [1993] give another theoretical result which shows that multiple independent walks can be profitable. They show that, after a certain amount of time, the probability to find a (sub)optimal solution with multiple runs of an iterative improvement algorithm is larger than the probability that it is found by a single simulated annealing run in the same amount of time.

The inherent parallelism of genetic algorithms is also based on Theorem 3.1, although in genetic algorithms the walks interact to combine good parts of individual solutions. The population size is therefore an important parameter of a parallel genetic algorithm, because it determines the speed-up and efficiency that can be obtained ultimately.

3.1.2 Multiple interacting walks

Interaction of walks can be used to faster direct the search into promising areas of the solution space. To model the interaction between parallel walks we use the concept of hyper neighborhoods. Hyper neighborhoods can be based on selection of a single solution from a set of solutions. Such hyper neighborhoods assign the same solution to different processors and are useful only if the walks performed by processors follow different paths from this solution. This can be achieved by either introducing randomization in walks or by enforcing different directions for walks. Other hyper neighborhoods are based on combining parts of different solutions to new solutions.

Processors exchange information when walks interact. This information, such as the occurrence of low-cost solutions in a population or the availability of multiple good parts of individual solutions, can be used to speed up the search. A possible way to use the additional information resulting from interaction of walks, is indicated by Lin & Kernighan [1973] and Kirkpatrick & Toulouse [1985], who observe that local minima often have many elements in common. These elements can be identified as the good parts of solutions. So the availability of multiple local minima allows fast identification of good parts of individual solutions that should not be changed in the course of the search. This then leads to a reduction of the sizes of neighborhoods.

Implementation issues. First, we discuss asynchronous and synchronous multiple walks. In synchronous multiple-walk parallelism, walks are synchronized at certain points in time—that is, the construction of a hyper neighbor can only be done in a state that depends on the state of all processors. In asynchronous multiple-walk parallelism construction or selection of a hyper neighbor is not synchronized at a point in time. Here, the number of processors with which a processor interacts to construct a hyper neighbor is typically a subset of all processors. In that case a processor can resume searching as soon as communication with that subset of processors has taken place, and it does not need to wait for other processors.

The amount of communication overhead involved with multiple-walk parallelism depends on how often a hyper neighbor is constructed, the hyper neighborhood structure at hand, the machine architecture, and on the mapping of walks on to processors. It is preferable to map interacting walks on to the individual processors of a given machine such that the resulting communication overhead is minimal. Load balancing is done by specifying an equal amount of work that is to be performed in between two interactions of walks. This can be done by fixing the number of steps between two communications, which typically means that hyper neighbors are not always constructed from local minima, since the number of steps to find a local minimum is not equal in each walk. An advantage of asynchronous walks over synchronous walks is their reduced load imbalance.

Examples. Parallel genetic local search algorithms are typical examples of local search algorithms with multiple-walk parallelism. The following hyper neighborhood is used in genetic algorithms. Two parent solutions, selected from —a subset of— the current population, are used in a cross-over operation to construct a new solution. In a cross-over operation good parts of individual solutions are combined. Both synchronous and asynchronous multiple-walk parallelism can be applied in genetic algorithms. Asynchronous parallel genetic algorithms are mostly based on an *island model* in which parents are chosen from a subset of the population instead of from the entire population. Here, we do not further elaborate on parallel genetic algorithms; for more details we refer the reader to [Jog, Suh & Van Gucht, 1991; Michalewicz, 1992; Mühlenbein, 1992].

Several authors have applied multiple-walk parallelism to simulated annealing and tabu search. A hyper neighborhood of a population of solutions that is often used here is the selection of the best solution in the population. In a tabu search algorithm different paths from this solution should be followed by different processors. This can by done by blocking certain exchanges or by using different search parameters for different processors. Note that in this way an intensification of the search around good solutions is achieved. Additional diversification can be achieved by combining long-term memory of different walks.

Malek, Guruswamy & Pandya [1989] present tabu search and simulated annealing algorithms with synchronous multiple-walk parallelism in which the best solution in the population is communicated to all processors after a fixed number of steps of the simulated annealing or tabu search algorithm. In their algorithm the hyper neighborhood of a solution consists of a single element, viz., the best solution come across in the entire population. Different parameter settings are used to guarantee that different paths are followed by processors. They obtain a good speed-up for seven processors.

Crainic, Toulouse & Gendreau [1995] present various synchronous multiple-walk tabu search algorithms. They propose two hyper neighborhoods: one in which the best solution in the entire population is selected by all processors, and one in which each of the P processors selects a different solution among the P best solutions come across since the previous interaction. Different paths are followed by choosing a different parameter setting for processors. Typically, the best-quality solutions are found by multiple independent walks. Crainic, Toulouse & Gendreau [1993a] also present asynchronous multiple-walk parallel tabu search algorithms. They propose two hyper neighborhoods in which processors interact asynchronously with a central processor that keeps track of either the best solution or the P best solutions found in the search process, where P is the number of processors. In the first hyper neighborhood this overall best solution is selected as hyper neighbor, and in the latter hyper neighborhood a hyper neighbor is randomly chosen from the P best solutions found. Compared to their synchronous algorithms they obtain slightly better final solutions. Although actual running times of the various synchronous and asynchronous algorithms are not presented, the number of iterations after which the overall best solution is found implies that the algorithms display little speed-up.

Aarts, De Bont, Habers & Van Laarhoven [1986] present parallel simulated annealing algorithms based on a combination of multiple interacting walks and single-step parallelism in which the size of the population is gradually decreased in the course of the search. They propose the following two hyper neighborhoods. In the first one the best solution from the entire population is chosen ($M = P$), and in the second one a solution is randomly chosen from two solutions ($M = 2$). The first hyper neighborhood results in synchronous multiple-walk parallelism, the second one in asynchronous multiple-walk parallelism. Similar algorithms using these hyper neighborhoods are implemented by Diekmann, Lüling & Simon [1993], who obtain a speed-up of 85 on a network of 120 transputers for the traveling salesman problem.

Moscato [1993] and Fox [1993] present parallel algorithms that combine genetic algorithms, simulated annealing, and tabu search. They claim a linear speed-up, but no empirical evidence is given. Mahfoud & Goldberg [1995] propose a combination of genetic algorithms and simulated annealing, but so far they do not present results for combinatorial optimization problems.

3.2 Single-walk parallelism

In this section we discuss a template that captures local search algorithms with single-walk parallelism. In an algorithm with single-walk parallelism one or more consecutive steps in the neighborhood graph are made in parallel. To this end, applications of the exchange function, which we call *exchanges*, are proposed that all act on the same solution. Since effectuation of a proposed exchange can prohibit effectuation of other exchanges, the proposed exchanges are combined in such a way that a feasible solution is constructed, which can imply that not all proposed exchanges can be effectuated. Let τ be an exchange function as defined in Definition 2.2. Then a single-walk parallel local search algorithm consists of the following steps.

(1) Partition the domain $\{(s, e_1, \ldots, e_r) \mid e_1, \ldots, e_r \in s\}$ of the exchange function τ for a given current solution $s \in \mathcal{S}$.

(2) Propose exchanges for each subdomain simultaneously.

(3) Effectuate —a subset of— the profitable exchanges found in step (2), which results in a new solution s'.

(4) Replace s by s', and continue steps (1) to (3) until some stopping criterion is met.

Single-step parallel local search, which is based on parallel neighborhood exploration, is a special case of the above concept. In single-step parallelism each processor examines a part of the neighborhood of the current solution s, and only a single exchange is effectuated in step (3). So speed-up is achieved only in step (2). Single-step parallelism implies that, in terms of a walk in the neighborhood graph, only a single step is made as solution s' is obtained from s by applying a single exchange. In an algorithm with multiple-step parallelism several exchanges are effectuated in step (3), which implies that a solution s' is constructed that can only be reached from s by performing multiple steps in the neighborhood graph—that is, multiple exchanges have to be performed to obtain s' from s; see Figure 3.1.

Below, we present a machine-independent template for single-walk parallel local search algorithms. First we introduce the concept of distributed neighborhood structures, which have to deal with the following issues. A distributed neighborhood has to specify how solutions and domains of the exchange function are decomposed. Furthermore, it has to specify how proposed exchanges are combined. These aspects are captured in the following definitions.

Definition 3.2. Let $\Omega = \{1, \ldots, P\}$ be a set of processors, $\mathcal{S}$ the set of solutions, and $\mathcal{U} = \bigcup_{s \in \mathcal{S}} \mathcal{P}(s)$ the set of *partial solutions*. A *solution distribution* δ gives a partial solution δ_p for each processor $p \in \Omega$. A *local neighborhood structure* $\mathcal{L} : \mathcal{U} \mapsto \mathcal{P}(\mathcal{U})$ gives sets of partial solutions for partial solutions. □

Similar to conventional neighborhoods, local neighborhoods are not given explicitly but are constructed using an exchange function. The following definition states how this is done.

Definition 3.3. Let $\tau : \mathcal{U} \times \mathcal{E}^r \mapsto \mathcal{U}$ be an exchange function that can be applied to partial solutions. A *domain distribution* $\lambda_p : \mathcal{U} \mapsto \mathcal{P}(\mathcal{E}^r)$ gives for each processor $p \in \Omega$ and partial solution $u \in U$ a set of arguments, chosen from u, for the exchange function τ. A *local neighborhood* $\mathcal{L}(u)$ is equal to $\{\tau(u, e_1, \ldots, e_r) \mid (e_1, \ldots, e_r) \in \lambda_p(u)\}$ for $u \in \mathcal{U}$. □

The domain distribution describes which elements of partial solutions are the arguments used by the exchange function to construct local neighbors. If solutions are decomposed, exchanges involving elements of different partial solutions cannot occur, and therefore several decompositions need to be examined. This issue is also described in the following definition of a distributed neighborhood.

Definition 3.4. Let $\mathcal{S}$ be a set of solutions, $\mathcal{U}$ the set of partial solutions, Ω a set of P processors, and τ an exchange function. Then, a *distributed neighborhood structure* $\mathcal{D}$ is a triple (Δ, λ, ϕ) defined as follows. A *distribution structure* Δ gives for each solution s and set of processors Ω a set of solution distributions, with $\Delta(s, \Omega) \subseteq \{\delta \in \Omega \to \mathcal{U} \mid \bigcup_{p \in \Omega} \delta_p = s\}$. λ gives for each $p \in \Omega$ a domain distribution $\lambda_p : \mathcal{U} \mapsto \mathcal{P}(\mathcal{E}^r)$ with $r \in \mathbb{N}^+$, where λ gives the arguments of the exchange function τ used to construct the local neighborhood $\mathcal{L}(\delta_p)$ of a partial solution δ_p. A *combination function* $\phi : \mathcal{U}^P \mapsto \mathcal{S}$ combines P partial solutions to a feasible solution. □

A distributed neighborhood structure consists of a distribution structure that specifies how solutions are distributed over processors, a domain distribution that defines local neighborhoods, and a combination function that specifies how proposed exchanges have to be combined. Note that only a subset of the proposed exchanges might be effectuated by the combination function, since effectuating all proposed exchanges might lead to infeasible solutions. Next, we formalize the notion of local optimality for a distributed neighborhood structure.

Definition 3.5. Given are an instance $(\mathcal{S}, f)$, with $f : \mathcal{U} \mapsto \mathbb{Z}$, a distributed neighborhood structure $\mathcal{D} = (\Delta, \lambda, \phi)$, and processor set Ω. Then, a solution $s \in \mathcal{S}$ is a *local minimum* of $\mathcal{D}$ if for each solution distribution $\delta \in \Delta(s, \Omega)$, processor $p \in \Omega$, and partial neighbor $\delta'_p \in \mathcal{L}(\delta_p)$ holds $f(\delta'_p) \geq f(\delta_p)$. □

Using the concept of distributed neighborhood structures we can formulate the template for single-walk parallel local search given in Figure 3.3. In this algorithm first a distribution of the current solution is chosen. Then, all processors

```
proc Single_Walk $(s : \mathcal{S})$
    var $\delta : \Omega \to \mathcal{U}$; $D : \mathcal{P}(\Omega \to \mathcal{U})$;
    begin
        $D := \emptyset$;
        while $\Delta(s, \Omega) \setminus D \neq \emptyset$ do
            $\delta \in \Delta(s, \Omega) \setminus D$;
            par $p \in \Omega$ do Iterative_Improvement $(\delta_p)$ rap;
            if gain found then $D := \emptyset$; $s := \phi(\delta_1, \ldots, \delta_P)$
            else $D := D \cup \{\delta\}$ fi
        od $\{s$ is a local minimum of $\mathcal{D}\}$
    end
```

Figure 3.3: A single-walk parallel local search algorithm.

simultaneously propose exchanges. Subsequently, a subset of the proposed exchanges is effectuated to obtain a new feasible solution. This process is repeated until no solution distribution of the current solution can be improved upon, at which point a local minimum has been found.

To guarantee termination of the algorithm, ϕ should combine a set of neighbors of s in such a way that, if any neighbor has lower cost than s, their combination also has lower cost than s. We define this termination condition as follows.

Definition 3.6. Given are a problem instance $(\mathcal{S}, f)$, a distributed neighborhood structure $\mathcal{D} = (\Delta, \lambda, \phi)$, and a set of processors Ω. Then, the combination function ϕ is called *progressive*, if for all $\delta \in \Delta(s, \Omega)$, $p \in \Omega$, and $s \in \mathcal{S}$ holds that $\exists_{\delta'_p \in \mathcal{L}(\delta_p)} f(\delta'_p) < f(\delta_p) \Rightarrow f(\phi(\delta'_1, \ldots, \delta'_P)) < f(s)$. □

An important issue is raised by the question how we can compare a distributed neighborhood structure $\mathcal{D}$ with a conventional neighborhood structure $\mathcal{N}$. This enables a comparison of the solutions found by a parallel local search algorithm with those found by a sequential local search algorithm. The following definition is useful for such a comparison.

Definition 3.7. Given are a problem instance $(\mathcal{S}, f)$, a neighborhood structure $\mathcal{N}$, and a distributed neighborhood structure $\mathcal{D}$ for this problem. Then, $\mathcal{D}$ is called *isomorphic* with $\mathcal{N}$ if, for any set of processors Ω, holds that each local minimum of $\mathcal{N}$ is a local minimum of $\mathcal{D}$, and vice versa. □

If a distributed neighborhood structure $\mathcal{D}$ is isomorphic with a neighborhood structure $\mathcal{N}$, then the expected average cost of final solutions found by sequential and parallel algorithms, are equal, provided that sequential and parallel algorithms have equal probabilities for finding given local minima.

Implementation issues. First, we discuss synchronous and asynchronous single-walk parallel algorithms. In an algorithm with synchronous single-walk parallelism some step of the algorithm can be performed only in a state dependent on the state of all processors. This can occur for example when global communication has to take place before a next step of the algorithm can be performed. Synchronization is typically required in a decentralized algorithm in which each processor has a copy of the current solution. In order to update all local copies of the current solution after some exchanges have been proposed, global communication is required. If there is a central control mechanism that manages access to the current solution, no inconsistencies in the current solution can occur. So processors can then proceed independently from each other, although it may occur that some of the proposed exchanges cannot be effectuated because the current solution has already been altered by another processor. The choice between synchronous or asynchronous parallelism is often determined by characteristics of the parallel machine at hand, such as the availability of common memory or the configurability of a machine in a master/slave model. As we want to abstract from machine-dependent issues, we do not further elaborate on this subject.

The speed-up obtained with asynchronous and synchronous single-walk parallelism depends on problem characteristics and on the architecture of the target machine. Two important issues to obtain a good speed-up are load balancing and communication overhead, which we briefly address below.

In many single-walk parallel local search algorithms global communication is necessary because effectuation of an exchange has a global impact on the characteristics of solution. If a machine allows direct communication with a central processor, or if it has common memory from which all processors can read efficiently, then the overhead of asynchronous global communication is limited. In many other cases, however, global communication between all processors causes a substantial communication overhead.

Load balancing is needed to reduce the amount of time processors are idle. In synchronous parallelism load balancing must be done by assigning equal amounts of work to all processors in between synchronization points. In asynchronous single-walk parallelism load balancing is needed only for processors that interact.

Finally, the efficiency of single-walk parallelism is also determined by the ratio between the time needed to evaluate the cost of a neighbor and to combinate and effectuate proposed exchanges. For this reason the data structures used to represent a solution are important, since these data structures determine the efficiency of operations that have to be performed to effectuate proposed exchanges.

3.2.1 Single-step parallelism

In the previous section, we have introduced a template to capture single-walk parallel local search algorithms. In order to apply single-step parallelism to a given problem, the distributed neighborhood of Definition 3.4 is instantiated as follows. The distribution structure is chosen as $\Delta(s, \Omega) = \{\delta : \Omega \rightarrow \mathcal{S} \mid \delta_p = s$ for all $p \in \Omega\}$ and the combination function is given by $\phi(s_1, \ldots, s_P) \in \{s_i \mid 1 \leq i \leq P\}$. Moreover, it is not difficult to define a domain distribution that partitions a conventional neighborhood, which guarantees that the distributed neighborhood is isomorphic with it. The distributed neighborhood of Definition 3.4 is now straightforward, since a complete solution is assigned to each processor, and the combination function effectuates one of the proposed exchanges. This means that the combination function only has to select a neighbor from the proposed neighbors.

An advantage of single-step parallelism is its general applicability. Moreover, it results in a distributed neighborhood structure that is isomorphic with a conventional neighborhood structure that uses the same exchange function. So, under mild conditions, it is guaranteed that the parallel algorithm finds the same quality solutions as a sequential algorithm.

A disadvantage of single-step parallelism is the amount of speed-up that can be obtained. A local search algorithm with single-step parallelism explores neighbors of a solution simultaneously. Subsequently, one of the neighbors replaces the current solution, which results in a single step in the neighborhood graph. The amount of speed-up that can be obtained by single-step parallelism strongly depends on the number of exchanges that has to be examined before a proposed exchange is accepted. If best improvement is used as pivoting rule, then single-step parallelism gives good speed-up, because finding the neighbor with lowest cost requires scanning of the entire neighborhood of a solution. Best improvement is often used in tabu search, so single-step parallelism can be effectively applied in tabu search algorithms. However, best improvement is not always the most effective pivoting rule since for many problems the same quality solutions can be found with first improvement in smaller running times.

For algorithms with first improvement the amount of speed-up that can be obtained by single-step parallelism depends on the stage of the search process that has been reached by the algorithm. If the current solution has few lower-cost neighbors, a situation arises that is almost similar to best improvement, since then it is more likely that a large proportion of neighbors has to be examined before a solution with lower cost is found. If the proportion of neighbors that would be accepted by a local search algorithm at a certain stage of the search process is given by α, then the maximum speed-up with single-step parallelism is $\frac{1}{\alpha}$, regardless of the number of processors; see also [Roussel-Ragot & Dreyfus, 1990; Azencott, 1992] for a theoretical analysis of the speed-up that can be obtained by single-step parallelism in simulated annealing. Generally, α is large in the beginning of the search process, and consequently the resulting speed-up is small at first. As the cost of the current solution decreases, α increases and thus the resulting speed-up. So for a local search algorithm with first improvement, single-step parallelism can typically be applied efficiently only in a later stage of a local search walk when low-cost solutions are reached.

Examples. One the first examples of single-step parallelism in the literature is the algorithm of Kravitz & Rutenbar [1987] who apply asynchronous single-step parallelism to simulated annealing for the cell placement problem. They report a speed-up of 2.5 on a shared memory machine with four processors. Diekmann, Lüling & Simon [1993] present asynchronous and synchronous single-step parallel simulated annealing algorithms for the traveling salesman problem and the link assignment problem. Their algorithms are implemented on a network of 120 transputers. They obtain a speed-up of 35 for the traveling salesman problem and a speed-up of 70 for the link assignment problem. Kindervater, Lenstra & Savelsbergh [1993] present a local search algorithm with synchronous single-step parallelism for the time-constrained traveling salesman problem. Verification of local optimality can be done in $\mathcal{O}(\log N)$ time with $\mathcal{O}(N^2/\log N)$ processors on a PRAM machine, but no empirical results on a parallel machine are given. Synchronous single-step parallelism is used by Ravikumar [1992] in an iterative best improvement algorithm for the traveling salesman problem.

As already mentioned, many tabu search algorithms also use best improvement as pivoting rule. Synchronous single-step parallelism can then be applied effectively, as is shown by Taillard [1990, 1991] for the flow shop sequencing problem and the quadratic assignment problem. Chakrapani & Skorin-Kapov [1993a, 1993b] use synchronous single-step parallelism in combination with best improvement for the traveling salesman problem and the quadratic assignment problem, respectively. Their algorithm is implemented on a massively parallel Connection Machine with 16,384 processors, but running times are compared to an implementation on a workstation only. They also report that 55 percent of the running time is spent for communication. Li & Pardalos [1992] use synchronous single-step parallelism in a parallel variable depth algorithm for the quadratic assignment problem.

3.2.2 Multiple-step parallelism

Essential to achieve a good speed-up with multiple-step parallelism is that a large proportion of the proposed exchanges can be effectuated in the combination step.

The applicability of multiple-step parallelism therefore strongly depends on the problem at hand, as the problem must allow effectuation of several proposed exchanges for a given solution.

Two approaches to design distributed neighborhood structures can be distinguished based on whether or not solutions are decomposed into partial solutions. If the cost difference between two neighboring solutions can be computed using only information on the removed and inserted parts of a solution, then it might be profitable to decompose a solution and assign each partial solution to a processor. This enables an efficient implementation of the combination function because a new solution can then be constructed without global communication. A disadvantage, however, is that exchanges that involve elements of different partial solutions cannot be proposed. In order to find the same quality solutions as a sequential local search algorithm, different distributions have to be examined, which might lead to a complex distribution structure.

Another approach in the design of distributed neighborhood structures is to provide each processor with a copy of the current solution and to define an appropriate domain distribution. This often leads to a distributed neighborhood that is isomorphic with a conventional neighborhood that uses the same exchange function. To achieve speed-up, however, an effective combination function is essential because effectuation of an exchange can prohibit effectuation of other exchanges to guarantee feasibility of solutions. Furthermore, global communication is often required in the combination function, which can lead to a large communication overhead depending on the target machine's architecture.

Examples. The traveling salesman problem of Example 2.1 is a typical example of a problem for which decomposition of a solution is often used. Here a solution is a Hamiltonian cycle, and the cost difference between neighbors can be computed using only the lengths of removed and inserted edges. Moreover, it is possible to combine proposed exchanges without extensive communication. Most of the examples we discuss below are implemented on a distributed-memory MIMD machine and use synchronous parallelism.

One of the first applications of multiple-step parallelism to the TSP is presented by Felten, Karlin & Otto [1985], who divide a tour in consecutive paths. Each path is assigned to a different processor. A new distribution is obtained by assigning adjacent edges that are assigned to different processors to the same processor. Fiechter [1994] presents a parallel tabu search algorithm for the TSP that uses a similar solution distribution. Partial solutions that are assigned to processors consist of one path in the tour. New distributions of a tour are obtained by assigning different paths to processors.

Allwright & Carpenter [1989] present a parallel simulated annealing algorithm for the TSP based on a distribution of a tour over a linear array of processors. A partial solution in this distribution consists of two non-adjacent paths in the tour. Edges are randomly reassigned to processors to obtain a new distribution. In Chapter 5 we present distributed neighborhood structures for the TSP based on a similar solution distribution. We prove that our distributed neighborhoods are isomorphic with the well-known 2-exchange and 3-exchange neighborhoods. Here, a new distribution is obtained by assigning a single edge to a new processor, which leads to synchronization between processors. In Chapter 5 we also present a parallel Lin-Kernighan algorithm based on domain decomposition. Each processor works on a copy of the entire tour, and the proposed exchanges are subsequently combined to a new tour, where as many as possible exchanges are effectuated in the combination function.

Bachem, Steckemetz & Wottawa [1994] propose a solution distribution based on a geometric partition of the cities in a TSP instance. The partial solutions, which consist of Hamiltonian paths, are combined by a tour construction heuristic. A new distribution is obtained by choosing a new geometric partition.

Applications of multiple-step parallelism can also be found in the placement of cells in circuit layouting. Shahookar & Mazunder [1991] present an overview of this field. In most of the examples simulated annealing is used. For cell placement problems a local change might have a global impact on a solution, since the cost of a solution is determined by the relative position of pairs of cells. If no restrictions are imposed, effectuation of proposed exchanges can lead to erroneous cost computation. Erroneous cost computation changes the convergence behavior of simulated annealing, but various empirical results in the papers below show that simulated annealing for cell placement is rather robust to errors in the cost evaluation. Romeo & Sangiovanni-Vincentelli [1991] present some theoretical results on erroneous cost evaluation in simulated annealing.

Darema, Kirkpatrick & Norton [1987] use a solution distribution in which they assign the cells to processors, so each partial solution consists of a subset of the set of cells, and all proposed exchanges are accepted by the combination function. A new distribution is obtained by randomly reassigning different cells to each processor. They compare two solution distributions: one that guarantees correct cost computations and one that allows errors in the cost computation. A drawback of the first approach is that only a limited number of processors can be used, which prohibits scaling of the algorithm. The second approach does not suffer from this drawback. Their empirical results show that both approaches find the same quality final solutions. Jones & Banerjee [1987] present a similar algorithm in which they also decompose a solution. A new decomposition is obtained by exchanging cells between neighboring processors, and the combination function accepts all proposed exchanges, which may lead to erroneous cost computations. The algorithm is implemented on a distributed MIMD machine. Casotto, Romeo & Sangiovanni-Vincentelli [1987] use a local search heuristic to find a new decomposition of the solution that minimizes the probability of erroneous cost computation. These errors are caused by accepting all proposed exchanges without checking the position of cells in the newly constructed solution. Their algorithm is implemented on a shared-memory parallel machine. A hybrid algorithm that uses both multiple-walk and multiple-step parallelism is presented by Rose, Snelgrove & Vranesic [1988]. They apply multiple-walk parallelism at high temperatures of the simulated annealing algorithm and multiple-step parallelism during low temperature annealing, because at low temperatures erroneous cost computation is less likely to occur due to the low acceptance rate.

Barbosa & Gafni [1989] present a simulated annealing algorithm with multiple-step parallelism that can be applied to problems for which only a single element is replaced in the exchange function of the neighborhood structure. Moreover, such an exchange must have limited interaction with other elements of a solution. They propose a solution decomposition approach that guarantees that no erroneous cost computation can occur when all proposed exchanges are effectuated, at the cost of a limited number of employable processors. As a consequence, the maximum speed-up that can be obtained with their approach depends on characteristics of the problem instance at hand.

Another example of multiple-step parallelism is the algorithm of Savage & Wloka [1991] for the graph partitioning problem, for which they propose a solution distribution based on decomposing a solution. The combination function either accepts all proposed exchanges if the resulting global solution has lower cost, or it rejects all proposed exchanges if the resulting global solution has higher cost. Such a combination function has low computational overhead but it also may effectuate individually deteriorating exchanges.

Boissin & Lutton [1993] present a framework for parallel simulated annealing based on multiple-step parallelism. Exchanges are proposed according to the conventional acceptance criterion of simulated annealing, and the combination function probabilistically accepts or rejects all proposed exchanges—that is, it accepts or rejects the global solution resulting from these exchanges. The algorithm is tested on the quadratic assignment problem and on a 0-1 quadratic function minimization problem, displaying moderate speed-ups when implemented on a connection machine and compared to implementations on a workstation.

Taillard [1993] and Garcia, Potvin & Rousseau [1994] propose parallel tabu search algorithms for vehicle routing. Taillard proposes a solution distribution in which entire, geographically close, routes are preferably assigned to processors that are directly connected in the processor network on which the algorithm is implemented. A new distribution is obtained by assigning elements to neighboring processors. Garcia, Potvin & Rousseau propose a domain distribution for the vehicle routing problem with time windows. Their combination function consecutively effectuates the proposed exchanges in order of descending gain.

3.3 Complexity issues of parallel local search

In this section we discuss some issues related to the parallel complexity of finding local optima. For an overview of the complexity theory of parallel computations, the reader is referred to [Greenlaw, Hoover & Ruzzo, 1995].

If PLS-complete problems require superpolynomial running times, as is conjectured by Yannakakis [1996], then parallel algorithms that find local minima using a polynomially bounded number of processors, which is the case in any realistic machine model, will also run in superpolynomial time. Therefore, we restrict ourselves to the parallel complexity of the *verification problem*, the problem of deciding whether a solution is a local optimum. Consider a problem instance $(\mathcal{S}, f)$ and a neighborhood structure $\mathcal{N}$, then we have the following result.

Theorem 3.2. *Let* $|\mathcal{N}(s)| = b(n) > 1$ *for all* $s \in \mathcal{S}$, *where* n *denotes the size of a solution, and let* $c(n)$ *be the time complexity of deciding whether* $f(s') \geq f(s)$ *for a given solution* $s' \in \mathcal{N}(s)$. *Then, a parallel algorithm for the verification problem exists that uses* $b(n)/\log b(n)$ *processors on a PRAM and that has a time complexity of* $\mathcal{O}(c(n)\log b(n))$ *and an efficiency of* $\mathcal{O}(1)$.

Proof. The sequential time complexity for this problem is $\mathcal{O}(c(n)b(n))$. Partition the neighborhood in $b(n)/\log b(n)$ sets that contain $\log b(n)$ solutions. Use an algorithm with synchronous single-step parallelism. Its time complexity is $\mathcal{O}(c(n)\log b(n)) + \mathcal{O}(\log(b(n)/\log b(n)))$. □

In Chapter 1 we have discussed the complexity class EP introduced by Kruskal, Rudolph & Snir [1990] that contains problems for which an efficient parallel algorithm on a PRAM machine exists. Recall that a problem is in EP if there exists a parallel algorithm with speed-up that scales with the instance size and for which the computational effort, the time complexity of the parallel algorithm multiplied by the number of processors, is equal to the time complexity of the sequential algorithm. Theorem 3.2 now leads to the following corollary.

Corollary 3.1. *If the complexity of the problem to decide whether* $f(s') \geq f(s)$ *for* $s \in \mathcal{S}$ *and* $s' \in \mathcal{N}(s)$ *is polynomially bounded in the instance size and if neighborhood sizes are also bounded polynomially, then the verification problem is in the class EP. Moreover, if the former problem is in NC, then the verification problem for* $\mathcal{N}$ *is in NC.*

Proof. Let $c(n) = \mathcal{O}(n^u)$ be the complexity of the problem to decide whether $f(s') \geq f(s)$ for $s \in \mathcal{S}$ and $s' \in \mathcal{N}(s)$, and let $b(n) = \mathcal{O}(n^v)$ be the neighborhood size for instances of size n. Let T_p be the time required by an algorithm for the verification problem with p processors. Then, $T_1 = \mathcal{O}(b(n)c(n)) = \mathcal{O}(n^{u+v})$ and according to Theorem 3.2 for $p = \frac{b(n)}{\log b(n)}$ holds $T_p = \mathcal{O}(c(n) \log b(n)) = \mathcal{O}(n^u \log n^v)$. It remains to show that $T_p = \mathcal{O}(T_1^\epsilon)$ for $\epsilon < 1$. It holds that $T_p = \mathcal{O}(n^u \log n) = \mathcal{O}(n^u n^\delta)$ for $\delta > 0$. For any δ and ϵ such that $0 < \delta < v$ and $\frac{u+\delta}{u+v} \leq \epsilon < 1$ holds $T_p = \mathcal{O}(n^{u+\delta}) = \mathcal{O}(n^{(u+v)\epsilon}) = \mathcal{O}(T_1^\epsilon)$. Furthermore, $T_p \cdot p = \mathcal{O}(T_1)$. Thus, the verification problem is in EP. Moreover, if $c(n) = (\log n)^{\mathcal{O}(1)}$ on a PRAM, then the verification problem is NC. □

Corollary 3.1 shows that verification of local optimality can be done in polylogarithmic time with a polynomially bounded number of processors, if the cost evaluation of a neighbor can be done in polylogarithmic time. This typically holds when the cost evaluation of a neighbor is $\mathcal{O}(1)$, for example in the 2-exchange neighborhood for the TSP. Corollary 3.1 shows that it is possible to design an efficient parallel algorithm for the verification problem with speed-up that is proportional to the size of the problem. The running time of this parallel algorithm can still be polynomial, but in practical situations one is satisfied with parallel algorithms with a good speed-up that scales with the instance size.

Another important topic is the complexity of finding a local optimum of a distributed neighborhood structure. The notion of PLS-completeness and the definition of the class PLS can also be applied to the problem of finding a local optimum of a distributed neighborhood structure. Using this extension, we obtain the following result.

Theorem 3.3. *Let a distributed neighborhood structure $\mathcal{D}$ be isomorphic with a neighborhood $\mathcal{N}$. If the problem of finding a local optimum of $\mathcal{N}$ is PLS-complete, then the problem of finding a local optimum of $\mathcal{D}$ is PLS-complete.* □

Finally, we discuss upper bounds on the speed-up that can be obtained with the various approaches discussed in this chapter. We assume that we have an infinite number of processors at our disposal. The number of processors that can be employed in multiple-walk parallelism is not bounded by a polynomial in the size of the instance, so here we do not have a polynomial upper bound on the maximum speed-up. In an algorithm with single-walk parallelism the maximum speed-up is at most the size of neighborhoods because at least one neighbor has to be evaluated by each processor that contributes to the speed-up. Most neighborhoods have low-order polynomial sizes. In practice the number of processors of a parallel machine is typically much smaller than the neighborhood size, so the number of available processors is then a trivial upper bound on the maximum speed-up.

4

Multiple Independent Walks

The most straightforward approach to introduce parallelism in local search is to perform multiple independent runs of a local search algorithm simultaneously since this requires no complex parallelization scheme or even dedicated parallel hardware. An important question is therefore to what extend such an approach can be successful. In this chapter we analyze the speed-up that can be obtained by performing several independent walks in the neighborhood graph simultaneously. In this analysis we concentrate on iterated local search for the traveling salesman problem but the conclusions drawn from it can be applied to other problems and local search variants as well. The outline of this chapter is as follows. Section 4.1 presents a theoretical and empirical average-case analysis of a 2-opt algorithm for the TSP. Section 4.2 gives a semi-empirical analysis of the average-case performance of an iterated 2-opt and Lin–Kernighan algorithm. The main results of this chapter are discussed in Section 4.3 in which we show how these results can be used to analyze the speed-up achieved by multiple independent walks.

The performance of local search algorithms can be quantified by the relative excess of the obtained final solutions and the required running time. Empirical results show that local search can find good-quality solutions within low-order polynomial running times. It is, however, conjectured that worst-case running times cannot be bounded polynomially; see also Section 2.1. Furthermore, it is not possible to give theoretical upper bounds on the relative excess of local minima. So there is a considerable difference between the worst-case and empirical average-case behavior of local search algorithms. Theoretical average-case analysis is therefore useful to provide a better understanding of local search algorithms. However, only a few results on average-case analysis of local search are presented in the literature.

The TSP, defined in Example 2.1, belongs to the class of NP-hard problems [Garey & Johnson, 1979]. We consider only *symmetric* TSP instances, in which distances satisfy $d_{ij} = d_{ji}$, for each $i, j \in V$. The average cost of the solutions found by local search strongly depends on the choice of the neighborhood structure. Therefore, various neighborhood structures have been introduced for the TSP, most of which are based on edge exchanges. In the 2-opt neighborhood $\mathcal{N}_2$ of Croes [1958], formally specified in Example 2.1, a tour t' is a neighbor of tour t, if t' can be obtained from t by removing two edges and inserting two edges such that t' is obtained.

For an overview of the worst-case complexity and empirical behavior of local search for the TSP, we refer to [Johnson & McGeoch, 1996]. Others have addressed the theoretical average-case behavior of local search. Stadler & Schnabl [1992] investigate the structure of the 2-opt neighborhood using simplifications for dependencies between neighbors that lead to a flawed model. Kern [1989] showed with a probabilistic analysis that the 2-opt algorithm for Euclidean instances of the TSP has an average-case running time that is polynomially bounded. Other probabilistic models for local search have been studied by Tovey [1985]. In this study artificial problems are considered with special neighborhood graphs consisting of regular structures, e.g., the hypercube. The cost function for these problems is chosen to induce an orientation on this graph. Different cost distributions are considered and for some cases low-order polynomial average-case running times are proved. More recently, Chandra, Karloff & Tovey [1994] obtained similar results for 2-opt and 3-opt algorithms for the TSP, and derived upper bounds on the average cost of the local minima obtained by these algorithms. Most effort is focused on providing upper bounds on the average-case behavior of local search. We are interested in the actual distribution of the cost of local minima and the required number of steps to find them, instead of upper bounds on the average-case behavior only.

4.1 A probabilistic analysis for the 2-opt neighborhood

In this section we discuss the distribution of local minima found by iterative best improvement algorithms with the 2-opt neighborhood and the distribution of the number of steps required to find local minima. The distribution of local minima and the distribution of final solutions found by iterative improvement are not equivalent because not all local minima have the same probability of being found by an iterative improvement algorithm. It may be the case that local minima with low cost have a larger attraction region than local minima with high cost and are therefore more often found by local search.

An instance of the TSP is completely specified by its distance matrix. As the first step in our approach we assume that the distances d_{ij} are independently drawn from a given distribution. Such instances are called *random distance matrix* instances. Such instances are commonly considered in probabilistic analysis; see for instance [Kirkpatrick & Toulouse, 1985; Weinberger, 1991]. However, it should be noted that the assumption of independence is in fact a restriction that excludes Euclidean instances, because then the distances are dependent. It is possible to associate a class of instances with a distribution by letting the edge lengths $\underline{d}_{ij}$ be independent identically distributed random variables with mean μ_l and variance σ_l^2. Consequently, we can also view the cost $\underline{f}(t)$ of a tour t as a random variable, i.e., the sum of n independent, identically distributed edge lengths, where n is the number of cities in an instance. According to the central limit theorem, $\underline{f}(t)$ has approximately a normal distribution with mean $\mu = n\mu_l$ and variance $\sigma^2 = n\sigma_l^2$. The corresponding density of tour lengths is called ω_{tour}.

4.1.1 The distribution of final solutions

Consider a tour $t \in \mathcal{S}_k$, where $\mathcal{S}_k$ is the set of solutions that can be reached in k best improvement steps, starting from an arbitrary initial solution. The analysis of iterative best improvement is based on the computation of the *step probability*

$$g_k(c, c') = \mathbb{P}\{\forall_{t' \in \mathcal{N}_2(t)}\ \underline{f}(t') > c' \mid \underline{f}(t) = c \wedge t \in \mathcal{S}_k\}, \tag{4.1}$$

that is, the conditional probability that all neighbors of tour t have costs larger than c' given that a tour t found after k steps has cost c. The computation of (4.1) is discussed in the next section. Various notions can be expressed in terms of the step probability g defined in (4.1). First, note that $g_0(c, c)$ is the probability that an arbitrary tour with cost c is a local minimum. Hence, the density of local minima is given by

$$A^{-1} g_0(c, c)\omega_{\text{tour}}(c)\,, \tag{4.2}$$

where A is the probability that an arbitrary tour is a local minimum, given by

$$\int_{-\infty}^{\infty} g_0(c, c)\omega_{\text{tour}}(c)dc. \tag{4.3}$$

Consider a tour $t \in \mathcal{S}_k$ that is not a local minimum. Let $\mathcal{S}'$ denote the set of local minima. Recall that in best improvement a step is made to a neighbor with lowest cost. To investigate such a step, we compute for $c' < c$ the probability

$$\mathbb{P}\{\min_{t' \in \mathcal{N}_2(t)} \underline{f}(t') \le c' \mid \underline{f}(t) = c \wedge t \in \mathcal{S}_k \setminus \mathcal{S}'\} =$$

$$\frac{\mathbb{P}\{\min_{\,t'\in\mathcal{N}_2(t)}\underline{f}(t') \le c' \mid \underline{f}(t) = c \wedge t \in \mathcal{S}_k\}}{\mathbb{P}\{t \in \mathcal{S}_k \setminus \mathcal{S}' \mid \underline{f}(t) = c\}} = \frac{1 - g_k(c, c')}{1 - g_k(c, c)},$$

where we have used the fact that t cannot be a local minimum if $\underline{f}(t) = c$ and $\min_{\,t'\in\mathcal{N}_2(t)} \underline{f}(t') \le c' < c$. The density corresponding with the above probability is

$$P_k(c, c') = \frac{\partial}{\partial c'}\mathbb{P}\{\min_{t'\in\mathcal{N}_2(t)} \underline{f}(t') \le c' \mid \underline{f}(t) = c \wedge t \in \mathcal{S}_k \setminus \mathcal{S}'\} = \frac{-\frac{\partial}{\partial c'} g_k(c, c')}{1 - g_k(c, c)}.$$

This expression is the density as function of c' of the cost after $k + 1$ best improvement steps, given that the tour found after k steps has cost c and is not a local minimum.

The density of the local minima found after k best improvement steps can be described by the following recurrence relations. Let ρ_k be the density of the local minima found after at most k steps and let η_k be the density of the remaining tours. If we start the sequence of best improvement steps with a randomly generated tour, then $\eta_0 = \omega_{\text{tour}}$ and $\rho_0(c) = 0$, for all c. After the k^{th} step the density of the residual tours equals η_k. Consider such a tour with cost c. There is a probability $g_k(c, c)$ that it turns out to be a local minimum. This is found out in the $k + 1^{\text{th}}$ step, hence

$$\rho_{k+1}(c) = \rho_k(c) + g_k(c, c)\eta_k(c). \tag{4.4}$$

On the other hand, there is a probability $1 - g_k(c, c)$ that the tour with cost c is not a local minimum. Then, with probability density $P_k(c, c')$, it is transformed into a tour with cost c' in the $k + 1^{\text{th}}$ step. Hence,

$$\eta_{k+1}(c') = \int_{c'}^{\infty} \eta_k(c)(1 - g_k(c, c))P_k(c, c')dc. \tag{4.5}$$

This set of recurrence relations allows us to compute the densities of the detected local minima and the residual tours after an arbitrary number of steps. Then, $\lim_{k\to\infty}\eta_k = 0$ and $\lim_{k\to\infty}\rho_k = \rho_{\text{fin}}$, the density of the final solutions. Moreover, note that $\rho_k - \rho_{k-1}$ is the density of the local minima found in the k^{th} step. So if $\underline{steps}$ is the random variable describing the number of steps until a local minimum is found, then

$$\mathbb{P}\{\underline{steps} = k\} = \int_{-\infty}^{\infty} g_{k-1}(c, c)\eta_{k-1}(c)dc. \tag{4.6}$$

4.1.2 Evaluation of the step probability

The problem has been reduced to evaluating the step probability (4.1). The joint distribution of $\underline{f}(t_0), \underline{f}(t_1), \ldots, \underline{f}(t_b)$ is needed to compute this probability for an arbitrary tour t_0 with neighboring tours $t_1, \ldots, t_b$, where $b = |\mathcal{N}_2(t_0)| = n \cdot (n-3)/2$. These tours have a large number of edges in common, so the corresponding tour lengths are not independent. Two arbitrary tours s and t that have m edges in common, have a covariance given by

$$\mathbb{E}\{(\underline{f}(t) - \mu) \cdot (\underline{f}(s) - \mu)\} = m{\sigma_l}^2,$$

where $\mu = n\mu_l$ is the mean tour length, μ_l the mean edge length, and ${\sigma_l}^2$ the variance of edge lengths. The tour t_0 and each of its neighbors t_i, with $1 \leq i \leq b$, have $n-2$ edges in common. However, two neighbors of t_0, say t_i and t_j, can have $n-3$ or $n-4$ edges in common. The case of $n-3$ common edges occurs if, while going from t_0 to t_i respectively t_j, a common edge is removed or inserted. Consequently, the $(b+1) \times (b+1)$ covariance matrix of the random variables $\underline{f}(t_0), \underline{f}(t_1), \ldots, \underline{f}(t_b)$ is rather complicated, which makes it difficult to compute or numerically approximate the right hand side of (4.1) for the 2-opt neighborhood. Therefore, in the remaining part of this section we concentrate on the neighborhood $\mathcal{N}_2'$ that is a restricted version of the 2-opt neighborhood in Example 2.1 defined by

$$\mathcal{N}_2'(t) = \{\tau_2(t, e_i, e_{h(t)}) \mid 1 \leq i < h(t) - 1 \vee h(t) + 1 < i \leq n\}.$$

So whereas in the conventional 2-opt neighborhood two arbitrary edges can be removed, here one of the edges to be removed from a tour $t = \{e_1, \ldots, e_n\}$ is fixed and is given by $h(t)$ for a function $h : \mathcal{S} \to \{1, \ldots, n\}$. In this neighborhood structure a tour t has only $b = |\mathcal{N}_2'(t)| = n-3$ neighbors, and all these neighbors have $n-3$ edges in common.

If we assume that the probability of (4.1) does not depend on the number of local search steps, so $g_k(c, c') = g(c, c')$ for all k, then the transition properties and several related notions for the best improvement algorithm can be computed for the $\mathcal{N}_2'$ neighborhood structure. Note that $g_0(c, c)$ can be computed without this assumption. If edge lengths are normally distributed with mean μ_l and variance ${\sigma_l}^2$, then the costs $\underline{f}(t_0), \underline{f}(t_1), \ldots, \underline{f}(t_b)$ have a joint normal distribution [Papoulis, 1965] with mean μ and covariance matrix R given by

$$\begin{array}{lll} R_{ii} & = n{\sigma_l}^2 & \text{for } 0 \leq i \leq b \\ R_{i0} = R_{0i} & = (n-2){\sigma_l}^2 & \text{for } 1 \leq i \leq b \\ R_{ij} = R_{ji} & = (n-3){\sigma_l}^2 & \text{for } 1 \leq i < j \leq b, \end{array}$$

where R_{ii} gives the variance, R_{i0} the covariance between t_0 and its neighbors, and R_{ij} the covariance between two neighbors of t_0. As (4.1) is a conditional probability, we have to compute the conditional density $\omega_{\text{cond}}(c_1, \ldots, c_b \mid c)$, i.e., the density of $\underline{f}(t_1), \ldots, \underline{f}(t_b)$ given that $\underline{f}(t_0) = c$. It is a known result (see for instance [Papoulis, 1965]) that ω_{cond} is again a joint normal density, with all means equal to $\mu' = \mu + R_{i0}/R_{00}(c - \mu) = (1 - 2/n)c + (2/n)\mu$. The variances are given by $r_{ii} = R_{ii} - {R_{0i}}^2/R_{ii} = (4 - 4/n){\sigma_l}^2$ and the covariances by $r_{ij} = R_{ij} - R_{i0}R_{j0}/R_{00} = (1 - 4/n){\sigma_l}^2$. Now (4.1) can be rewritten as

$$g(c, c') = \int_{c'}^{\infty} dc_1 \cdots \int_{c'}^{\infty} dc_b \; \omega_{\text{cond}}(c_1, \ldots, c_b \mid c).$$

The essential observation is that now all covariances are equal and positive. In this case the b-fold integral can be simplified to a single integral [Stuart & Ord, 1987]. This results into

$$g(c, c') = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-\frac{s^2}{2}} \left(\frac{1}{2} \operatorname{erfc} \left(\frac{c' - (1 - \frac{2}{n})c - 2\mu_l}{\sqrt{6}\sigma_l} - s\sqrt{\frac{1}{6} - \frac{2}{3n}} \right) \right)^b ds,$$

which can be computed numerically.

4.1.3 Empirical results

We validate our model for the density of final solutions ρ_{fin} and for the density (4.6) of the required number of steps by comparing them with empirically obtained densities. We consider instances with 200 and 400 cities in which edge lengths are distributed according to a standard normal distribution, which implies that tour lengths can be negative. Other distributions of edge lengths in which costs of neighbors can be approximated by a joint normal distribution, may also be considered. The results are computed from averages over five instances, although there is little difference between the means of local minima in individual instances since these quantities are self-averaging, which means that these quantities are equal for sufficiently large instances in which edge lengths are samples from the same distribution. For each instance we have sampled 20,000 local minima found by an iterative best improvement algorithm. The fixed edge $h(t)$ in the neighborhood $\mathcal{N}_2'(t)$ is randomly chosen for a tour t.

Figure 4.1 and 4.2 give the results for instances with 200 and 400 cities, respectively. In these figures, the density ρ_{fin} of the costs of final solutions found by a best improvement algorithm is depicted. Furthermore, the density of (4.6) of the number of steps required by a best improvement algorithm to find a local minimum is given. We observe that the theoretical predictions fit well with the empirical results, so our model adequately describes the behavior of local search for these instances.

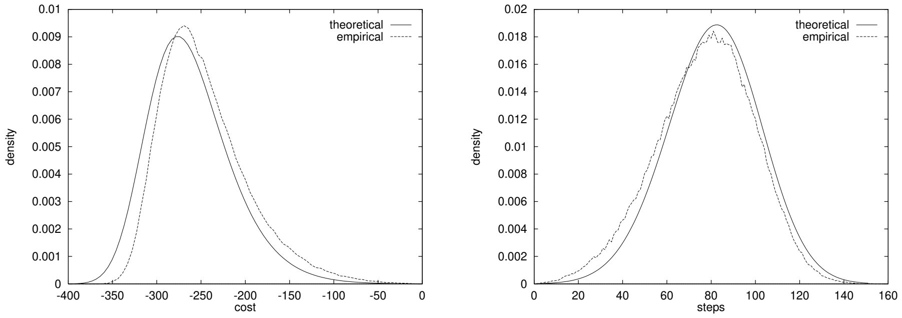

Figure 4.1: Results for instances with 200 cities.

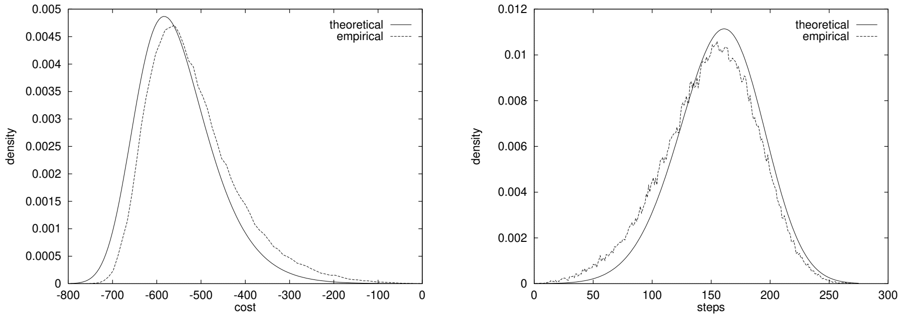

Figure 4.2: Results for instances with 400 cities.

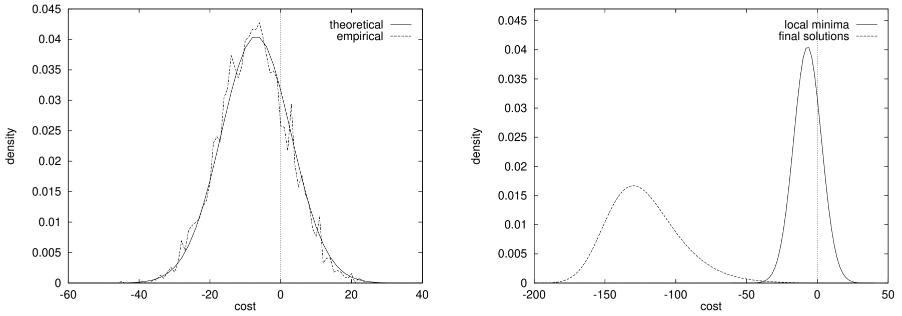

Figure 4.3: Density of local minima.

The probability that an arbitrary tour is a local minimum is given by (4.3). For $n = 100$ this probability amounts to $1.23 \cdot 10^{-4}$. We determined empirically the density of local minima for five instances with 100 cities by randomly generating 37 million tours of which 4410 turned out to be local minima. The lefthand side of Figure 4.3 gives the corresponding empirical density as well as the theoretical density of local minima given by (4.2). Again, we obtain a good fit between theoretical and empirical results.

The righthand side of Figure 4.3 presents the density of local minima as given by (4.2) and the density of final solutions as obtained by iterative best improvement for instances with 100 cities. Typically, the average cost of final solutions is much lower than the average cost of local minima. This implies that low-cost local minima have a much larger attraction region and therefore a much higher probability of being found by local search than high-cost local minima, which is of course an advantageous property for the performance of local search.

Empirical distributions for 2-opt and Lin–Kernighan. Next, we present distributions of final solutions obtained by a first improvement 2-opt algorithm and a Lin–Kernighan algorithm, discussed in more detail in Chapter 5, to show that the distribution shapes observed for the more artificial random distance matrix instances and restricted 2-opt neighborhood we used in our theoretical analysis, are also instructive for real-world Euclidean instances.

Figure 4.4 shows empirical results for a number of real world instances from the TSP library of Reinelt [1991], where the number in an instance name denotes

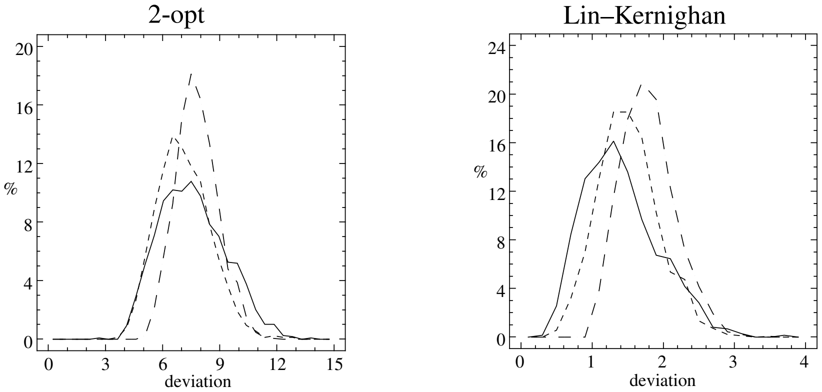

Figure 4.4: The distribution of final solutions. For the 2-opt algorithm, the solid line gives the results for kro200, the short-dashed line for lin318, and the long-dashed line for pcb442. For the Lin–Kernighan algorithm these lines give results for pcb442, u574, and pr1002, respectively.

its size. In these figures, the frequency is plotted against the relative excess over the optimal solution value. Our empirical investigation of the distribution of final solutions obtained by 2-opt and Lin–Kernighan algorithms reveals two interesting properties. First, we have statistically validated that final solutions are distributed according to gamma distributions. Secondly, we have observed that the standard deviation of these distributions decreases when the instance size grows. These characteristics are an important aspect of a neighborhood for iterated local search or other more advanced local search algorithms, because they indicate the additional computational effort required by iterated local search to find lower-cost local minima.

4.2 A semi-empirical analysis of iterated local search

Iterated local search algorithms repeatedly execute an iterative improvement algorithm. Each time the iterative improvement algorithm terminates, the last or best local minimum found is modified and the iterative improvement algorithm is restarted with the modified local minimum [Johnson, 1990; Martin, Otto & Felten, 1991; Boese, Kahng & Muddu, 1994]. The best heuristic to handle the TSP is the iterated Lin–Kernighan algorithm of Johnson [1990]. In Johnson's iterated Lin–Kernighan algorithm the best local minimum found so far is modified by a 4-exchange, which replaces four edges by four new edges, and used as starting solution for the next run of the Lin–Kernighan heuristic.

In this section we approximate the average running times iterated local search algorithms need for finding a solution within a given relative excess over the optimal solution. Time is measured by the number of tours whose costs are evaluated. Our analysis is of a semi-empirical nature, which means that the average number of evaluations is expressed as a function of empirically obtained parameters. These parameters are the distribution of final solutions and the average number of steps needed to find them.

The average number of iterations and the number of evaluations performed per iteration are needed to approximate the average running time. We, furthermore, assume that an iterated local search algorithm samples local minima, which means that we neglect the intermediate solutions generated by the iterative improvement algorithm. Let $\underline{itnr}_\epsilon$ be the random variable that gives the number of iterations until a solution with a given relative excess ϵ has been found, then

$$\mathbb{P}\{\underline{itnr}_\epsilon \leq i\} = 1 - \prod_{k=1}^{i}(1 - \mathbb{P}\{\underline{f}(s_k) \leq (1+\epsilon) f_{\text{opt}}\}), \qquad (4.7)$$

where s_k is the final tour obtained at iteration $k \in \mathbb{N}$.

To determine the average number of iterations needed, we assume that the modification mechanism is primarily a diversification mechanism. Thus, we consider subsequent local minima to be independent, and consequently the probability $\mathbb{P}\{f(s_k) \leq c\}$ is independent of k. This may seem a rather strong assumption since the modification mechanism only changes four edges in a local minimum, but the final solution obtained from this modified local minimum typically differs in much more edges from the original local minimum. Given the density ρ_{fin} of the costs of final solutions, we then have

$$\mathbb{P}\{f(s_k) \leq (1+\epsilon) f_{\text{opt}}\} = \int_{-\infty}^{(1+\epsilon) f_{\text{opt}}} \rho_{\text{fin}}(c) dc.$$

Let p_ϵ denote the above probability, then the probability of (4.7) is given by

$$\mathbb{P}\{\underline{itnr}_\epsilon \leq i\} = 1 - (1 - p_\epsilon)^i. \tag{4.8}$$

Expression (4.8) implies that the number of iterations is distributed according to a geometrical distribution. So the average number of iterations to find a solution within a given relative excess ϵ is equal to $1/p_\epsilon$.

In order to approximate the average number of evaluations needed per iteration, we need the following observations. After the first iteration the iterative improvement algorithm starts with a tour that differs only four edges from a local minimum. Consequently, the average number of evaluations required in subsequent iterations is substantially lower than that required in the first iteration, which is also observed from empirical results. Hence, we differentiate between the following two values.

- k_1, the average number of evaluations needed to reach a local minimum from a start solution.
- k_2, the average number of evaluations needed to reach a local minimum from a solution obtained by modifying a local minimum.

The values of k_1 and k_2 are obtained empirically. They depend strongly on the heuristic used to construct start solutions and on the instance at hand.

Let $\underline{evals}_\epsilon$ denote the random variable that gives the number of evaluations until a solution with a given relative excess ϵ is found. Then, $\underline{evals}_\epsilon = k_1 + (\underline{itnr}_\epsilon - 1) k_2$, and

$$\mathbb{P}\{\underline{evals}_\epsilon \leq j\} = \mathbb{P}\{\underline{itnr}_\epsilon \leq \lceil \frac{j - k_1}{k_2} \rceil\} = \\ \begin{cases} 0 & \text{if } j < k_1, \\ 1 - (1 - p_\epsilon)^{\lceil \frac{j-k_1}{k_2} \rceil} & \text{if } j \geq k_1. \end{cases} \tag{4.9}$$

The expected number of evaluations needed to find a solution within a given relative excess ϵ is equal to $k_1 + (\frac{1}{p_\epsilon} - 1) k_2$.

4.2.1 Empirical results

We have analyzed iterated 2-opt and Lin–Kernighan algorithms with a 4-exchange similar to the one utilized by [Johnson, 1990] of the best local minimum found as modification mechanism. Both algorithms have been tested on instances from the TSP library. In order to acquire the average number of evaluations empirically, 100 executions of the iterated local search algorithm have been performed for each instance.

Figures 4.5 and 4.6 show the probability of (4.9) for finding a solution within a relative excess ϵ in a given number of evaluations by the iterated 2-opt algorithm for the instance kro200 and the iterated Lin–Kernighan algorithm for the instance u574, respectively. The value of p_ϵ, the probability that a local minimum has at most the given relative excess ϵ, and the value of k_1 and k_2 are obtained empirically by sampling 1,000 local minima obtained from random start solutions or applying 4-exchanges to local minima, respectively. The solid curves represent the

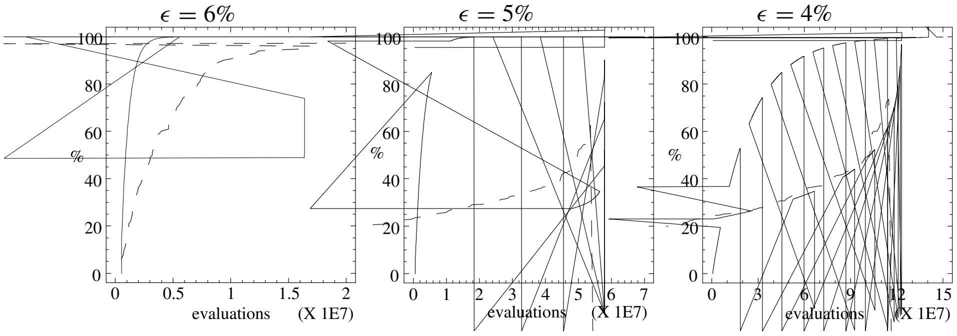

Figure 4.5: Probability of finding a tour with at most relative excess ϵ as function of the running time using iterated 2-opt for kro200.

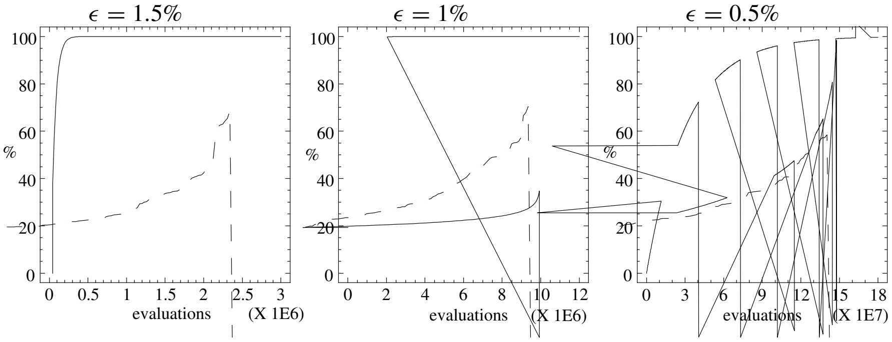

Figure 4.6: Probability of finding a tour with at most relative excess ϵ as function of the running time using iterated Lin–Kernighan for u574.

theoretically predicted distributions, the dashed curves the empirically obtained distributions. Iterated local search for other instances, not shown in Figure 4.5 and 4.6, displays a similar behavior.

We observe that a good agreement between our theoretical prediction and the empirical results is only obtained for low values of the desired relative excess ϵ. This is explained by the observation that even a small under- or overestimation of the probability p_ϵ can have a large effect on the theoretical curves. Furthermore, we have found that the empirical distributions fit well with geometrical distributions that are slightly translated to compensate for the initial behavior.

4.3 Parallel iterated local search

In Section 4.2 we have given an expression for the probability that an iterated local search algorithm has found a solution with a given relative excess ϵ over the optimal solution assuming that subsequent local minima are independent. Consider an algorithm that carries out P independent runs of an iterated local search algorithm, and let $\underline{itnr}_{\epsilon,P}$ be the random variable that gives the number of iterations until this algorithm has found a solution with a given relative excess ϵ. We have that

$$\mathbb{P}\{\underline{itnr}_{\epsilon,P} \leq i\} = 1 - (1 - p_\epsilon)^{iP},$$

where p_ϵ denotes the probability that a local minimum has a relative excess of at most ϵ. Hence, the expected number of iterations needed by P parallel runs of an iterated local search algorithm equals

$$\frac{1}{1 - (1 - p_\epsilon)^P}.$$

Recall that k_1 and k_2 denote the average number of evaluations needed for finding the first local minimum and that for finding subsequent local minima, respectively. At least one iteration is needed so the average running time for finding a suboptimal solution with P independent parallel runs can be approximated by

$$k_1 + (\frac{1}{1 - (1 - p_\epsilon)^P} - 1)k_2.$$

Consequently, the expected speed-up for P independent parallel runs is given by

$$su(P) = \frac{k_1 - k_2 + \frac{k_2}{p_\epsilon}}{k_1 - k_2 + \frac{k_2}{1-(1-p_\epsilon)^P}}.$$

In order to analyze the scalability of this approach, we still have to determine the number of processors that can be employed effectively. Therefore, we examine the typical speed-up of the algorithm as a function of the problem parameters.

The speed-up is bounded by

$$\lim_{P\to\infty} su(P) = 1 + (\frac{1}{p_\epsilon} - 1)\frac{k_2}{k_1}.$$

We see that more processors can be employed effectively when final solutions with low relative excess ϵ are sought because then p_ϵ is small. More processors can also be employed when k_1 and k_2, the times for finding the first local minimum and subsequent local minima, are roughly equal. This generally occurs when smaller instances are attempted, or when better-quality starting solutions are utilized. In particular, for $k_2/k_1 \to 1$ holds that the speed-up is bounded by $1/p_\epsilon$. Note that $k_2/k_1 = 1$ means that the probability of finding a suboptimal solution is distributed according to a geometrical distribution. On the other hand, for $k_2/k_1 \to 0$ there is no speed-up at all. Furthermore, $\lim_{p_\epsilon\to 0} su(P) = P$, which implies that in this case a nearly linear speed-up can be achieved with multiple independent walks.

4.3.1 Computational study

Next, we present a computational study of the expected behavior of parallel independent runs of an iterated local search algorithm, which can be characterized by the obtained efficiency.

In order to compute the expected speed-up, the empirical distributions presented in Section 4.2 are fitted with translated geometrical distributions, which is necessary to get a good fit for the initial part of the distribution that corresponds with the time needed to find the first local minimum. The empirical distributions are accurately described by this translated geometrical distribution, except for the distributions in which the desired relative excess was relatively high. In these cases, the probability of finding a solution with a given relative excess increases initially less than exponentially. Consequently, our approximation overestimates the achieved efficiency for runs in which final solutions with high relative excesses suffice.

Figure 4.7 shows efficiencies that can be reached with parallel algorithms that perform multiple independent runs of iterated 2-opt or Lin–Kernighan algorithms. The expected efficiency is plotted as a function of the number of employed processors. We observe that good speed-ups can be obtained with multiple independent runs of an iterated local search algorithm. Moreover, the efficiency increases when the desired relative excess is lowered, as explained by the observations in the previous section.

Summarizing this chapter, we remark that it is possible to achieve good speed-ups with multiple independent walks of an iterated local search algorithm or other algorithms that sample local minima, such as tabu search algorithms, if the probability of finding suboptimal solutions is distributed geometrically. For this it is

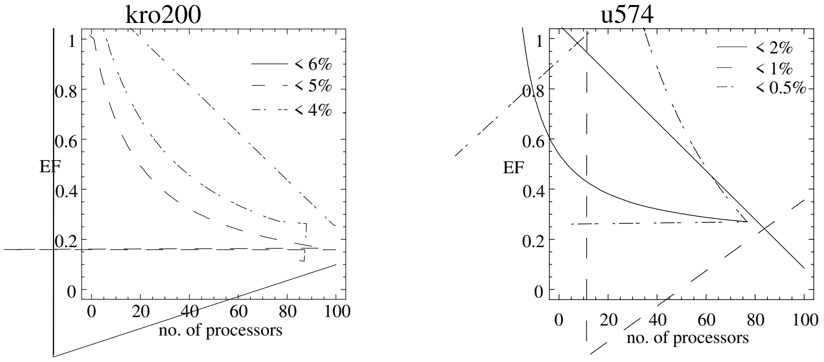

Figure 4.7: Efficiency of parallel iterated 2-opt (left) and iterated Lin–Kernighan (right) for kro200 and u574 and given relative excesses.

essential that the time to reach a region in the neighborhood graph that contains high-quality solutions, starting from an initial solution, is roughly equal to the time to reach other such regions from this region. Nearly linear speed-ups can be achieved if solutions are sought whose relative excess is substantially lower than the average relative excess of local minima. An important prerequisite for these results is that the iterated local search algorithm at hand should eventually be able to find solutions within a given relative excess when continued sufficiently long, which in fact requires that a neighborhood is sufficiently connected and that each solution in a connected component of the neighborhood graph has a positive probability of being visited in a local search walk.

5

The Traveling Salesman Problem

In this chapter we present local search algorithms with multiple-step parallelism for the traveling salesman problem based on the 2-opt and 3-opt neighborhoods of Lin [1965]. Furthermore, we present a parallel implementation of the variable-depth algorithm of Lin & Kernighan [1973]. Our parallel Lin–Kernighan algorithm uses neighborhood reduction techniques and efficient data structures. The algorithm is tested on a network of 64 transputers and on a network of 32 PowerPC's. Its performance is empirically analyzed for real-world problem instances with up to 85,900 cities from Reinelt's TSP library. Our parallel Lin–Kernighan algorithm is competitive with the best known sequential implementations of the Lin–Kernighan algorithm, both with respect to quality of final solutions and running times.

5.1 Local search for the traveling salesman

Much of the theory for combinatorial optimization has been developed using the TSP as a proving ground [Lawler, Lenstra, Rinnooy Kan & Shmoys, 1985]. Impressive results have been obtained using elaborate branch and bound algorithms. The largest real-world problem solved to optimality is currently an instance with 7,397 cities, which required several months of running time on a network of powerful workstations [Applegate, Bixby, Chvátal & Cook, 1994].

A classical example of a local search algorithm for the TSP is the variable-depth algorithm of Lin & Kernighan [1973]. The best approximation algorithm for the TSP is the iterated Lin–Kernighan algorithm of Johnson [1990], which uses multiple runs of the Lin–Kernighan algorithm with a random 4-exchange to obtain a new starting solution. It finds solutions with average quality of 0.7% over the Held-Karp lower bound on random Euclidean instances.

However, large real-world instances, such as those originating from circuit board lay-outing [Reinelt, 1992] and x-ray crystallography [Bland & Shallcross, 1989], still require substantial amounts of running time. To reduce the running time of the Lin–Kernighan algorithm various advanced data structures and neighborhood reduction techniques have been proposed [Bentley, 1990; Fredman, Johnson, McGeoch & Ostheimer, 1993; Reinelt, 1994]. One way to further reduce running time is by using parallelism. In the literature several examples of parallel local search algorithms for the TSP are given. Multiple-walk parallelism for the TSP is studied in [Malek, Guruswamy & Pandya, 1989; Diekmann, Lüling & Simon, 1993]. Single-step parallelism is studied by Chakrapani & Skorin-Kapov [1993a], and multiple-step parallelism is studied in [Felten, Karlin & Otto, 1985; Allwright & Carpenter, 1989; Fiechter, 1994; Bachem, Steckemetz & Wottawa, 1994; Verhoeven, Aarts, Van de Sluis & Vaessens, 1992; Verhoeven & Aarts, 1994]. All these algorithms, however, are not competitive, both with respect to running times and quality of final solutions, with the best existing sequential algorithms, i.e., efficient implementations of the Lin–Kernighan algorithm. Our goal is to design a parallel algorithm that finds the same quality solutions as the best sequential algorithm in a smaller amount of running time.

Neighborhoods for the TSP. The TSP can be reformulated as the problem of finding a Hamiltonian cycle of minimal length in a complete weighted graph; see Example 2.1. We concentrate on the Euclidean TSP in which the cities are given by coordinates in a Euclidean space. Most neighborhood structures for the TSP are based on the exchange of a number of edges. Besides the 2-opt neighborhood of Example 2.1, we mention the following ones.

- 3-*opt.* In this neighborhood a tour t' is a neighbor of tour t if it can be obtained from t by removing three edges and inserting three other edges. This exchange is called a 3-exchange.
- *Or-opt.* A tour t' is a neighbor of tour t if it can be obtained from t by removing a path with at most three subsequent cities from t and inserting it between two other cities in t. This is called an Or-exchange.
- *Lin–Kernighan.* This is a variable-depth neighborhood structure in which the number of edges that is exchanged to obtain a neighbor is not fixed but depends on the current tour t. Basically, a series of 2-exchanges is constructed by a combination of first improvement, best improvement, and limited backtracking [Lin & Kernighan, 1973].

The empirical average-case time complexity of local search algorithms using the above neighborhood structures is low-order polynomial. Furthermore, computational results show that good-quality solutions can be found with these neighborhood structures. 2-opt, 3-opt, and Lin–Kernighan give final solutions with average quality of 6.4%, 3.5%, and 2.1%, respectively, over the Held-Karp lower bound on random Euclidian instances. For excellent overviews of what can be achieved with local search for the TSP, we refer to [Johnson, 1990; Johnson & McGeoch, 1996; Reinelt, 1994].

The outline of the remainder of this chapter is as follows. Section 5.2 presents parallel 2-opt and 3-opt algorithms. Section 5.3 discusses the implementation of the Lin–Kernighan algorithm. Section 5.4 discusses the parallel Lin–Kernighan algorithm and presents numerical results on different parallel platforms.

5.2 Parallel 2-opt and 3-opt algorithms

The concept of distributed neighborhoods for multiple-step parallelism has been introduced in Section 3.2. A distributed neighborhood structure defines a partition of neighborhoods by partitioning the domain of the exchange function. It also defines a combination function that combines several exchanges to a feasible solution. In order to enable efficient combination of 2-exchanges it is convenient to decompose a tour into several partial solutions each consisting of two paths. However, in this way it is not possible to examine 2-exchanges in which edges from different partial solutions are replaced. To guarantee that all exchanges examined in the 2-opt neighborhood are also examined in the distributed 2-opt neighborhood, several tour decompositions have to be included in the distribution structure of a distributed 2-opt neighborhood.

5.2.1 A distributed 2-opt neighborhood

Next, we define a distributed 2-opt neighborhood $\mathcal{D}_2$ that is isomorphic with the 2-opt neighborhood $\mathcal{N}_2$ defined in Example 2.1. Recall that a distributed neighborhood is a triple (Δ, λ, ϕ), where Δ is a distribution structure, λ a domain distribution, and ϕ a combination function.

First, we define a distribution structure that gives a set of distributions in which each partial solution consists of two paths. A tour is a set of N edges, and we index the successive edges with indices from 0 to N. A partial solution δ_p is defined as a set of edges that constitute two paths. One path is the set of edges with indices in $D(p) = \{m_p, \dots, n_p - 1\}$ and contains a_p edges, and the other path is the set of edges with indices in $D'(b, p) = \{m'_p - b, \dots, n'_p - b - 1\}$ for some b, with $0 \leq b \leq B$, and it contains a'_p edges. The parameter B is needed to prove isomorphism properties of various distributed neighborhood structures based on this

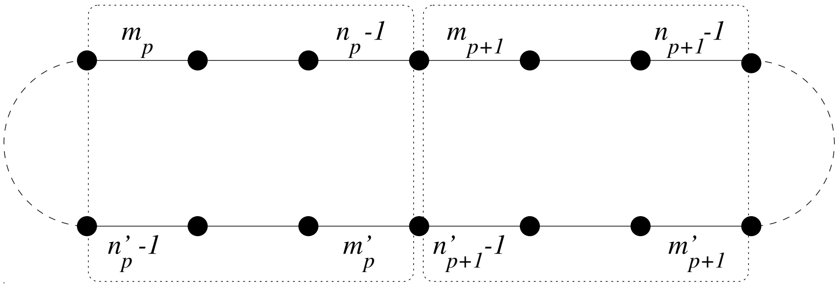

Figure 5.1: Linear distribution.

distribution structure, as we see later on. Different distributions are generated by assigning one edge to an adjacent partial solution. Figure 5.1 gives an example of this distribution for partial solutions δ_p, δ_{p+1} with $a_p = a'_p = a_{p+1} = a'_{p+1} = 3$ and $b = 0$. In the following definition this distribution structure is stated formally. All additions and subtractions are done modulo N.

Definition 5.1. Let P with $P \geq 1$ and B with $B \geq 1$ be given. Let $a, a' : \{0, \ldots, P\} \rightarrow \{0, \ldots, N\}$ be mappings such that $a_p \geq 1$, $a'_p \geq 1$ for all $0 \leq p < P$, and $a_p + a'_p = N \operatorname{div} P + 1$ for $0 \leq p < N \bmod P$, and $a_p + a'_p = N \operatorname{div} P$ for $N \bmod P \leq p < P$. Furthermore, define $m, m', n, n' : \{0, \ldots, P\} \rightarrow \{0, \ldots, N\}$ by $m_0 = 0$, $n_p = m_p + a_p$ for $0 \leq p < P$, $m_{p+1} = n_p$ for $0 \leq p < P - 1$, and $m'_{P-1} = n_{P-1}$, $n'_p = m'_p + a'_p$ for $0 \leq p < P$, and $m'_{p-1} = n'_p$ for $1 \leq p < P$. Define for all $0 \leq p < P$ and $0 \leq b \leq B$

$$D(p) = \{m_p, \ldots, n_p - 1\},$$

$$D'(b, p) = \begin{cases} \{m'_p - b, \ldots, n'_p - b - 1\} & \text{for } 0 < p < P - 1, \\ \{m'_p - b, \ldots, n'_p - 1\} & \text{for } p = 0, \\ \{m'_p, \ldots, n'_p - b - 1\} & \text{for } p = P - 1. \end{cases}$$

Let $t \in \mathcal{S}$ with $t = \{e(i) \mid 0 \leq i < N\}$, then a solution distribution $\delta_t(b, c)$ of t, for $0 \leq b \leq B$ and $0 \leq c < N$, is defined by

$$\delta_t(b, c)(p) = \{e(i - c) \mid i \in D(p) \cup D'(b, p)\}.$$

A *linear* distribution structure Δ is defined by

$$\Delta(t, P) = \{\delta_t(b, c) \mid 0 \leq b \leq B \wedge 0 \leq c < N\}.$$

□

The following definition gives the domain distribution λ that defines which exchanges are included in the local neighborhoods of partial solutions.

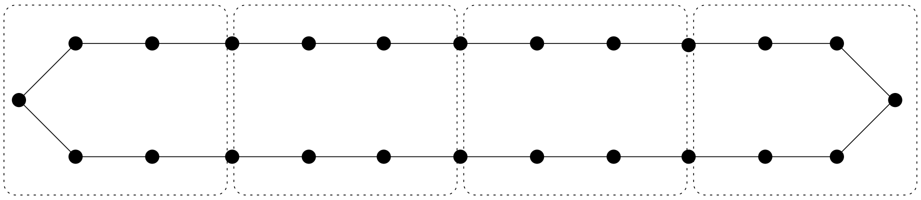

Figure 5.2: Distribution of a tour.

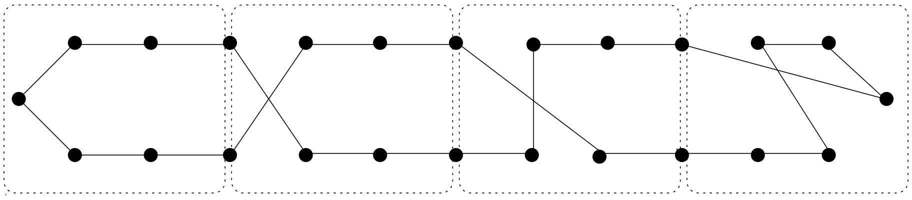

Figure 5.3: Examples of local neighbors.

Definition 5.2. Let Δ be a linear distribution structure with $B = 1$, and let $t \in \mathcal{S}$, $0 \le b \le B$, $0 \le c < N$, $0 \le p < P$, and let $\tau = \delta_t(b, c)(p)$ be a partial solution with $\delta_t(b, c) \in \Delta(t, P)$. Define a domain distribution λ as follows,

$$\lambda_p(\tau) = \begin{cases} \{(e(m_p - c), e(i - c)) \mid i \in D'(0, p)\} & \text{if } b = 0, \\ \{(e(i - c), e(m'_p - 1 - c)) \mid i \in D(p)\} & \text{if } b = 1. \end{cases}$$

□

Each local neighborhood defined by the domain distribution λ consists of at most N/P neighbors obtained by applying 2-exchanges to partial solutions that remove a fixed edge in one path of a partial solution and one arbitrary edge in the other path of a partial solution. Figure 5.2 gives an example of a distribution of a tour and Figure 5.3 gives some examples of local neighbors obtained by applying 2-exchanges to partial solutions.

The combination function ϕ, which joins several partial solutions to a feasible tour, is straightforward when a linear distribution structure is utilized, since exchanges applied to a given partial solution do not interact with exchanges applied to other partial solutions. The parts of a tour that are not included in a partial solution can be represented by two imaginary edges that connect terminal cities of a partial solution. The addition of these imaginary edges that are not actually included in a tour to a partial solution transforms it into a single subtour. It is obvious that partial solutions can be merged to a feasible tour as long as each partial solution is a single subtour that includes these imaginary edges; cf. Figure 5.2.

Using the above definitions, the distributed 2-opt neighborhood can be defined as follows.

Definition 5.3. A distributed 2-opt neighborhood structure $\mathcal{D}_2$ is given by a triple (Δ, λ, ϕ), where Δ is a linear distribution structure, with $B = 1$, as specified in Definition 5.1, λ is a domain distribution as specified in Definition 5.2, and the combination function ϕ is given by $\phi(\tau_0, \ldots, \tau_{P-1}) = \bigcup_{0 \leq p < P} \tau_p$ for partial solutions τ_p. □

Next, we show that the distributed neighborhood structure $\mathcal{D}_2$ is isomorphic with $\mathcal{N}_2$, i.e., a local minimum of $\mathcal{D}_2$ is also a local minimum of $\mathcal{N}_2$. For this it has to be shown that each 2-exchange evaluated to determine local optimality for $\mathcal{N}_2$ is also evaluated in $\mathcal{D}_2$. First, we need the following lemma.

Lemma 5.1. *Let D, D' be defined by Definition 5.1. Let $L(p) = \{(m_p, j) \mid j \in D'(0, p)\}$ and $L'(p) = \{(i, m'_p - 1) \mid i \in D(p)\}$. Then,*

$$\cup_{0 \leq q \leq p} \{j - i \mid (i, j) \in L(q) \cup L'(q)\} = \{m'_p - n_p, \ldots, N - 1\}.$$

Proof. Use induction to p. First, note that $\{j - i \mid (i, j) \in L(p) \cup L'(p)\} = \{m'_p - n_p, \ldots, n'_p - m_p - 1\}$. Furthermore holds $n'_0 = N$ and $m_0 = 0$. So for $p = 0$ holds $\{j - i \mid (i, j) \in L(0) \cup L'(0)\} = \{m'_0 - n_0, \ldots, N - 1\}$. For the induction step, note that $\{j - i \mid (i, j) \in L(p+1) \cup L'(p+1)\} = \{m'_{p+1} - n_{p+1}, \ldots, m'_p - n_p - 1\}$. □

Using this lemma, the following result can be obtained.

Theorem 5.1. *$\mathcal{D}_2$ is isomorphic with $\mathcal{N}_2$.*

Proof. Let $t = \{e_1, \ldots, e_N\} \in \mathcal{S}$. Recall that $\mathcal{N}_2(t) = \{\tau_2(t, e_i, e_j) \mid 0 \leq i < N \wedge 0 \leq j - i < N\}$. Applying Lemma 5.1 for $p = P - 1$ shows that $\cup_{0 \leq q < P} \{j - i \mid (i, j) \in L(q) \cup L'(q)\} = \{0, \ldots, N - 1\}$. Hence, all required differences $j - i$ between arguments e_i, e_j of the 2-exchange function τ_2 that are included in $\mathcal{N}_2$ are also included in the domain distribution of $\mathcal{D}_2$ specified by L and L'. Note that L and L' define the indices of the edges removed in the local neighborhoods of partial solutions defined by $\mathcal{D}_2$. Furthermore, $\mathcal{D}_2$ contains N different solution distributions, and consequently $\{e_t(m_p - c) \mid 0 \leq c < N\} = t$ for all p. So all exchanges evaluated for verifying local optimality of t for $\mathcal{N}_2$ are also evaluated for verifying local optimality of t for $\mathcal{D}_2$. □

The parallel complexity of verifying local optimality of a tour for $\mathcal{D}_2$ is equal to $\mathcal{O}(N \cdot \frac{N}{P})$ with P processors since each local neighborhood has size $\frac{N}{P}$ and N distributions have to be considered. The communication overhead for obtaining new distributions is $\mathcal{O}(1)$ on a message-passing MIMD machine with a ring topology , as distribution $\delta_t(1, c)$ can be obtained from distribution $\delta_t(0, c)$, and similarly $\delta_t(0, c+1)$ from $\delta_t(1, c)$, by sending a single edge to an adjacent processor in the ring. Hence, a new distribution can be obtained from the current distribution in constant time as follows. Even-numbered processors first send an edge to right-adjacent processors, subsequently they receive an edge from left-adjacent processors. So the time needed to obtain a new distribution is at most twice the time to send a single edge to a neighboring processor.

The total number of 2-exchanges that are evaluated for verifying local optimality of a tour for $\mathcal{D}_2$ is $\mathcal{O}(N^2)$, because P processors each evaluate $\mathcal{O}(N \cdot \frac{N}{P})$ 2-exchanges. This observation shows that the total computational effort of the parallel algorithm for verifying local optimality is equivalent with that of the sequential algorithm. The number of processors that can be employed is at most $\frac{N}{2}$ since each partial solution must contain at least two edges.

5.2.2 A distributed 3-opt neighborhood

Computational experiments have shown that 3-opt local search algorithms find solutions with considerable lower cost than solutions found by 2-opt algorithms [Lawler, Lenstra, Rinnooy Kan & Shmoys, 1985; Johnson, 1990]. Good results with respect to solution quality and running time are reported in particular with the Or-opt neighborhood. The Or-opt neighborhood is a restricted 3-opt neighborhood in which a path consisting of at most three subsequent cities is inserted between two other cities in a tour. The Or-opt neighborhood can be generalized to the neighborhood $\mathcal{N}_{3,B}$ in which only those 3-exchanges are examined in which the minimum number of adjacent edges, after removal of three edges, is smaller than B. Johnson [1990] reports good quality results with this kind of restricted 3-opt neighborhood. Formally $\mathcal{N}_{3,B}$ is defined as follows.

Definition 5.4. The exchange function $\tau_3 : \mathcal{S} \times (V \times V)^3 \to \mathcal{P}(\mathcal{S})$ gives for each tour $t \in \mathcal{S}$ the set of eight different tours that can be obtained by replacing three edges with three other edges. The restricted 3-exchange neighborhood structure $\mathcal{N}_{3,B}$ is given by $\mathcal{N}_{3,B}(t) = \cup_{0 \leq i < N \wedge i < j < N-B \wedge j < k \leq j+B} \tau_3(t, e_i, e_j, e_k)$. □

Note that the Or-opt neighborhood is a subset of the $\mathcal{N}_{3,3}$ neighborhood. Next, we define a distributed neighborhood $\mathcal{D}_{3,B}$ that is isomorphic with the neighborhood $\mathcal{N}_{3,B}$. The distribution structure of $\mathcal{D}_{3,B}$ is a linear distribution structure as defined in Definition 5.1. Its domain distribution, which specifies the local neighborhood of each partial solution, is defined as follows.

Definition 5.5. Let Δ be a linear distribution structure with $a_p = 1$ and $a'_p \geq B$ for all $0 \leq p < P$, and let $t \in \mathcal{S}$ and $0 \leq b \leq B$. Let $\tau = \delta_t(b, c)(p)$ be a partial solution with $\delta_t(b, c) \in \Delta(t, P)$ for $0 \leq c < N$. Define

$$L(p) = \{(m_p, j, k) \mid m'_p \leq j \leq n'_p - B \ \wedge \ j < k \leq j + B\},$$
$$L'(b, p) = \{(m_p, m'_p - b, m'_p - b + i) \mid 0 < i \leq B\}.$$

Then, domain distribution λ is defined as follows.

$$\lambda_p(\tau) = \begin{cases} \{(e(i-c), e(j-c), e(k-c)) | (i, j, k) \in L(p)\} & \text{if } b = 0, \\ \{(e(i-c), e(j-c), e(k-c)) | (i, j, k) \in L'(b, p)\} & \text{if } b > 0,\ p < P-1, \\ \emptyset, & \text{if } b > 0,\ p = P-1. \end{cases}$$

□

The sets L, L' specify the indices of the edges that are to be removed in a local neighborhood. Each local neighborhood specified by the domain distribution λ either consists of at most $B \cdot \frac{N}{P}$ neighbors obtained by applying 3-exchanges to partial solutions that remove a fixed edge in one path of a partial solution and two arbitrary edges in the other path of a partial solution, or it consists of at most B neighbors obtained by applying 3-exchanges that remove two fixed edges, one in each path, and one arbitrary edge.

The combination function is similar to that of $\mathcal{D}_2$, since a linear distribution structure is used, which implies that local neighbors of partial solutions can be merged to feasible tours, provided that all local neighbors, with addition of two additional edges, are single subtours.

Definition 5.6. A distributed neighborhood structure $\mathcal{D}_{3,B}$, with $B \in \mathbb{N}^+$, is defined by a triple (Δ, λ, ϕ), where Δ is a linear distribution structure as specified in Definition 5.1, λ is a domain distribution as specified in Definition 5.5, and the combination function ϕ is given by $\phi(\tau_0, \dots, \tau_{P-1}) = \bigcup_{0 \le p < P} \tau_p$ for partial solutions τ_p. □

Next, we show that the distributed neighborhood structure $\mathcal{D}_{3,B}$ is isomorphic with $\mathcal{N}_{3,B}$, i.e., a local minimum of $\mathcal{D}_{3,B}$ is also a local minimum of $\mathcal{N}_{3,B}$, which requires that each 3-exchange evaluated to determine local optimality for $\mathcal{N}_{3,B}$ is also evaluated in $\mathcal{D}_{3,B}$. To show this, we need the following lemma.

Lemma 5.2. *Let* $0 \le p < P-1$, *and* $0 \le b \le B$. *Define the set* $W(p)$ *that consists of all differences between indices in* $L(p)$ *and* $L'(b, p)$ *as follows.*

$$W(p) = \begin{cases} \{j - i \mid (i, j, k) \in L(p)\} \cup & \\ \bigcup_{0 < b \le B} \{j - i \mid (i, j, k) \in L'(b, p)\}, & \text{if } 0 \le p < P-1, \\ \{j - i \mid (i, j, k) \in L(p)\} & \text{if } p = P-1. \end{cases}$$

Then,

$$\bigcup_{0 \le q \le p} W(q) = \{m'_p - B - m_p, \dots, N - B\}.$$

Proof. Use induction to p. First, note that $W(p) = \{m'_p - B - m_p, \dots, n'_p - B - m_p\}$. So for $p = 0$ holds $W(p) = \{m'_p - B - m_p, \dots, N - B\}$ as $n'_0 = N$ and $m_0 = 0$. For the induction step, note that $W(p+1) = \{m'_{p+1} - B -$

$m_{p+1}, \ldots, m'_p - B - m_p - 1\}$ since $n'_{p+1} = m'_p$ and $m_{p+1} = n_p = m_p + a_p = m_p + 1$. □

The above lemma leads to following result.

Theorem 5.2. *$\mathcal{D}_{3,B}$ is isomorphic with $\mathcal{N}_{3,B}$.*
Proof. Recall that $\mathcal{N}_{3,B}(t) = \cup_{0 \le i < N \wedge i < j < N-B \wedge j < k \le j+B}\ \tau_3(t, e_i, e_j, e_k)$ for each $t \in \mathcal{S}$. First note that for $p = P - 1$ holds $W(p) = \{m'_p - m_p, \ldots, n'_p - B - m_p\} = \{1, \ldots n'_p - B - m_p\}$ as $m'_p = m_p + 1$. This combined with Lemma 5.2 for $p = P - 2$ shows that $\cup_{0 \le q < P}\ W(q) = \{1, \ldots, N - B\}$. Hence, all required differences of indices $j - i$ and $k - j$ between arguments of the 3-exchange function in $\mathcal{N}_{3,B}$, are also included in the domain distribution of $\mathcal{D}_{3,B}$ specified by L and L'. Furthermore, the distribution structure of $\mathcal{D}_{3,B}$ contains N different solution distributions, and consequently $\{e_t(m_p - c) \mid 0 \le c < N\} = t$ for all p. So all exchanges evaluated for verifying local optimality of t for $\mathcal{N}_{3,B}$ are also evaluated for verifying local optimality of t for $\mathcal{D}_{3,B}$. □

$\mathcal{D}_{3,B}$ contains $P{\cdot}N$ local neighborhoods of partial solutions $\delta_t(0, c)(p)$ with size $(\frac{N}{P} - B)B$ and $P{\cdot}N{\cdot}B$ local neighborhoods of partial solutions $\delta_t(b, c)(p)$ with size B for $0 < b \le B$. As P local neighborhoods can be searched in parallel, the parallel complexity of verifying local optimality of a tour is $\mathcal{O}(NB\frac{N}{P})$ using P processors. The communication overhead for obtaining new distributions is $\mathcal{O}(1)$ since new distributions can be obtained from current distributions in a similar way as discussed in the previous section for the parallel 2-opt algorithm.

The parallel 3-opt algorithm requires the same computational effort as the sequential 3-opt algorithm, as the total number of 3-exchanges examined to verify local optimality for $\mathcal{D}_{3,B}$ and $\mathcal{N}_{3,B}$ are equal, viz. $\mathcal{O}(BN^2)$. Furthermore, at most N div $(B + 1)$ processors can be used in a parallel 3-opt algorithm for a given instance of size N and choice of parameter B in $\mathcal{D}_{3,B}$.

5.2.3 Computational results

The distributed neighborhoods $\mathcal{D}_2$ and $\mathcal{D}_{3,B}$ are mapped onto a message-passing MIMD machine with a ring network topology by assigning adjacent partial solutions to adjacent processors. A partial solution consists of two paths and the number of edges in a path determines the local neighborhood size. The number of edges in each of the two paths of a partial solution is not fixed because performing edge exchanges can change the number of edges in each of these paths. Since communication has to take place as soon as local neighborhoods have been explored, load imbalance may occur when path lengths are not equal.

To obtain a new distribution, each processor sends one edge to its left adjacent processor; to obtain the next distribution each processor sends one edge its right

P	$\epsilon(\%)$	T(s)	eff
1	7.5	144.2	1
16	7.8	13.0	0.69
64	7.4	5.2	0.43
128	7.6	5.5	0.20

P	$\epsilon(\%)$	T(s)	eff
1	7.9	1436	1
64	7.4	29	0.77
128	7.5	18	0.62
256	7.8	19	0.29

Table 5.1: Computational results for att532 and pr1002 with $\mathcal{D}_2$.

P	$\epsilon(\%)$	T(s)	eff
1	11.6	1960	1
64	10.8	52	0.59
128	10.8	31	0.49
256	11.5	32	0.24

P	$\epsilon(\%)$	T(s)	eff
16	5.4	481	1
64	5.2	103	1.17
128	4.6	69	0.87
256	5.3	76	0.40

Table 5.2: Computational results for rl1304 and d2103 with $\mathcal{D}_2$.

P	$\epsilon(\%)$	T(s)	eff
16	8.8	685	1
64	9.1	189	0.91
128	8.6	109	0.79
256	9.3	66	0.65

P	$\epsilon(\%)$	T(s)	eff
64	8.8	312	1
128	8.6	168	0.93
256	8.6	108	0.72
512	8.7	79	0.49

Table 5.3: Computational results for pr2392 and pcb3038 with $\mathcal{D}_2$.

P	$\epsilon(\%)$	T(s)	eff
128	9.0	645	1
256	9.3	315	1.02
512	8.6	215	0.75

P	$\epsilon(\%)$	T(s)	eff
128	8.7	2671	1
256	8.7	1396	0.96
512	9.2	819	0.82

Table 5.4: Computational results for rl5934 and rl11849 with $\mathcal{D}_2$.

P	$\epsilon(\%)$	T(s)	eff
1	3.4	706	1
25	3.5	51	0.55
50	3.3	21	0.67

P	$\epsilon(\%)$	T(s)	eff
1	3.9	9948	1
25	4.0	671	0.59
50	3.9	324	0.61

Table 5.5: Computational results for att532 and rl1002 with $\mathcal{D}_{3,3}$.

adjacent processor, such that the tour is rotated. The communication overhead for obtaining new distributions is $\mathcal{O}(1)$ when the orientation of a tour has not changed due to the effectuation of edge exchanges. This situation occurs during verification of local optimality. Another situation occurs if some path assigned to a processor consists of one edge. If the orientation of partial solutions has changed through exchanges and edges are sent and received simultaneously, an empty path may arise when the received edge is connected to the other subpath than the path from which one edge is sent. In this case this processor has to wait until it receives an edge before it sends this edge. So the communication overhead is determined by the length of the longest sequence of adjacent processors in the ring for which all minimum path lengths are equal to one. More formally, the time complexity of communication overhead is $\mathcal{O}(1+\max_{0\leq p,q<P \mid \forall_{p\leq i<q}\, l(i)=1}\, q-p)$, where $l(i)$ is the minimum length of a path assigned to processor i. If all paths of partial solutions consist of at least two edges, the communication overhead is constant since then it is always possible to send an edge to an adjacent processor and still maintaining a partial solution that consists of two paths. The worst case communication behavior occurs when all processors own a path with length one, but in practice this rarely happens.

The parallel 2-opt and 3-opt algorithms have been implemented on networks consisting of 50 and 512 T805 transputers configured in a ring. A tour is represented by a linked list, which makes it possible to effectuate proposed exchanges in constant time. Initial tours are constructed using the nearest neighbor heuristic [Lawler, Lenstra, Rinnooy Kan & Shmoys, 1985]. A nearest neighbor tour is constructed as follows. Start in an arbitrarily chosen initial city and repeatedly choose the unvisited city closest to the current city. Once all cities have been chosen, return to the initial city. We have tested the algorithms on instances with several thousands of cities that originate from the TSP library of Reinelt [1991]. The number in the name of an instance denotes the number of cities in this instance.

Tables 5.1 – 5.4 list the computational results obtained with the neighborhood $\mathcal{D}_2$. All results are averages computed over ten runs started from different initial solutions. In these tables, P is the number of processors, and the average relative excess, measured in percentages, over the minimal tour length or the best known lower bound is given by ϵ. The average running time in seconds is given by T. The efficiency, defined as the speed-up divided by P, is given in the column labeled "eff". For large instances we were not able to run the algorithm for $P=1$, due to the limited availability of running time so in these cases the efficiency is computed relatively to the smallest number of employed processors. The computational results in Tables 5.1 – 5.4 show that the number of processors has no influence on the average cost of final solutions. This is explained by the fact that

$\mathcal{D}_2$ is isomorphic with $\mathcal{N}_2$. Moreover, good efficiencies of more than 50 percent are obtained, if the partial solutions are sufficiently large. This results in speed-ups of up to a factor 80 with 128 processors.

Table 5.5 lists the results obtained with $\mathcal{D}_{3,B}$, for $B = 3$. Recall that the Or-opt neighborhood is included in the restricted 3-opt neighborhood $\mathcal{N}_{3,3}$ with which $\mathcal{D}_{3,3}$ is isomorphic. Again, it can be observed that the same quality final solutions are found regardless of the number of employed processors. This is explained by the isomorphism of $\mathcal{D}_{3,B}$ with $\mathcal{N}_{3,B}$. Moreover, good efficiencies are achieved using $\mathcal{D}_{3,B}$. The quality of final solutions obtained with $\mathcal{D}_{3,3}$ is much better than that obtained with the 2-opt neighborhood at the cost of a substantial increase in running time.

5.3 The Lin–Kernighan neighborhood

The most effective neighborhood for the TSP is the one proposed by Lin & Kernighan [1973] in which a variable number of edges is replaced instead of a fixed number of edges as done in the 2-opt or 3-opt neighborhoods. Using elaborated data structures and neighborhood reduction techniques, sophisticated sequential Lin–Kernighan algorithms require less running time than our parallel 2-opt and 3-opt algorithms and find better quality results. In the remainder of this chapter, we outline a parallel Lin–Kernighan algorithm that is competitive with the most advanced sequential implementations of the Lin–Kernighan neighborhood.

The idea behind the Lin–Kernighan neighborhood structure is that for any given tour the optimal tour can be obtained from it by exchanging the appropriate set of edges. Its exchange function τ_λ tries to construct this set by repeatedly removing and adding edges to the given tour. One property of this exchange function is that the number of edge exchanges is not fixed but depends on the given tour. Let $t \in \mathcal{S}$ and let $x_0 \in t$ be an edge in t. Then τ_λ can be outlined as follows:

(1) Remove edge x_0 from t. The result is a Hamiltonian path H_0, i.e., a path that visits each city only once. Set variable $i = 0$.

(2) Add an edge $y_i \notin t$ to the end of the Hamiltonian path H_i and remove an edge $x_{i+1} \in t$ from H_i, such that $H_{i+1} = t \backslash \{x_j \mid 0 \leq j \leq i+1\} \cup \{y_j \mid 0 \leq j < i + 1\}$ is a minimum-length Hamiltonian path. Edge x_{i+1} is uniquely determined by y_i. Increment i with one.

(3) Check if tour t', obtained by closing H_i, has lower cost than t^*, the best tour found so far. If this is the case, replace t^* with t'.

(4) Repeat steps (2)–(3) as long as a given *gain criterion* is satisfied.

(5) If no tour with lower cost than t is found, set $i = 1$ and repeat steps (2)–(4) while selecting a different edge y_1 that is added to H_1. If after a given

number of choices for y_1 no shorter tour has been found, set $i=0$ and repeat steps (2)–(4) for different edges y_0 that are added to H_0. Return t^*.

Let t_i be the tour obtained by closing H_i and let t_{i+1} be the tour obtained from H_{i+1}. Then t_{i+1} can be obtained from t_i by performing the 2-exchange on t that removes edges x_i and x_{i+1}. So the exchange function tries to construct a sequence of 2-exchanges that, performed on initial tour t, results in a shorter tour t^*. Every 2-exchange in this sequence removes an edge added by the preceding 2-exchange, viz., the edge added in step (3) to check if a shorter tour is constructed.

Let t^* be a tour with $f(t^*) < f(t)$. Then t^* can be obtained from t by adding the set of edges $t^* \setminus t$ to t and removing the set of edges $t \setminus t^*$ from t. Exchange function τ_λ tries to construct these sets by iteratively choosing one edge x_{i+1} that has to be removed and one edge y_i that has to be added, until all edges of both sets are chosen. The removal of x_{i+1} decreases the tour length and the addition of y_i increases the tour length. The net result of the exchange of edge pair (x_{i+1}, y_i) is called the *gain* and is defined by $|x_{i+1}| - |y_i|$, where $|e|$ denotes the length of an edge e. It is obvious that the cumulative gain of all pairs (x_i, y_i) should be positive, otherwise $f(t^*) \geq f(t)$. The gain criterion is based on this observation: new edge pairs are chosen as long as the cumulative gain of the pairs (x_{i+1}, y_i) is non-negative.

Let v be the last city of Hamiltonian path H_i. Then edge y_i, which is added to the end of H_i, is chosen from a predefined set of edges that depends on v. Edge y_i is chosen from this set such that the gain resulting from adding y_i and removing x_{i+1} to H_i, which is given by $|x_{i+1}| - |y_i|$, is maximized.

Furthermore, if no sequence of 2-exchanges can be constructed that gives a shorter tour, then for given edges x_0 and y_0 each 3-exchange that cannot be obtained by performing two consecutive 2-exchanges and that removes x_0 from t and adds y_0 is applied to t. If no 3-exchange results in a shorter tour, the backtrack mechanism in step (5) ensures a new Hamiltonian path H_1 is constructed by choosing a different y_0. Again, if no sequence of 2-exchanges can be constructed that gives a shorter tour, the above 3-exchange mechanism is applied. This is repeated until a shorter tour is found, or a given number of alternatives for y_0 and y_1 have been examined. If for all $x_0 \in t$ and all possible choices for y_0 and y_1, τ_λ cannot find a shorter tour, then the final solution is not only a local minimum with respect to $\mathcal{N}_\lambda$ but also with respect to $\mathcal{N}_3$. In practice only a restricted number of alternatives for y_0 and y_1 are examined. Using the exchange function τ_λ outlined above, we can define the neighborhood structure $\mathcal{N}_\lambda$.

Definition 5.7. Let $t \in \mathcal{S}$ be a tour. The Lin–Kernighan neighborhood structure $\mathcal{N}_\lambda$ is defined by $\mathcal{N}_\lambda(t) = \{\tau_\lambda(t, x_0) \mid x_0 \in t\}$. □

Although τ_λ returns a single tour t^*, several tours are examined during the construction of t^*. The Lin–Kernighan algorithm consists of an iterative improvement algorithm that uses the neighborhood structure $\mathcal{N}_\lambda$.

5.3.1 Neighborhood reduction

In the previous section we have outlined how the exchange function τ_λ constructs a neighbor of a given tour t by repeatedly inserting edges in a Hamiltonian path. Let y be an edge that is to be added to the Hamiltonian path that ends in city l. We want to consider only those edges $y = (l, m)$ that are likely to be in an optimal tour. So for each city l a set of cities, the *candidate set* for l, is to be determined such that the edges between l and these cities are likely to be in an optimal tour.

When edge x is removed, edge y is chosen from the candidate set such that $|x| - |y|$ is maximized. The time complexity for selecting y is $\mathcal{O}(\gamma)$, where γ is the size of the candidate set. This can be costly for large candidate sets. On the other hand, a candidate set that is too small may result in poor-quality solutions. In their original paper, Lin and Kernighan suggest for the candidate set of l the set of its five nearest neighbors. The disadvantage of such a set is that its size is fixed, and it does not exploit the geometrical structure of an instance. Reinelt [1992] suggests another approach based on a Voronoi diagram for a set of cities. A Delaunay graph is the geometric dual of a Voronoi diagram, and it connects cities that share a boundary in a Voronoi diagram. The following definition gives a partition of $\mathbb{R}^2$ into N convex polygons.

Definition 5.8. Let $d(c, c')$ denote the Euclidean distance between two points $c, c' \in \mathbb{R}^2$, and let $V = \{c_1, \ldots, c_N\}$ be a set of points in $\mathbb{R}^2$, with $N \geq 3$. Define for $c \in V$ the *Voronoi region* $vr(c, V)$ as

$$vr(c, V) = \{c'' \in \mathbb{R}^2 \mid \forall_{c' \in V \setminus \{c\}}\, d(c'', c) \leq d(c'', c')\}.$$

□

Then for all $c'' \in vr(c, V)$ no point in V is closer to c'' than c. The boundary that separates different Voronoi regions that have points in common is called a *Voronoi edge*. We define the Delaunay graph as follows.

Definition 5.9. Define the *Delaunay graph* G for a set of points V by

$$G = (V, \{\ (c, c') \in V^2 \mid c \neq c' \wedge |vr(c, V) \cap vr(c', V)| > 1\ \}).$$

□

A triangulation —a decomposition of a polygon into triangles— that contains the Delaunay graph can be computed with an algorithm from Fortune [1987]. We call this triangular graph the *extended Delaunay graph*. A triangulation contains at most $3N - 3$ edges, where N is the number of vertices of G. This means that the average number of adjacent cities of a given city is at most six in the extended Delaunay graph. The candidate set of a city consists of the cities that can be reached in k steps in the extended Delaunay graph, with $k \geq 1$. Such candidate sets are called *k-th order Delaunay sets*. We use first and second order Delaunay sets because higher order sets are too computationally expensive.

5.3.2 Efficient data structures

The following operations are applied to a tour or Hamiltonian path: *Pred*, *Succ*, *TwoChange*, *ThreeChange*, and *InBetween*. *Pred* and *Succ* return the predecessor and successor, respectively, of a given city in a Hamiltonian path. *TwoChange* reverses a part of a Hamiltonian path starting from a given city to the last city in the path. *ThreeChange* performs a 3-exchange on a Hamiltonian path and requires a boolean returned by *InBetween* that denotes whether a given city is located between two other cities in a path.

Performing a 2-exchange on a tour requires reversal of a part of the tour, which has a complexity of $\mathcal{O}(N)$ for tours of size N if the tour is represented by an array that lists the cities in tour order. This becomes quite expensive for the exchange function τ_λ that performs sequences of 2-exchanges on Hamiltonian paths. Fredman, Johnson, McGeoch & Ostheimer [1993] discuss three tour representations, two-level trees, segment trees, and splay trees that perform 2-exchanges more efficiently than the array data structure. We have implemented two-level trees and segment trees as splay trees catch up with these data structures only for instances with more than one million cities.

For each representation we explain how to implement operations *Succ* and *InBetween* and how to implement the reversal of a subpath in *TwoChange*. The implementation of *Pred* is analogous to that of *Succ*. *ThreeChange* is implemented as a sequence of reversals. There is one operation, we have not mentioned yet. In an execution of τ_λ backtracking may occur, so some of the constructed Hamiltonian paths have to be stored for later use. There are two approaches to store these paths. The easiest way is to copy the entire path, the other, more complicated, way is to reconstruct the required path from the current path by undoing the 2-exchanges that lead from the required path to the current path. For each representation, we describe how we store a path.

Array representation

The best-known data structure to represent a tour is the array representation. A tour of size N is represented by two N-sized one-dimensional arrays A and B. Array A lists the cities in the order as they occur in the tour, and array B is the inverse of A, i.e., $B\,(A\,(i)) = i$, for $1 \leq i \leq N$.

Using these two arrays the relative position a city has in a Hamiltonian path

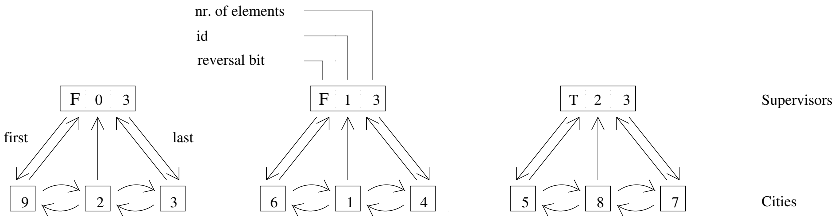

Figure 5.4: An example of a two-level tree.

is given in array B and therefore operations *Succ* and *InBetween* have complexity $\mathcal{O}(1)$. Reversing a part of a Hamiltonian path is done by reversing parts of arrays A and B, resulting in a $\mathcal{O}(N)$ complexity of operation *TwoChange*. Storing both arrays for backtracking by copying them is an $\mathcal{O}(N)$ operation.

Two-level tree representation

A 2-exchange reverses the direction in which a part of the Hamiltonian path is traversed. The idea of the two-level tree data structure is to partition the path in approximately equal-sized segments that contain consecutive cities, where each segment has a reversal bit that indicates the direction in which it should be traversed. Figure 5.4 shows an example of the two-level tree data structure. The segments are represented by doubly linked lists of cities, each connected to a *supervisor*. The supervisor contains the represented segment's reversal bit but also links to the first and last element of its doubly linked list. Furthermore, each supervisor includes the number of elements of its segment and it has a unique identification number to distinguish it from other supervisor nodes. The Hamiltonian path can be constructed by walking from segment to segment and traversing each segment according to its supervisor's reversal bit. The two-level tree in figure 5.4 represents the path $< 9, 2, 3, 6, 1, 4, 7, 8, 5 >$.

To determine the successor of a city, the city itself in the two-level tree has to be found. To speed up the search for a city, an array that contains the address of each city is maintained. If the city is located, the next city —or previous city, depending on the reversal bit of the supervisor— in the list contains the successor. If there is no next city, the consecutive supervisors must be checked until a non-empty segment is found. If there are no empty segments, it follows that operation *Succ* has complexity $\mathcal{O}(1)$.

Let the number of segments be $\lceil \frac{N}{g} \rceil$, where N is the number of cities and g is the *groupsize*, the approximate size of each segment, and let v_0, v_1 and v_2 be three cities on the represented Hamiltonian path. To find out whether v_1 is visited before v_2, if one starts from v_0 and follows the Hamiltonian path, the identification number of each city's supervisor is needed. If these three numbers are different we can determine if v_1 is in a segment between the segments that contain v_0 and v_2. If two or more numbers are the same, the corresponding segments have to be traversed. Since each segment contains approximately g elements, operation *InBetween* has worst-case complexity $\mathcal{O}(g)$.

Reversing a part of the path in *TwoChange* is implemented by reversing the order of some supervisors, flipping their reversal bit, and by relocating some cities to other segments. Due to the number of supervisors and the size of each segment this can be implemented in $\mathcal{O}(\max\{\lceil \frac{N}{g} \rceil, g\})$ time. If g is approximately $\sqrt{N}$, then *TwoChange* has complexity $\mathcal{O}(\sqrt{N})$. However, a 2-exchange can change the length of a segment. One might argue that to guarantee this complexity the segments should be rebalanced after each 2-exchange, but this results in a large overhead and is therefore not efficient. Moreover, the more or less random 2-exchanges can disturb but also restore the balance.

In order to enable backtracking we have to copy the required paths or reconstruct them from the current path. Copying a path takes $\mathcal{O}(N)$ time and reconstructing a path takes $\mathcal{O}(l\sqrt{N})$ time, where l is the number of 2-exchanges that lead from the current path to the required path. In practice l is much smaller than $\sqrt{N}$ and therefore we choose to reconstruct the required path.

Segment tree representation

The segment tree representation is based on the observation that it suffices to specify which segments of a path H need to be reversed to construct H' without explicitly reversing these segments. The following code fragment illustrates this idea. Here H_0, H_1 and H_2 are Hamiltonian paths derived from a tour t with length 100. $h[i]$ is the city at the i-th position in H_0, and *TwoChange*(c, H_0) is the Hamiltonian path that is obtained when the path in H_0 from c to the last city in H_0 is reversed. $a + b$ denotes the concatenation of paths a and b.

```
{ $H_0 = h[1, \dots, 100]$}
  $H_1$ :=TwoChange ($h[75]$, $H_0$);
{ $H_1 = h[1, \dots, 75] + h[100, \dots, 76]$}
  $H_2$ :=TwoChange ($h[25]$, $H_1$);
{ $H_2 = h[1, \dots, 25] + h[76, \dots, 100] + h[75, \dots, 26]$}
  $H_3$ :=TwoChange ($h[35]$, $H_2$);
{ $H_3 = h[1, \dots, 25] + h[76, \dots, 100] + h[75, \dots, 36] + h[26, \dots, 35]$}
```

The *permanent* tour is the representation of the current solution in a Lin–Kernighan algorithm. *Temporary* Hamiltonian paths are Hamiltonian paths constructed

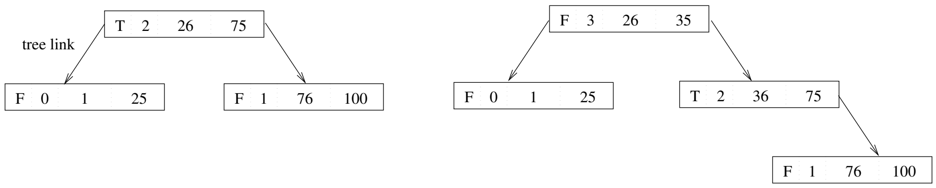

Figure 5.5: The segment trees for H_2 (left) and H_3 (right).

in a λ-exchange. In the above example *t* is the permanent tour, and H_0, H_1, H_2, H_3 are temporary paths. Temporary paths are represented by a sequence of records of the form [*rev*, *begin*, *end*] where *rev* is a boolean and *begin* and *end* are elements of $\{1, \ldots, N\}$ for tours of size N. Such a record, called a *segment node*, represents the subpath of the permanent tour from position *begin* to position *end*, when *rev* does not hold, or its reverse when *rev* holds. The segment nodes are stored in an array in the order in which the subpaths they represent occur in the temporary path.

The position a city has in the permanent tour is needed to find its successor. Given this position, the segment node that contains the city can be found. To accelerate the search for a city in the temporary path, a tree is constructed that contains pointers to all the segment nodes. The inorder traversal of this tree lists the segments in the order in which they occur in H_0. The tree structures for paths H_2 and H_3 are shown in Figure 5.5. We implement the tree as a height-balanced tree [Bayer, 1972]. Search and insert operations on these trees have complexity $\mathcal{O}(\log i)$, where i is the number of elements of the tree. The tree structure grows with the number of 2-exchanges performed on the permanent tour, which results in more expensive search operations. The example at the beginning of this section shows that with each 2-exchange at most one new segment node is created. If H' is the result of performing *l* consecutive 2-exchanges on a Hamiltonian path H, the segment tree that represents H' contains *l* elements. Because the tree search operations can be performed in log *l* time, operations *Succ* and *InBetween* have complexity $\mathcal{O}(\log l)$. Performing a 2-exchange involves the creation of a new segment node and the reversal of the order of other segment nodes. Since there are $\mathcal{O}(l)$ segments, operation *TwoChange* has complexity $\mathcal{O}(l)$.

The Hamiltonian paths H_0 and H_1 have to be stored in case backtracking takes place. For the segment tree representation storage of these paths can be done in $\mathcal{O}(1)$ time by copying the array and tree of segment nodes. The permanent tour is represented using the array representation, because in each call to each operation, the position a city has in the permanent tour is needed. Using the array representation the position of a city can be determined in $\mathcal{O}(1)$ time.

representation	*Succ*	*TwoChange*	*BackUp*	*Effectuate*
array	$\mathcal{O}(1)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	-
two-level tree	$\mathcal{O}(\sqrt{N})$	$\mathcal{O}(\sqrt{N})$	$\mathcal{O}(k\sqrt{N})$	-
segment tree	$\mathcal{O}(\log k)$	$\mathcal{O}(k)$	$\mathcal{O}(1)$	$\mathcal{O}(N)$

Table 5.6: Complexity of tour operations for tours of size N.

Since the path operations become more expensive as the size of the segment tree increases, we update the permanent tour and reduce the segment tree to a tree that contains a single segment node as soon as a cost-improving λ-exchange has been found. This is done by effectuating the sequence of 2-exchanges that leads to the tour t^*, which takes $\mathcal{O}(lN)$ time, where l is the length of the sequence, or by reading the segments consecutively, which takes $\Theta(N)$ time. Which method is more efficient depends on the contents of the individual segments nodes.

Table 5.6 summarizes the cost of each tour operation for the data structures to represent a tour. For each tour representation the cost to perform operation *Pred* is equal that of operation *Succ*. It should be noted that for the segment tree tour representation, the cost to perform a tour operation on a Hamiltonian path H_i depends on i, since the height of a segment tree is equal to $\log i$. Therefore, an upper bound k is often imposed on the maximum number of attempted 2-exchanges that are represented in a segment tree. In this way, the maximum height of the segment tree is bounded by $\log k$.

5.3.3 Computational complexity

Next, we analyze the total time complexity of all tour operations that are performed in the exchange function τ_λ. The number of times each tour operation is performed in τ_λ depends on the given tour. Therefore, we define the following sequence of tour operations that is performed if τ_λ is called with a Hamiltonian path H_0 that is obtained by removing edge x_0 from a tour.

(1) A Hamiltonian path H_1 is constructed by adding an edge to and removing an edge from H_0. One *Succ* operation is performed for each element of the candidate set of the last city of H_0 to find the edge that should be added. On average, γ *Succ* operations are performed, where γ denotes the average candidate set size.

(2) Once an edge is found, operation *TwoChange* is performed once and operation *Pred* twice.

(3) Steps (1)–(2) are repeated κ times to construct a sequence of Hamiltonian paths H_i with $0 \leq i < \kappa < k$.

representation	complexity
array	$\mathcal{O}(\kappa\gamma + \kappa N)$
two-level tree	$\mathcal{O}(\kappa\gamma\sqrt{N})$
segment tree	$\mathcal{O}(\kappa\gamma \log\kappa + \kappa^2)$

Table 5.7: Total complexity of tour operations in τ_λ.

In all, operation *Succ* is performed at most $\kappa\gamma$ times, *TwoChange* at most κ times and *Pred* at most 2κ times. Furthermore, Hamiltonian paths H_0 and H_1 have to be stored in case backtracking has to take place.

Table 5.7 presents for each tour representation the total time complexity of tour operations performed in the exchange function τ_λ to construct a sequence of κ 2-exchanges. N denotes the number of cities and γ the size of the candidate sets. Note that for large candidate set sizes γ, e.g. $\gamma \approx N$, the total complexity of tour operations for the array representation is $\mathcal{O}(\kappa N)$, for the two-level trees $\mathcal{O}(\kappa N\sqrt{N})$, and for the segment trees $\mathcal{O}(N\kappa \log\kappa)$. This suggests that the two-level trees and segment trees are not suited for large candidate sets.

For the Delaunay sets defined in Section 5.3.1, the average candidate set size γ is at most six. In that case the complexity of the array, two-level tree and segment tree representation reduces to $\mathcal{O}(\kappa N)$, $\mathcal{O}(\kappa\sqrt{N})$ and $\mathcal{O}(\kappa^2)$, respectively. Experiments show that in general κ is much smaller than N, so the larger N is, the more efficient the segment tree representation is compared to the array and two-level tree representations. It should be noted, however, that we have not added the costs of effectuating the proposed 2-exchanges for the segment tree representation, which takes $\mathcal{O}(N)$ time. If only a few 2-exchanges are needed in τ_λ to find a shorter tour, the given complexities suggest that the two-level tree representation is more efficient than the segment tree representation. However, if large amounts of 2-exchanges are evaluated that do not result in shorter tours, the complexity of τ_λ is dominated by the cost to evaluate these 2-exchanges, which is more expensive for two-level trees than for segment trees.

5.3.4 Computational results

We have implemented the sequential Lin–Kernighan algorithm on a SUN Sparc ELC workstation with 16 Mbyte of memory. There are a few differences between the algorithm of Lin & Kernighan [1973] and our implementation of it. First of all, we consider at most five edges for y_0 and y_1, the edges added to H_0 and H_1, for backtracking. This is motivated by the observation that, if any gain is found by inserting an edge, it is usually one of the first choices for these edges, due to the way y_0 and y_1 are chosen. We limit the length of the sequence of 2-exchanges in τ_λ to k, where $k = 50$. For the segment tree representation, each operation becomes more expensive when the sequence of performed 2-exchanges becomes larger. Fredman, Johnson, McGeoch & Ostheimer [1993] mention that a limit of 50 on the length of this sequence does not influence the final tour length.

We used the nearest neighbor heuristic to construct the initial tour. We have tested our Lin–Kernighan algorithm on several instances from Reinelt's TSP library. Table 5.8 presents the results. The items in the tables are averages over ten runs of the algorithm starting from different initial tours. The first column lists the instances and their sizes. The columns labeled first and second order contain the computational results obtained with the first and second order Delaunay sets defined in Section 5.3.1. We have tested the tour representations with both candidate sets. The columns labeled "a", "t" and "s" contain the average running times in seconds obtained with arrays, two-level trees, and segment trees, respectively. The column labeled ϵ gives the average relative excess in percentages over the optimal solution or, when the optimum is not known, over the best-known lower bound.

We observe that the relative difference in running times between the representations becomes smaller as the size of the candidate set increases. This can be explained by the observation that for larger candidate sets, the total complexity of all tour operations is increasingly determined by the number of *Succ* operations. This operation is more expensive for the two-level trees and in particular for segment trees than for arrays.

Furthermore, we observe that segment trees outperform two-level trees, and that two-level trees outperform arrays. In the previous section we have shown that for small candidate sets, such as the first and second order Delaunay sets, the running time of τ_λ is dominated by the complexity of a 2-exchange. So for larger instances the differences between running times becomes larger, provided that only a small proportion of the examined 2-exchanges leads to a shorter tour, because segment trees have to be consolidated— that is, proposed 2-exchanges have to be effectuated whenever a shorter tour is found. Apparently this is the case because the segment trees outperform the two-level trees for larger instances.

We have also tested five instances with complete candidate sets consisting of all cities. For these five instances, not listed in Table 5.8, the best result is obtained once with first order Delaunay sets, three times with second order Delaunay sets and only once with complete candidate sets. For all other instances we have tested, we obtained better solutions with second order Delaunay sets than with first order Delaunay sets. For most instances we obtained slightly better results with third order Delaunay sets than with second order Delaunay sets. This small improvement in quality is paid for with a large increase in running time.

Another interesting observation is that the relative difference in running times

cand. set	first order				second order			
repr.	a	t	s		a	t	s	
	$T(s)$			ϵ	$T(s)$			ϵ
rd400	15.9	6.7	3.5	1.82	27.5	12.6	10.1	1.48
pcb442	10.0	4.0	2.9	1.42	17.6	9.1	8.5	1.13
u574	34.2	8.4	6.0	2.22	59.5	22.0	19.0	1.90
p654	23.3	11.7	6.9	0.46	45.7	31.8	25.2	1.01
rat783	48.5	17.5	6.5	2.01	75.2	33.7	17.2	1.71
pr1002	121	44	13	2.42	178	74	35	2.45
u1060	132	29	15	2.13	204	82	46	1.89
pcb1173	143	36	15	2.86	167	66	31	1.97
d1291	72	20	9	4.08	142	69	49	3.75
u1432	90	22	12	2.30	131	46	34	2.12
d1655	193	52	19	3.63	274	105	57	3.05
rl1889	297	51	20	2.87	438	123	78	2.56
d2103	132	41	13	3.30	185	91	44	3.26
u2152	221	71	18	3.12	345	110	56	2.61
pr2392	732	226	39	2.88	1003	284	95	2.04
pcb3038	1181	305	50	2.22	1491	418	111	1.87
fl3795	459	169	27	4.92	841	446	131	4.07
fnl4461	2741	654	62	1.88	4578	918	189	1.62
rl5915	2822	277	69	3.05	3201	547	260	2.61
rl5934	2422	254	57	3.29	3764	660	258	2.97
pla7397	3381	1148	105	3.22	3607	1475	227	2.49
rl11849	14562	1282	181	2.83	21411	2195	950	2.30
brd14051	–	10092	374	2.79	–	8433	663	2.72
d18512	–	23827	475	2.34	–	19401	1005	2.17
pla33810	–	12849	810	2.58	–	21748	1656	2.42
pla85900	–	63831	2795	2.94	–	–	6261	2.74

Table 5.8: Computational results for the sequential Lin–Kernighan algorithm.

between our implementation of segment trees and two-level trees is much more significant than the difference observed by Fredman, Johnson, McGeoch & Ostheimer [1993]. This might be explained by the fact that they do not balance the segment trees. As a consequence, the height of a segment tree obtained after i 2-exchanges is bounded by $2i+1$ in their implementation, whereas it is bounded by $\log i$ in our implementation. This height determines the complexity of *Succ* and *Pred* operations.

5.4 A parallel Lin–Kernighan algorithm

In Section 5.2 we have presented parallel 2-opt and 3-opt algorithms based on decomposing a solution into a number of disjoint partial solutions consisting of two separate paths that are assigned to different processors. Subsequently, all partial solutions can be changed independently from each other since edge exchanges only affect the edges involved in a given partial solution. However, initial experiments showed that such a solution decomposition is not effective for the Lin–Kernighan neighborhood. Although a larger number of λ-exchanges is accepted, the total cumulative gain in all these simultaneously performed exchanges is equal to, or less than, the gain made in a single λ-exchange applied to the entire tour. So it is too restrictive to limit the edges that can be changed in τ_λ beforehand. Therefore, we outline a different approach in the remainder of this section.

In an algorithm with multiple-step parallelism several consecutive steps in the neighborhood graph are made in parallel. Let P denote the number of processors. A Lin–Kernighan algorithm with multiple-step parallelism based on domain decomposition can then be outlined as follows.

(1) Partition the domain $\{(t, x_0) \mid x_0 \in t\}$ of the exchange function τ_λ applied to a tour $t \in \mathcal{S}$ into P subdomains. Each processor evaluates one subdomain and proposes at most one profitable exchange.

(2) Communicate profitable exchanges to other processors.

(3) Combine and effectuate a subset of the profitable exchanges found in step (1), which results in a new tour t'.

(4) Replace t by t', and continue steps (1)–(3) until no improvement is found.

A single iteration of our algorithm consists of three phases—that is, evaluating exchanges in step (1), communicating profitable exchanges in step (2), and effectuating proposed exchanges in step (3). First, each processor evaluates the exchanges that are assigned to it. Each processor has to evaluate at most $\lceil \frac{N}{P} \rceil$ exchanges, where N is the number of cities and P the number of processors. Each processor proposes at most one profitable exchange. As soon as a processor has found a profitable exchange or it has evaluated all its exchanges, it has to wait until other processors have finished evaluating their exchanges. Global communication then takes place to broadcast all proposed exchanges to all processors. Subsequently, a subset of proposed exchanges has to be effectuated such that a feasible solution is constructed. Since all processors have knowledge of all proposed exchanges, combination of proposed exchanges can be done locally and results in the same tour on all processors.

5.4.1 A distributed Lin–Kernighan neighborhood

Recall that a distributed neighborhood consists of a distribution structure, a domain distribution, and a combination function. Next, we discuss these components for the distributed Lin–Kernighan neighborhood.

Domain distribution. Our distributed Lin–Kernighan neighborhood uses a distribution structure in which the entire current tour is assigned to each processor. The domain distribution for the exchange function τ_λ is also easy to formulate since each application of τ_λ on a tour t requires only a single additional argument, an edge x_0 of t. The set of N arguments of τ_λ in a neighborhood $\mathcal{N}_\lambda(t)$ therefore consists of all pairs (t, x_0) for $x_0 \in t$. We can partition this set of arguments into P (almost) equal-sized subsets. Subsequently, each subset is assigned to a different processor p, and each processor executes τ_λ with the arguments (t, x_0) chosen from the subset assigned to p to construct its local neighborhood. A distributed Lin–Kernighan neighborhood that uses the above domain distribution is isomorphic with the conventional Lin–Kernighan neighborhood as this domain distribution partitions this neighborhood.

Combination function. Since each processor executes τ_λ on the entire tour t and without any restriction on edges that may be exchanged, it can occur that the same edges are replaced or inserted by different processors. This may result in infeasible solutions when all proposed edge exchanges are effectuated. Even if proposed edge exchanges are disjoint, it can occur that a tour is split into several subtours when all exchanges are effectuated. Hence, the combination function can only effectuate a subset of the proposed exchanges. Essential to obtain a good speed-up is that the total gain made in each iteration of the parallel algorithm is as large as possible, i.e., the gain achieved with effectuating a subset of all proposed exchanges should be maximal. This problem, which we call the traveling salesman combination problem (TSCP), can be formulated as follows.

Definition 5.10 (TSCP). Given are a TSP instance, a tour t, and a collection $L = \{l_1, \ldots, l_q\}$ of q edge sets included in t. Let $l'_1, \ldots, l'_q$ be sets of edges such that $t \setminus l_i \cup l'_i \in \mathcal{N}_\lambda(t)$ for $1 \leq i \leq q$, and let $L' = \{l_1, \ldots, l_q, l'_1, \ldots, l'_q\}$. Then, the problem is to find a subset L^* of L' such that $t^* = (t \setminus \bigcup_{1 \leq j \leq q} l_j) \cup (\bigcup_{k \in L^*} k)$ is a tour, and $f(t^*)$ is minimal. □

In the TSCP one is asked to find a subset L^* of the proposed exchanges L such that the tour obtained by replacing the edges in L with the edges in L^* has minimal length. Unfortunately, we have the following result.

Theorem 5.3. *The decision variant of the TSCP is NP-complete.*
Proof. It is not hard to verify that the TSCP is in NP. Next, we reduce the set packing problem (SPP) to the TSCP. In the SPP a collection C of finite sets with elements from a set W is given and a positive integer $m \leq |C|$. The question is whether C contains at least m mutually disjoint sets. The SPP is NP-complete [Garey & Johnson, 1979].

We construct a tour t, with $|t| = |W| + |C|$, as follows. Map each $w \in W$ onto an edge e of a tour t with length $d(e) = 0$. Associate with each set $c \in C$ an unique edge $e_c \in t$ on which no $w \in W$ is mapped, such that c, whose elements are mapped onto a set of edges E_c, is mapped onto a tour $t'_c \in \mathcal{N}_\lambda(t)$ with $t \setminus t'_c = E_c \cup \{e_c\}$, i.e., edges $l_c = E_c \cup \{e_c\}$ are removed from t to construct t'_c. τ_λ adds edges e' to t'_c such that $(t'_c \setminus t) \cap (t'_{c'} \setminus t) = \emptyset$ for all $c' \in C \setminus \{c\}$. Choose $d(e_c) = 1$ and $d(e') = 0$ for all $e' \in t'_c \setminus t$ and $c \in C$. Hence, $f(t) - f(t'_c) = 1$ for all t'_c associated with sets $c \in C$. An SPP instance is now reduced to an TSCP instance in which the question is whether there exists a tour t^* with $f(t) - f(t^*) \geq m$, because for all $l_c, l_{c'} \in L$ that are included in t but not in t^* holds that $l_c \cap l_{c'} = \emptyset$, otherwise t^* is not a tour since the inserted edges in t'_c and $t'_{c'}$ are different. Consequently, it holds that the sets c associated with the edge sets sets $l_c \in L^*$ are mutually disjoint. So the problem of determining whether C contains m mutually disjoint subsets is equivalent with determining whether there exists a tour t^* with $f(t) - f(t^*) \geq m$ since the gain made by exchanging an edge set $l \in L^*$ is at most one. □

Considering that the problem of finding an optimal subset of the proposed exchanges whose effectuation results in a maximal gain is NP-hard, we use the following heuristic for selecting a subset of the proposed exchanges that is effectuated. The idea is to combine proposed edge reversals to a feasible tour by attempting to perform them on the current tour in order of descending gain. For this, we construct the following sequence of tours t_p with $0 \leq p \leq P$ and $t_0 = t$. Let e_p be an edge for which $f(\tau_\lambda(t, e_p)) < f(t)$. Then $t_{p+1} = t_p$ if $\tau_\lambda(t_p, e_p)$ does not result in a feasible tour with lower cost, otherwise $t_{p+1} = \tau_\lambda(t_p, e_p)$. A proposed λ-exchange consists of a sequence of k 2-exchanges. To compute t_{p+1} this sequence of 2-exchanges is performed on t_p. If a 2-exchange in the sequence of $\tau_\lambda(t, e_p)$ removes an edge x which does not occur in t_p, i.e., this edge has already been removed by a previous λ-exchange then $\tau_\lambda(t_p, e_p)$ cannot be effectuated, so $t_{p+1} = t_p$. The part of the sequence of 2-exchanges proposed in $\tau_\lambda(t, e_p)$ up to the 2-exchange that can no longer be performed can still be effectuated, provided that it gives a shorter tour t_{p+1}.

5.4.2 Computational complexity

The complexity of an iteration of the parallel Lin–Kernighan algorithm depends on the target-machine on which the algorithm is executed. Our target-machine is a message-passing MIMD machine with a 2-dimensional torus as interconnection network. We restrict the complexity discussion to this machine.

Complexity of the proposal phase. The complexity of the proposal phase depends on the representation of a tour. It is at most N/P times the complexity of τ_λ as given in Table 5.7, where N is the number of cities and P the number of processors. After this phase a processor can be idle for some time, until all other processors have completed the proposal phase, at which point the communication phase begins. The idle time should of course be minimal, and therefore a good load balance is important. The load imbalance is determined by the time between the first and the last processor to finish the evaluation of exchanges. This amount of time is to be minimized. It is, however, not possible to determine an equal amount of work for each processor beforehand, because the time needed to evaluate exchanges depends on the (variable) number of edges replaced in an exchange. Fortunately, the maximum difference between the first and the last processor to finish the proposal phase decreases when P increases, because then the maximum number of exchanges evaluated by processors decreases. Consequently, also the load imbalance decreases with increasing P.

Communication overhead. As soon as all processors have stopped evaluating exchanges, all-to-all broadcasting of proposed exchanges occurs. This part of the algorithm is called the communication phase. It makes no difference for the communication overhead whether communication is synchronous or asynchronous, because even if a processor can send a message to another processor before this processor wants to receive it, a processor can only proceed as soon as it has received the exchanges proposed by all other processors. Using the algorithm of Section 6.2, all-to-all broadcasting on a two-dimensional $2x \times 2y$ torus can be done in $\mathcal{O}(x + y)$ time, which is equal to $\mathcal{O}(\sqrt{P})$ if the network topology is a $\sqrt{P} \times \sqrt{P}$ torus, where P is the number of processors.

The parallel algorithm performs multiple exchanges in a single iteration. Consequently, the total number of all-to-all broadcasts in a run of the algorithm decreases with an increasing number of processors, provided that the total number of exchanges needed to find local minima is independent of the number of employed processors. So the time needed for all-to-all broadcasting increases sublinearly with an increasing number of processors, but the total number of all-to-all broadcasts decreases with an increasing number of processors.

Complexity of the combination function. The complexity of effectuating proposed exchanges by the combination function depends on the data structure used to represent tours. Assume that a proposed λ-exchange consists of a sequence of k 2-exchanges in which the edges x_i are replaced by the edges y_i for $0 \leq i \leq k$. Effectuating this proposed exchange is then similar to applying τ_λ to the tour t_q in which the edges x_i that are to be removed by τ_λ are known beforehand; here t_q is the tour obtained by the combination function through effectuating the preceding q proposed exchanges. Since x_i is already given, only a single *Succ* operation has to be performed to check whether x_i still exists in the tour t_q. So for the two-level tree representation the complexity to effectuate a proposed exchange is $\mathcal{O}(k\sqrt{N})$, and the total complexity of the combination function to effectuate all proposed exchanges is $\mathcal{O}(Pk\sqrt{N})$, where k is the maximum number of 2-exchanges in any proposed λ-exchange.

For the segment trees representation effectuating a proposed λ-exchange on the permanent tour requires $\mathcal{O}(N)$ time. So if proposed exchanges are effectuated each time after checking whether a proposed exchange can be performed, the combination function requires at least $\Omega(PN)$ time for effectuating P exchanges. However, it is possible to postpone the effectuation of exchanges on the permanent tour until all proposed exchanges have been checked and implemented in the segment tree. This implies that a single, possibly large, segment tree is built that represents all proposed exchanges. If K denotes the total number of 2-exchanges summed over all proposed λ-exchanges, then the height of this segment tree is $\mathcal{O}(\log K)$. Since only a single *Succ* operation has to be performed in each 2-exchange, the total complexity to build a segment tree that represents the tour obtained after effectuating all proposed exchanges, is $\mathcal{O}(K \log K + K^2)$. Effectuation of the proposed exchanges on the permanent tour requires $\mathcal{O}(N)$ time. The total complexity of the combination function is therefore $\mathcal{O}(N + K \log K + K^2)$. Recall that in each λ-exchange an upper bound on the number of 2-exchanges is imposed, which is equal to 50 in our implementation. Hence, $K = \mathcal{O}(P)$ and the complexity of the combination function is $\mathcal{O}(N + P^2)$. The combination function effectuates at most P exchanges, which requires $\mathcal{O}(PN)$ time in a sequential algorithm. So the total amount of time needed for effectuating proposed exchanges in the parallel Lin–Kernighan algorithm that uses segment trees is less than that required by a sequential algorithm.

Important issues to obtain a good speed-up with our parallel Lin–Kernighan algorithm are load balancing, communication overhead, and the proportion of proposed exchanges effectuated by the combination function. The load imbalance is expected to decrease with an increasing number of processors. The time needed for all-to-all broadcasting is $\mathcal{O}(\sqrt{P})$, which grows sublinearly with increasing number of processors, but the number of all-to-all broadcasts is expected to decrease with increasing number of processors. The number of proposed exchanges that can be effectuated depends on the tour at hand but is bounded by N/k, where k is a lower bound for the number of replaced edges in a single λ-exchange. In our parallel algorithm only the work involved in a rejected exchange does not contribute to the speed-up, and it is therefore important that most of the proposed exchanges are effectuated. It is likely that more exchanges are rejected when the number of processors increases. For a fixed number of processors it is likely that less exchanges are rejected when larger instances are attempted.

cand. set	first order					second order				
P	1	16	32	48	64	1	16	32	48	64
	$T(s)$	speed-up				$T(s)$	speed-up			
d2103	360	3.3	4.9	6.2	4.9	926	4.7	4.8	7.5	7.7
u2152	517	5.5	9.8	9.5	10.0	1053	5.2	9.0	11.7	11.5
pr2392	1317	5.0	8.8	13.6	16.2	2140	5.6	10.1	12.5	15.7
pcb3038	2091	6.0	12.1	14.3	16.8	2699	6.7	11.0	15.4	16.9
rl5915	2241	4.5	7.7	11.2	12.4	9498	8.1	11.9	15.6	16.8
rl5934	2267	4.1	7.9	10.6	12.0	9772	5.7	10.4	13.0	18.4
pla7397	9041	5.7	8.9	13.7	14.9	14987	5.1	9.9	14.5	18.2

Table 5.9: Computational results on the Parsytec GCel for two-level trees.

cand. set	first order					second order				
P	1	16	32	48	64	1	16	32	48	64
	$T(s)$	speed-up				$T(s)$	speed-up			
d2103	70	2.8	4.0	4.5	3.3	400	3.7	3.8	4.5	4.6
u2152	131	4.3	5.7	5.3	6.2	421	4.2	6.1	7.3	7.5
pr2392	252	5.0	6.9	8.6	8.9	671	5.2	7.6	8.8	9.9
pcb3038	356	5.3	8.7	9.2	9.3	810	5.7	8.5	10.2	11.0
rl5915	448	4.9	7.6	9.2	8.8	2197	5.3	7.6	9.1	9.4
rl5934	439	4.5	6.9	8.3	8.4	2321	4.8	6.3	8.5	9.8
pla7397	782	5.2	6.7	8.9	8.8	2026	5.0	7.0	8.5	10.0
rl11849	1352	5.3	8.1	9.7	9.9	4504	5.5	6.7	10.7	11.6
brd14051	2409	6.5	10.2	12.0	13.7	5427	7.2	11.5	13.9	16.4

Table 5.10: Computational results on the Parsytec GCel for segment trees.

cand. set	first order					second order				
P	1	8	16	24	32	1	8	16	24	32
	$T(s)$	speed-up				$T(s)$	speed-up			
rl5915	140	2.5	4.4	5.5	6.8	509	3.2	5.7	6.6	9.8
rl5934	200	3.1	5.5	7.1	8.6	612	3.0	5.6	9.4	8.7
pla7397	815	3.9	6.7	10.5	10.9	904	2.9	4.9	6.5	9.7
rl11849	1645	3.4	7.8	8.8	12.0	3027	3.5	6.6	10.3	13.3
brd14051	2042	1.6	3.6	5.9	7.2	4317	2.2	4.7	8.1	12.8
d18512	6719	3.4	7.8	9.7	11.3	8553	4.1	6.9	12.3	14.4

Table 5.11: Computational results on the Parsytec Xplorer for two-level trees.

cand. set	first order					second order				
P	1	8	16	24	32	1	8	16	24	32
	$T(s)$	speed-up				$T(s)$	speed-up			
rl5915	39	3.3	4.6	5.3	6.1	175	3.6	5.2	6.2	7.8
rl5934	41	3.2	4.7	5.4	6.3	178	3.1	4.8	6.8	6.8
pla7397	57	2.8	4.1	5.4	5.3	146	3.2	4.6	5.6	7.0
rl11849	132	3.7	5.5	6.4	7.6	373	3.3	5.2	6.3	7.9
brd14051	206	4.0	6.0	8.6	9.8	461	4.4	7.0	9.0	10.9
d18512	278	3.8	6.8	8.0	9.7	552	4.1	6.8	9.9	10.8
pla33810	443	2.7	4.3	5.3	6.6	982	3.6	4.9	6.1	8.1
pla85900	1628	2.9	3.9	5.2	6.0	3852	3.1	5.9	6.4	7.7

Table 5.12: Computational results on the Parsytec Xplorer for segment trees.

P	1	8	16	24	32	1	8	16	24	32
	$T(s)$					$\epsilon_{\text{avg}}(\%)$				
rl5915	39	12	8.4	7.3	6.4	3.3	2.7	2.8	2.9	2.9
rl5934	41	13	8.7	7.5	6.7	3.3	3.1	3.5	3.3	3.2
pla7397	57	20	14	11	11	3.0	3.0	2.7	2.9	3.2
rl11849	132	36	24	21	18	3.0	2.7	2.6	2.5	2.8
brd14051	206	51	34	24	21	2.9	2.7	2.8	2.7	2.7
d18512	278	74	41	34	29	2.4	2.3	2.3	2.3	2.2
pla33810	442	163	104	84	67	2.7	2.6	2.6	2.5	2.6
pla85900	1628	563	431	311	273	2.9	2.7	2.7	2.6	2.6

Table 5.13: Running times and average relative excess ϵ for segment trees and first order Delaunay sets.

P	1	8	16	24	32	1	8	16	24	32
	$T(s)$					$\epsilon_{\text{avg}}(\%)$				
rl5915	174	48	34	28	23	2.6	2.3	2.3	2.3	2.6
rl5934	178	46	32	26	21	3.0	2.8	2.9	2.8	2.7
pla7397	143	68	37	29	28	2.6	2.4	2.8	2.5	2.5
rl11849	373	113	72	60	47	2.4	2.4	2.3	2.4	2.4
brd14051	461	106	66	51	42	2.6	2.5	2.6	2.5	2.6
d18512	552	137	81	56	51	2.2	2.2	2.1	2.2	2.1
pla33810	982	272	200	160	121	2.4	2.5	2.3	2.5	2.5
pla85900	3852	1244	658	601	500	2.7	2.7	2.9	2.6	2.6

Table 5.14: Running times and average relative excess ϵ for segment trees and second order Delaunay sets.

5.4.3 Computational results

We have implemented the parallel Lin–Kernighan algorithm in C on two parallel machines running the Parix operating system. The first one is a Parsytec GCel that consists of 512 T805 transputers with 4 Mbyte local external memory. The second one is a Parsytec PowerXplorer that consists of 32 processing units that are based on the PowerPC 601 microprocessor. Each processing unit has 4 bidirectional communication link interfaces and a local external memory of 32 Mb. Both machines are configured in a two-dimensional torus. Although the computational power of a T805 transputer is much less than that of a PowerPC processor —a T805 is roughly 15 times slower—, the ratio of communication vs. computation time is better for the GCel, which means that communication is relatively less costly on the GCel than on the PowerXplorer. A comparison on these two platforms may therefore indicate the dependence of our algorithm on the communication/computation performance ratio.

We have tested the parallel algorithm on the same instances as the sequential algorithm. All running times are given in seconds. Tables 5.9 and 5.10 give the speed-up obtained on the Parsytec GCel. The largest instance we could test on this network was pla7397 because of memory limitations. Tables 5.11 and 5.12 give the speed-up obtained on the PowerXplorer network. The results are averages over ten runs of the algorithm starting from different initial nearest neighbor tours. Each table presents the average running times for the runs on a single processor and the average speed-up for the runs on multiple processors.

From these tables we observe that larger and better scaling speed-ups can be obtained for the two-level tree data structure than for the segment tree data structure. However, the achieved speed-ups for two-level trees are not sufficient to make them competitive with segment trees, because the smallest running times are evidently obtained with the segment tree data structure.

Furthermore, we observe that in most cases better speed-ups are obtained on the GCel. This is explained by the better communication vs. computation ratio of this machine. The smallest overall running times, however, are clearly obtained on the PowerXplorer, because of the much larger computational power of this machine. Also, we observe that better speed-ups are obtained when using larger candidate sets, such as those consisting of second order Delaunay sets. This is explained by the increasing computation time through using larger candidate sets, which is beneficial for the speed-up of the algorithm.

Tables 5.13 and 5.14 show that the quality of the solutions found by the parallel algorithm is equal to that obtained by the sequential algorithm. This is no surprise, since the distributed Lin–Kernighan neighborhood used by the parallel algorithm is isomorphic with the conventional Lin–Kernighan neighborhood.

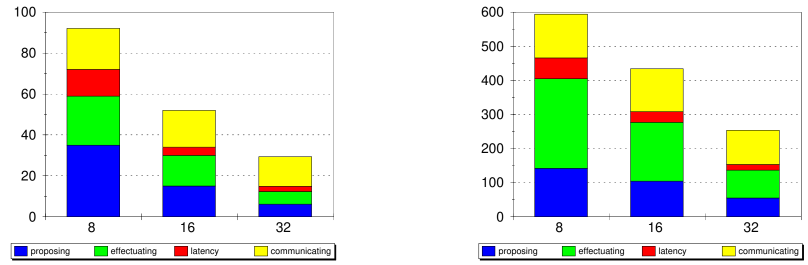

Figure 5.6: Running time profiles for d18512 (left) and pla85900 (right).

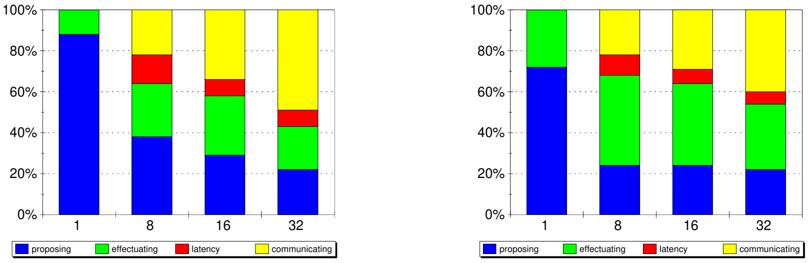

Figure 5.7: Proportional profiles for d18512 (left) and pla85900 (right).

However, most approaches based on partitioning cities either lead to a deteriorated solution quality or display little speed-up, which was also observed in our earlier attempts to design an efficient parallel Lin–Kernighan algorithm.

Figure 5.6 and 5.7 present some more detailed information on the total and proportional amount of time spent by processors in each of the following four phases: proposing exchanges, being idle, communicating exchanges, and effectuating exchanges. Measurements are done on the PowerXplorer using segment trees and first-order Delaunay sets. First, we note that for a given instance, the total amount of time spent in each phase of the algorithm decreases with an increasing number of processors. Typically, the total time needed in the proposal phase, the idle time, and the effectuation time decrease almost linear with an increasing number of processors. This is explained by the observations in Section 5.4.2. Furthermore, we observe that the proportional amount of time to combine and effectuate proposed exchanges increases for larger instances and decreases for larger number of processors. This is also explained in Section 5.4.2. Finally, we observe that also the total time needed for communication decreases with an increasing number of processors. This indicates that the decrease in the total number of

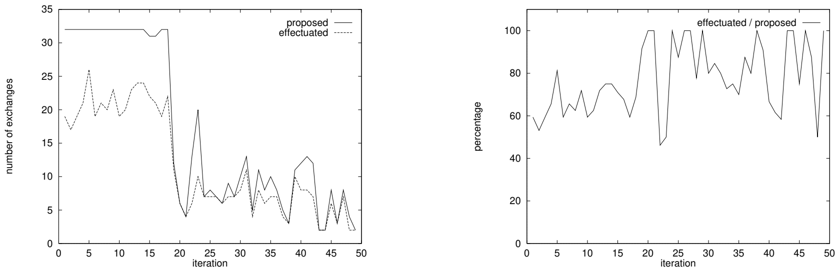

Figure 5.8: Total and proportional amounts of proposed and accepted exchanges.

all-to-all broadcasts dominates the increase in time needed for all-to-all broadcasting. The proportional amount of time spent in communication, however, increases for larger number of processors, and is the bottle-neck for the speed-up of the algorithm. So the scalability of our algorithm for larger number of processors depends on the efficiency of all-to-all broadcasting.

The typical behavior of the combination function is illustrated in Figure 5.8 in which the absolute number of proposed and accepted exchanges is given as well as the proportional amounts for each iteration of the parallel algorithm when applied to the instance rl5934 using 32 processors. We observe that during the first iterations of the algorithm all processors find profitable exchanges of which 60% to 80% can be effectuated. After this, the number of profitable exchanges drops significantly and the effectuation rate varies between 40% and 100%. Overall it is fair to say that the combination function utilized in the parallel algorithm is quite effective, considering the intricate behavior of the exchange function τ_λ.

In this chapter we have shown that multiple-step parallelism can be applied successfully in the design of effective local search algorithms for the TSP. We have used solution decomposition for the parallel 2-opt and 3-opt algorithms and domain decomposition for the parallel Lin–Kernighan algorithm. Computational results show that reasonable speed-ups are achieved and that the parallel Lin–Kernighan algorithm is competitive with sophisticated sequential implementations, both with respect to running times and quality of final solutions. Moreover, multiple-step parallelism based on domain decomposition is fairly robust with respect to the underlying Lin–Kernighan implementation and data structures used to represent tours. Hence, the proposed approach will give good speed-ups for most Lin–Kernighan implementations provided that the target machine allows efficient all-to-all communication.

6

The Steiner Tree Problem

This chapter discusses sequential and parallel local search for the Steiner tree problem in graphs. We introduce novel neighborhoods whose computational time and space complexity is smaller than those known in the literature. We present computational results for benchmark instances from Beasley [1990] and instances derived from real-world TSP instances, which contain up to 18,512 vertices and 325,093 edges. These results show that good-quality solutions can be obtained in moderate running times.

Furthermore, we present a parallel local search algorithm based on multiple-step parallelism and an optimal polynomial-time combination function. Computational results show that good speed-ups can be obtained without loss in quality of final solutions.

6.1 Local search for the Steiner tree problem

In the Steiner tree problem in graphs a minimum weight subtree has to be found that includes a prespecified subset of vertices of a graph. The Steiner tree problem occurs in several practical applications, such as the design of telephone, pipeline, and transportation networks, and the design of integrated circuits. Although the Steiner tree problem is NP-hard [Hwang, Richards & Winter, 1992], several sophisticated optimization algorithms exist that are able to solve instances with up to 2,500 vertices and 62,500 edges, at the cost however of substantial amounts of running time, viz., several hours on a powerful workstation or supercomputer. In addition many heuristics have been proposed, most of which have running times that are polynomially bounded at the risk of finding sub-optimal solutions. Hwang, Richards & Winter [1992] present an overview of both optimization algorithms and heuristics.

Local search has been applied so far only to relatively small instances of the Steiner tree problem containing up to 100 vertices, requiring several minutes of running time. Such instances are well within the range of current optimization algorithms, using a smaller amount of running time. So up to now local search has not been competitive with the best known optimization algorithms. In this chapter we present a local search algorithm that is able to find solutions with a relative excess of a few percent requiring moderate running times. Moreover, it is able to handle large problem instances in acceptable amounts of time.

Formally the Steiner tree problem in graphs (STPG) is defined as follows.

Definition 6.1 (STPG). Given are an undirected graph $G = (V_G, E_G)$, a function $d : E_G \to \mathbb{N}$ that assigns weights to edges, and a set of *terminals* $X \subseteq V_G$. The problem is to find a subtree $T = (V_T, E_T)$ of G with $X \subseteq V_T \subseteq V_G$ and $E_T \subseteq E_G$ such that the sum of the edge weights $\sum_{e \in E_T} d(e)$ is minimal. □

Vertices in $V_G \setminus X$ are called *non-terminals*. Since no non-terminals with degree one are included in an optimal solution, the solution space $\mathcal{S}$ consists of all subtrees T of G with $X \subseteq V_T$ that contain no non-terminals with degree one. Such a tree is called a *Steiner tree*. Non-terminals in a Steiner tree T are called the *Steiner vertices* of T. Steiner vertices with a degree at least three are called *key vertices*. A *key path* is a path in a Steiner tree T of which all intermediate vertices are Steiner vertices with degree two in T and whose end vertices are terminals or key vertices. It follows directly that a minimal Steiner tree consists of key paths that are shortest paths between key vertices or terminals. A basic property of the STPG is that Steiner trees contain at most $|X| - 2$ key vertices, and consequently they consist of at most $2|X| - 3$ key paths.

6.1.1 Neighborhoods for the STPG

The question of finding appropriate neighborhoods for the Steiner tree problem in graphs has been addressed by several authors. Duin & Voß [1993] distinguish between *node-oriented* neighborhoods based on exchanging Steiner vertices, and *edge-oriented* neighborhoods based on edge exchanges.

Osborne & Gillett [1991] propose a neighborhood based on the observation that a Steiner tree T can be represented by its vertices V_T since its edges E_T can be determined by computing a minimum spanning tree for the subgraph of G induced by V_T. Neighbors are then constructed by adding and removing elements to and from V_T. A similar neighborhood is used in the genetic algorithm of Kapsalis, Rayward-Smith & Smith [1993]. Although the neighborhood size is $\mathcal{O}(|V_G|)$, verification of local optimality is computationally expensive for these neighborhoods, as construction of neighbors requires a minimum spanning tree computation that has $\mathcal{O}(|E_G| \log |V_G|)$ time complexity.

Voß [1992] and Dowsland [1991] propose 1-opt neighborhoods based on exchanging key paths. Neighbors are constructed by removing one key path from a Steiner tree T and connecting the remaining two components by a shortest path between two arbitrarily chosen vertices, one in each component, such that a Steiner tree is obtained. A disadvantage of this neighborhood is that the complexity of verification of local optimality, and thus the complexity of a single local search step, is $\mathcal{O}(|X||V_G|^2)$, since a Steiner tree T contains at most $2|X| - 3$ key paths. Moreover, a pre-processing step of $\mathcal{O}(|V_G|^3)$ is needed to compute shortest paths between all pairs of vertices, and storing all paths requires $\mathcal{O}(|V_G|^2)$ space complexity. Dowsland [1991] furthermore presents an extended 1-opt neighborhood that consists of the above neighborhood extended with Steiner trees obtained by connecting the two components by shortest paths from any vertex not in the tree to any two vertices, one in each component. This extension is needed to prove complete connectivity of this neighborhood. The time complexity to evaluate all neighbors in this neighborhood is $\mathcal{O}(|X||V_G|^3)$, which makes it not suited for local search algorithms in which neighborhoods have to be enumerated, e.g., iterative improvement or tabu search.

We present novel neighborhood structures and exchange functions with improved time and space complexities, which makes these neighborhoods more suitable for larger problem instances. Moreover no pre-processing step is required to compute all-pairs shortest paths. The exchange function of these neighborhoods is based on the following observation. Removal of key path splits a Steiner tree into components. Reconnection of these components can be considered as a new STPG instance in which components are treated as terminals. A STPG instance with two terminals can be solved in polynomial time by computing a shortest path between these terminals. This observation gives rise to the following neighborhood.

Definition 6.2. Let T be a Steiner tree that consists of K key paths, $l_1, \ldots, l_K$. Let S_i, S_i' be the two components that remain after removal of key path l_i from T. Let $sp : \mathcal{P}(V_G) \times \mathcal{P}(V_G) \rightarrow \mathcal{P}(E_G)$ give a shortest path from a subset of vertices to another subset of vertices. Then, the neighborhood $\mathcal{N}_1$ is defined by

$$\mathcal{N}_1(T) = \{S_i \cup S_i' \cup sp(V_{S_i}, V_{S_i'}) \mid S_i \cup l_i \cup S_i' = T \ \wedge \ 1 \leq i \leq K\}.$$

□

The number of key paths in neighboring Steiner trees can differ. Removal of a key

```
**proc** $sp$ $(W, W' : \mathcal{P}(V_G))$
  **begin**
    **for** $v \in V_G \setminus W$ **do** $m(v) := \infty$ **od**
    **for** $v \in W$ **do** $m(v) := 0$ **od**
    $Y_0 := \emptyset;\ Y_1 := W;\ v \in W;$
    **while** $Y_1 \neq \emptyset \ \wedge\ v \notin W'$ **do**
      $v \in \{w \in Y_1 \mid m(w) = \min_{w' \in Y_1} m(w')\};$
      **if** $v \notin W'$ **then**
        **for** $v' \in \{w \in V_G \mid (v, w) \in E_G\}$ **do**
          **if** $v' \notin Y_0$ **then**
            $m(v') := \min\{m(v'), m(v) + d(v, v')\};$
            $Y_1 := Y_1 \cup \{v'\}$
          **fi**
        **od**
      **fi**
      $Y_0 := Y_0 \cup \{v\};\ Y_1 := Y_1 \setminus \{v\}$
    **od** $\{m(v)$ gives the length of the shortest path from $W$ to $W'\}$
  **end**
```

Figure 6.1: A multiple-source shortest path algorithm.

path that ends in a key vertex with degree three turns that key vertex into a non key vertex and the two remaining key paths are merged into a single key path. Addition of a key path that ends in a non key vertex converts that vertex into a key vertex, and the key path that passes through it is split into two key paths. So $|T| - 2 \leq |T'| \leq |T| + 2$ for $T' \in \mathcal{N}_1(T)$, where $|T|$ denotes the number of key paths in T. The number of key paths whose removal has to be considered in a given neighborhood $\mathcal{N}_1(T)$ is at least $|X| - 1$ and at most $2|X| - 3$ since a Steiner tree T contains at least $|X| - 1$ and at most $2|X| - 3$ key paths. Hence, the size of a neighborhood is at most $2|X| - 3$. An interesting observation is that solutions only have neighbors with lower or equal cost, because only paths with at most the length of the removed path are inserted since this length is an upper bound on the length of the shortest path between components. So an attempted replacement of a key path in a Steiner tree can lead to the same Steiner tree if no shorter path exists. In particular this can imply that local minima have no other neighbors.

The function *sp* is computed using the algorithm of Figure 6.1. The time complexity of this algorithm is $\mathcal{O}(|E_G| \log |V_G|)$ if the set Y_1 is represented by the classical *heap* data structure. Identification of the two components that arise when a key path is removed from a Steiner tree can be done in $\mathcal{O}(|V_G|)$ time.

An important observation is that the algorithm of Figure 6.1 computes the shortest paths to vertices in ascending order. In this algorithm $m(v)$ gives the length of a shortest path to a vertex $v \in Y_0$. The algorithm can be terminated as soon as $m(v) \geq d(l_i)$ for a vertex $v \in Y_0$ and a removed key path l_i, because then the shortest path from S_i to S'_i is at least $d(l_i)$ long. Consequently, replacement of l_i cannot lead to a lower-cost tree. This leads to the following upper bound for the time complexity of the function sp. Let T be a Steiner tree from which a path l with length $d(l)$ is removed, which results in two components S, S'. Let W be the largest set of vertices that are within distance $d(l)$ from S for some l in T. Then, the complexity of the function sp is $\mathcal{O}(|V_G| + |W|^2 \log |W|)$.

A consequence of inserting only shortest paths between components is that the neighborhood $\mathcal{N}_1$ is not connected, i.e., it is not always possible to reach a globally minimal Steiner tree by a sequence of exchanges, as can be seen by a simple example. Therefore, we present a neighborhood $\mathcal{N}'_1$ that is a small augmentation of $\mathcal{N}_1$ such that it is sufficiently connected, i.e., it is possible to reach an optimal solution from any solution by a sequence of exchanges. Moreover, $\mathcal{N}'_1(T)$ also contains neighbors with higher cost than that of a Steiner tree T, so here neighborhoods of local minima contain other Steiner trees. Such neighborhoods are needed in tabu search algorithms to escape from local minima.

In $\mathcal{N}'_1$ also a single key path is replaced with another key path, as is the case in $\mathcal{N}_1$. The supplement of $\mathcal{N}'_1$ to $\mathcal{N}_1$ consists of neighbors constructed by adding shortest paths from one component to the other component via vertices that are positioned at a distance of one edge from the other component.

Definition 6.3. Let T be a Steiner tree that consists of K key paths, $l_1, \ldots, l_K$. Then, the neighborhood $\mathcal{N}'_1(T)$ is equal to

$$\{S_i \cup S'_i \cup sp(V_{S_i}, v) \cup \{(v, v')\} \mid T = S_i \cup l_i \cup S'_i \wedge v' \in S'_i \wedge (v, v') \in E_G \wedge 1 \leq i \leq K\}.$$

□

The complexity of evaluating the cost of all neighbors in a neighborhood $\mathcal{N}'_1(T)$ of a Steiner tree T is $\mathcal{O}(|X||E_G| \log |V_G|)$, as all neighbors originating from removal of the same key path can be evaluated in a single execution of the multiple-source shortest path algorithm. The neighborhood $\mathcal{N}'_1(T)$ is a strict subset of the extended 1-opt neighborhood of Dowsland [1991] and all excluded neighbors originating from removal of a given key path have higher cost than those that are included in $\mathcal{N}'_1(T)$. The average cost of neighbors in $N'_1(T)$ is therefore lower than in Dowsland's extended 1-opt neighborhood, which also has a much larger computational complexity than $\mathcal{N}'_1$, viz. $\mathcal{O}(|X||V_G|^3)$. $\mathcal{N}'_1$ is not completely connected since it is not possible to transform a Steiner tree into another Steiner tree that differs only in a single key path that is not a shortest path between two components. However, we have the following result.

Theorem 6.1. *The neighborhood structure $\mathcal{N}_1'$ is sufficiently connected.*
Proof. Let T^* be an optimal Steiner tree, let T be a Steiner tree, and let $sp(A, B)$ denote a shortest path from component A to B. Partition T^* into subtrees such that all leaves are terminals and all other vertices are Steiner vertices. Let $S \subseteq T^*$ be such a subtree that is not duplicated in T. We consecutively add adjacent key paths in S to T, starting with the key paths rooted from the leaves in S. We distinguish between two cases based on whether vertex i with which a vertex $a \in T$ has to be connected, is included in T.

Let $i \in T$, and let $sp(a, i)$ be a key path in T^* not in T that is to be added. Remove a key path l from T not in T^* such that addition of $sp(a, i)$ would result in a Steiner tree. This is always possible since addition of $sp(a, i)$ to T gives a cycle in T of which at least one key path is not included in T^*. Removal of l splits T in components A and B with $a \in A$ and $i \in B$. Let $(j, i) \in sp(a, i)$ and construct $T' = A \cup B \cup sp(A, j) \cup \{(j, i)\} \in \mathcal{N}_1'(T)$. If $sp(a, j) \neq sp(A, j) = sp(a', j)$, then addition of $sp(a, j)$ to T' results in a cycle. So there exists a key path in T' not in T^* whose replacement with $sp(a, j)$ would result in a Steiner tree. We can repeat the above procedure, adding one edge of $sp(a, i)$ at a time, until the entire path $sp(a, i)$ has been added.

Let $i \notin T$, then i is a key vertex connected with at least three vertices $a, b, c \in S$ that also exist in T. Let $sp(a, i)$ be a key path in T^* not in T that is to be added. Remove a key path l from T not in T^* such that addition of $sp(b, a)$ would result in a Steiner tree. This is always possible since addition of $sp(b, a)$ to T gives a cycle in T of which at least one key path is not included in T^*. Removal of l splits T in components A and B with $a \in A$ and $b \in B$. Let $(j, a) \in sp(a, i)$ and construct $T' = A \cup B \cup sp(B, j) \cup \{(j, a)\} \in \mathcal{N}_1'(T)$. Next, remove a key path l' from T' not in T^*, which results in components A' and C, such that addition of $sp(c, j)$ would give a Steiner tree. Let $(j', j) \in sp(a, i)$ and construct $T'' = A' \cup C \cup sp(C, j') \cup \{(j', j)\} \in \mathcal{N}_1'(T')$. Repeating these steps of adding edges of $sp(a, i)$ one after the other, starting alternately from b and c, gives a Steiner tree that includes i, at which point the former case applies.

Sort the key paths of S according to the minimum number of key paths that need to be traversed to reach a terminal. Once a key path of subtree S in T^* has been added to T, other key paths from S that do not exist in T^* are added to T in order of ascending rank. Repeating this construction for other subtrees $S \subset T^*$ not duplicated in T gives a valid sequence of exchanges in $\mathcal{N}_1'$ that transforms a Steiner tree T into an optimal Steiner tree T^*. □

The neighborhood structures $\mathcal{N}_1$ and $\mathcal{N}_1'$ are based on the insertion of a single shortest path between two components. They are inspired by the polynomially solvable Steiner tree problem with two terminals. Chen [1983] presents a polynomial algorithm for the Steiner tree problem with three terminals. This motivates the following neighborhood in which two paths are removed from a Steiner tree. Reconnection of the remaining three components can be done optimally in polynomial time with the following algorithm that improves upon the time complexity of the algorithm of Chen [1983]. The algorithm is based on the following property.

Property 6.1. Consider an STPG instance with three terminals s, s', s'', and let T be a minimal Steiner tree with key vertex w. Then, the length of the path in T from w to s and the length of the path from w to s' is at most $d(sp(s, s'))$. □

Note that it is possible that w coincides with a terminal. If we replace terminals s, s', s'' with three components S, S', S'' we see that the key vertex w in a minimal Steiner tree that contains the components S, S', S'' is included in the set W of vertices for which the shortest path to S and the shortest path to S' is at most $d(sp(S, S'))$. This optimal key vertex w can be found by the following algorithm.

(1) Construct the set of vertices for which the shortest path to S is at most $d(sp(S, S'))$. This set is given by the final value of the set Y_1 in the algorithm of Figure 6.1 for the computation of $sp(S, S')$. Similarly, the set of vertices for which the shortest path to S' is at most $d(sp(S, S'))$ is given by the final value of Y_1 in the computation of $sp(S', S)$. The set W of candidate key vertices is the intersection of these sets.

(2) Determine the shortest path from W to S'' using the algorithm of Figure 6.1. In this algorithm $m(v)$, $v \in W$, has to be initialized as the distance from S to v plus the distance from S' to v. The root vertex w of this path is the key vertex in a minimal Steiner tree T for the components S, S', S''.

The time complexity to find w is $\mathcal{O}(|E_G| \log |V_G|)$. At most three additional key paths need to be added to construct a minimal Steiner tree that contains S, S', and S''. Let $stp_3(S, S', S'')$ denote this set of key paths that can be found with the above algorithm. Using the function stp_3, we can define the following 2-exchange neighborhood.

Definition 6.4. Let T be a Steiner tree that consists of K key paths, $l_1, \dots, l_K$. Let the function stp_3 return, on input of three components, the additional key paths in a minimal Steiner tree that contains these components. Then, the neighborhood $\mathcal{N}'_2(T)$ is equal to

$$\{S \cup S' \cup S'' \cup stp_3(S, S', S'') \mid T = S \cup S' \cup S'' \cup l_i \cup l_j \ \wedge \ 1 \le i < j \le K\}.$$

□

The size of this neighborhood is $\mathcal{O}(|X|^2)$ as the number of key paths in a Steiner tree is at most $2|X|-3$. The complexity of identifying S, S', S'' is $\mathcal{O}(|V_G|)$, and the above algorithm to implement the function stp_3 requires $\mathcal{O}(|E_G|\log|V_G|)$ time. Hence, the complexity to verify local optimality of a Steiner tree for $\mathcal{N}_2'$ is $\mathcal{O}(|X|^2|E_G|\log|V_G|)$. The time complexity of the algorithm to implement stp_3 can be reduced by terminating it as soon as $m(v) > d(l_i)+d(l_j)$, where $m(v)$ is the summed length of the shortest paths from S, S', and W to $v \in V_G$, because then replacing key paths l_i and l_j cannot lead to a Steiner tree with lower cost.

For all neighbors $T' \in \mathcal{N}_2'(T)$ holds that $f(T') \leq f(T)$ since neighbors are constructed by computing a minimal Steiner tree to connect the remaining components after removal of two key paths. Removal of two key paths can, furthermore, lead to a Steiner tree in which Steiner vertices with degree one exist. This occurs if the two removed key paths share a key vertex with degree three in T. The remaining key path from this vertex can also be removed from T, which leads to an additional cost decrease of T'. So in some cases three key paths are removed from a Steiner tree T to construct its neighbors in $\mathcal{N}_2'(T)$. The neighborhood $\mathcal{N}_1(T)$ is generally not a subset of $\mathcal{N}_2'(T)$, and in some cases their intersection can even be empty. However, the following result holds.

Theorem 6.2. *Let a Steiner tree $T \in \mathcal{S}$ be given. If T is a local minimum of $\mathcal{N}_2'$, then T is also a local minimum of $\mathcal{N}_1$. Moreover, let $T' \in \mathcal{N}_1(T)$ and $T'' \in \mathcal{N}_2'(T)$ with $l, l' \in T$, $l \notin T'$, and $l, l' \notin T''$, then $f(T'') \leq f(T')$.*

Proof. Let T be a local minimum of $\mathcal{N}_2'$ and assume that T is not a local minimum of $\mathcal{N}_1$. Then, for some $T' = S \cup S' \cup \{l\} \cup sp(S \cup S' \cup \{l\}, S'') \in \mathcal{N}_1(T)$, in which key path l' is removed, holds $f(T') < f(T)$. T' is also a Steiner tree that connects the components S, S', S'' that arise when key paths l and l' are removed from T, so $f(T') \geq f(T)$, which contradicts the assumption. Furthermore, for $T'' = S \cup S' \cup S'' \cup stp_3(S, S', S'') \in \mathcal{N}_2'(T)$ holds that $f(T'') \leq f(T')$ as T'' is a minimal Steiner tree that connects the components S, S', and S''. □

A disadvantage of the neighborhood $\mathcal{N}_2'$, which limits its practical usefulness, is its time complexity of $\mathcal{O}(|X|^2|E_G|\log|V_G|)$ for verifying local optimality. Therefore, we present the following neighborhood $\mathcal{N}_2$ that is a restriction of $\mathcal{N}_2'$ with a smaller time complexity for verifying local optimality.

Definition 6.5. Let T be a Steiner tree that consists of K key paths, $l_1, \ldots, l_K$. Let the function stp_3 return, on input of three components, the additional key paths in a minimal Steiner tree that contains these components. Then, the neighborhood $\mathcal{N}_2(T)$ is equal to

$$\{S \cup S' \cup S'' \cup stp_3(S, S', S'') \mid T = S \cup S' \cup S'' \cup l_i \cup l_j \wedge l_i \cap l_j \neq \emptyset \wedge 1 \leq i < j \leq K\}.$$

□

To obtain a neighbor in $\mathcal{N}_2$ two key paths are removed that have a vertex in common, which can be a terminal or a key vertex. Typically, if this key vertex has degree three in the Steiner tree, the remaining key path is also removed. The three remaining components are connected using the function stp_3 to compute a minimal Steiner tree for three components. The size of the neighborhood $\mathcal{N}_2(T)$ for a Steiner tree T is $\mathcal{O}(\kappa|X|)$, where κ is the maximum degree of a key vertex in T. Consequently, the complexity of verifying local optimality of T for $\mathcal{N}_2$ is then $\mathcal{O}(\kappa|X||E_G|\log|V_G|)$, which is substantially less than the complexity of verifying local optimality for $\mathcal{N}'_2$. Moreover, it still holds that a local minimum of $\mathcal{N}_2$ is a local minimum of $\mathcal{N}_1$ since $\min_{T' \in \mathcal{N}_2(T)} f(T') \leq \min_{T'' \in \mathcal{N}_1(T)} f(T'')$ for a Steiner tree $T \in \mathcal{S}$ for similar reasons as outlined in the proof of Theorem 6.2.

6.1.2 Enumeration of neighborhoods

An important aspect for the running time of an iterative first-improvement algorithm is how neighborhoods are enumerated. Neighborhoods should be enumerated such that exchanges that are not likely to lead to lower-cost solutions are examined last. Of course it is generally not possible to determine beforehand which exchanges do not lead to a lower-cost solution, but often exchanges that do not yield any gain when applied to a Steiner tree T also do not lead to a lower-cost Steiner tree when applied to a neighbor $T' \in \mathcal{N}(T)$. This implies that it is profitable to enumerate neighborhoods in such a way that exchanges that have not yet been examined, are explored first; this is called *circular neighborhood search* [Papadimitriou & Steiglitz, 1982]. To this end, a function b is introduced that assigns a boolean to arguments of the exchange function that indicates whether the corresponding exchange has been evaluated and has not led to cost decrease. While enumerating the neighborhood of the current solution in a local search algorithm, first those exchanges are examined for which this boolean is false. If no lower-cost neighbor has been found for these exchanges, then the remaining exchanges are evaluated.

In the neighborhood $\mathcal{N}_1$ only a single path is removed in an exchange, so booleans can be associated with key paths, and the space complexity for storing the boolean function b is $\mathcal{O}(|X|)$. In the neighborhood $\mathcal{N}_2$ exchanges can be identified by marking key vertices, because if a key vertex in a Steiner tree has degree three, neighbors are obtained by removing the three key paths that end in this key vertex. Considering that most key vertices in a Steiner tree have degree three, we can associate booleans with key vertices to indicate whether they have been exchanged. The space complexity of b can then be reduced to $\mathcal{O}(|X|)$. In tabu search certain exchanges are also not explored, but an essential difference with the tabu list in tabu search and the usage of the function b is that b is only used to achieve circular neighborhood search, not to prohibit exploration of exchanges.

6.1.3 Computational results

We have implemented iterative first-improvement algorithms that use the neighborhoods $\mathcal{N}_1$ and $\mathcal{N}_2$. A Steiner tree is represented by a list of records that contain key paths. Each record also contains cross references to key paths that share end vertices with this key path. Furthermore, a function is maintained that gives for each vertex included in the Steiner tree, a key path that contains this vertex. Given the end vertices of a key path it is possible to find the corresponding key path by following the cross references. In this way removal and addition of key paths can be done in a time linearly bounded by the number of vertices in a key path and the degrees of the end vertices. It should be noted that removal or addition of key paths can require that other key paths are joined to one key path or split into two key paths.

Computational results are presented for 40, randomly generated, instances from Beasley [1990]. Furthermore, a number of real-world Euclidean traveling salesman problem instances from Reinelt's TSPLIB are transformed to Steiner tree problem instances, since for many combinatorial optimization problems it is observed that real-world instances are typically much harder to solve than randomly generated instances. This transformation is done as follows. The graph required in an STPG instance is the extended Delaunay graph for the corresponding TSP instance as defined in Definition 5.9. Recall that an extended Delaunay graph G is a planar graph that contains at most $3n$ edges, where n is the number of vertices. We have utilized k-th order Delaunay graphs, with $k = 1, 2, 3, 4$, to construct STPG instances with more edges. From each graph, constructed in this way, three STPG instances are derived by randomly designating 15%, 25%, or 35% of the vertices as terminals. No reduction techniques to reduce Steiner tree problem instances have been used. Initial solutions are constructed with a shortest path heuristic [Takahashi & Matsuyama, 1980] as follows. Starting with an initial terminal, terminals closest to the tree constructed so far are consecutively connected by shortest paths to this tree until all terminals are added.

Tables 6.1 and 6.2 present the results obtained with our iterative improvement algorithm. In these tables $|V|$, $|E|$, and $|X|$ give the number of vertices, edges, and terminals, respectively. For all problems but one (e18), in Beasley's series D and E, optimal solutions are known [Beasley, 1989] For the Euclidean STPG instances optimal solutions are only known for the instances f1 – f24 that contain up to 1,000 vertices. These solutions are computed by the exact algorithm of Duin [1994], which is among the fastest exact algorithms available for the STPG. It can handle instances with at most 1,000 vertices due to memory limitations. The optimal solution value is given in the column "opt". The best solution found in ten runs of the iterative improvement algorithm is given in the column labeled

					$\mathcal{N}_1$			$\mathcal{N}_2$		
	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert X\rvert$	opt	best	ϵ_{avg}	$t\,(s)$	best	ϵ_{avg}	$t\,(s)$
d1	1000	1250	5	106	106	2.64	0.7	106	0.94	1.1
d2			10	220	220	0.73	0.3	220	0.27	0.6
d3			167	1565	1567	0.22	9.4	1565	0.00	13.7
d4			250	1935	1939	0.34	11.9	1935	0.10	16.2
d5			500	3250	3254	0.18	24.7	3251	0.12	44.5
d6		2000	5	67	70	5.67	0.8	67	2.23	1.9
d7			10	103	103	0.00	0.9	103	0.00	1.2
d8			167	1072	1082	1.64	8.2	1075	1.13	11.6
d9			250	1448	1454	0.70	14.5	1450	0.58	23.4
d10			500	2110	2119	0.70	29.5	2113	0.44	41.0
d11		5000	5	29	29	4.14	0.8	29	3.05	1.0
d12			10	42	42	3.81	0.1	42	0.24	0.3
d13			167	500	509	2.46	9.2	502	1.86	23.5
d14			250	667	674	1.27	15.0	670	0.90	26.3
d15			500	1116	1123	0.79	32.6	1116	0.46	73.8
d16		25000	5	13	13	3.08	0.1	13	0.00	0.3
d17			10	23	23	4.35	0.3	23	2.64	0.4
d18			167	223	236	7.40	12.3	230	4.41	30.8
d19			250	310	336	9.32	22.8	321	5.77	53.5
d20			500	537	558	4.49	53.4	547	2.29	112.0
e1	2500	3125	5	111	111	0.00	0.2	111	0.00	0.1
e2			10	214	214	2.62	0.5	214	0.00	1.2
e3			417	4013	4035	0.64	57.1	4023	0.34	86.0
e4			625	5101	5118	0.35	85.2	5103	0.20	192.6
e5			1250	8128	8130	0.11	171.4	8128	0.01	305.3
e6		5000	5	73	73	2.04	0.2	73	0.00	0.3
e7			10	145	145	3.70	0.7	145	1.69	2.2
e8			417	2640	2661	0.98	76.2	2655	0.73	170.8
e9			625	3604	3633	1.03	120.1	3617	0.65	212.5
e10			1250	5600	5621	0.47	242.9	5605	0.37	431.0
e11		12500	5	34	34	4.41	0.2	34	0.00	0.4
e12			10	67	67	1.49	0.6	67	0.59	1.0
e13			417	1280	1312	2.82	77.5	1299	1.98	182.3
e14			625	1732	1756	1.59	139.4	1745	1.07	293.1
e15			1250	2784	2794	0.69	265.2	2784	0.28	728.8
e16		62500	5	15	15	6.67	0.3	15	0.00	1.0
e17			10	25	25	4.80	0.8	25	2.40	1.3
e18			417	568*	613	8.87	94.1	593	5.04	300.8
e19			625	758	809	7.68	156.3	790	4.63	397.5
e20			1250	1342	1398	4.71	384.5	1370	2.77	845.9

Table 6.1: Results for Beasley's series D and E.

					$\mathcal{N}_1$		$\mathcal{N}_2$	
	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert X\rvert$	opt	ϵ_{avg}	$t(s)$	ϵ_{avg}	$t(s)$
f1	783	2322	117	2992	1.51	4.4	1.13	7.3
f2			196	3826	0.69	7.5	0.51	12.8
f3			274	4401	1.23	8.9	0.77	21.5
f4		7532	117	2899	1.92	5.1	1.44	10.1
f5			196	3744	1.85	8.8	1.29	26.2
f6			274	4368	1.46	10.1	0.52	27.1
f7		28365	117	2892	1.03	28.9	0.67	54.7
f8			196	3742	1.40	18.2	0.89	35.0
f9			274	4368	0.98	19.5	0.47	44.8
f10		109895	117	2892	1.13	24.9	0.51	42.5
f11			196	3742	1.47	51.5	1.04	124.6
f12			274	4368	1.01	53.9	0.35	163.6
f13	1000	2981	150	6509412	2.30	11.2	1.53	35.9
f14			250	8123458	1.26	16.8	0.91	29.7
f15			350	9804964	1.23	24.0	0.67	57.2
f16		9699	150	6316653	2.27	11.3	1.35	30.2
f17			250	7946357	0.71	19.5	0.55	34.3
f18			350	9675663	1.06	26.1	0.82	71.9
f19		38080	150	6297956	1.97	20.2	1.05	35.0
f20			250	7931250	0.61	41.8	0.46	84.7
f21			350	9673224	0.94	98.7	0.36	295.7
f22		184597	150	6297956	1.63	66.4	1.01	143.2
f23			250	7931250	0.67	99.1	0.33	161.6
f24			350	9673224	0.50	151.7	0.35	277.1

	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert X\rvert$	best	ϵ_{avg}	$t(s)$	ϵ_{avg}	$t(s)$
g1	3795	11326	569	13201	0.66	153	0.18	291
g2			949	14693	0.54	276	0.24	497
g3			1328	15868	0.73	367	0.27	1064
g4		52684	569	13070	0.36	237	0.10	503
g5			949	14569	0.54	394	0.10	670
g6			1328	15681	0.60	547	0.27	1551
g7		325093	569	13012	0.49	850	0.11	1867
g8			949	14556	0.50	1171	0.23	2927
g9			1328	15616	0.65	1770	0.20	3009
g10	11849	35532	1777	381324	0.62	2476	0.12	5816
g11			2962	478669	0.68	4296	0.17	12338
g12			4147	559579	0.55	5740	0.14	14940
g13	14051	42128	2108	153663	0.59	2698	0.06	5002
g14			3513	197335	0.67	4565	0.15	10682
g15			4918	234213	0.63	6177	0.11	14424
g16	18512	55510	2777	204314	0.71	4761	0.09	7998
g17			4628	271714	0.74	7369	0.13	15622
g18			6479	322698	0.67	10338	0.14	30807

Table 6.2: Results for Euclidean instances.

"best" and the average relative excess of these ten final solutions in percentages over the optimal solution, or best known upper bound for the series G, is given by ϵ_{avg}. The average running time in seconds on a Sun Classic workstation is given by $t(s)$.

We observe that our iterative improvement algorithms find solutions with relative excess of at most a few percents over the optimal solution for the instances of Beasley [1990] in a small amount of running time (at most a few minutes, whereas exact algorithms require several hours on supercomputers or powerful workstations for instances with 2,500 vertices). For several instances optimal solutions are found. For the Euclidean instances with up to 1,000 vertices we are able to find solutions with relative excesses of 0.3% – 2.2% in a few seconds. Furthermore, the algorithms are able to handle substantially larger instances in moderate amounts of running time.

We are able to deal with much larger instances than those studied in other local search approaches as presented by Dowsland [1991], Osborne & Gillett [1991], and Kapsalis, Rayward-Smith & Smith [1993], who all deal with instances of at most 100 vertices, requiring running times that range from several minutes to a few hours. Moreover, our approach has limited memory requirements since only a compact representation of an instance is stored. Most other heuristic and exact approaches require a pre-processing step of $\mathcal{O}(|V_G|^3)$ time and $\mathcal{O}(|V_G|^2)$ space to compute and store all-pairs shortest paths, which makes it hard to handle large instances.

6.2 Parallel local search for the Steiner tree problem

Although good-quality solutions can be found with local search for the STPG, running times are still considerable for the larger instances. In this section we investigate the applicability of multiple-step parallelism to reduce these running times. An algorithm with multiple-step parallelism for the STPG can be outlined as follows. Here, P denotes the number of processors.

(1) Partition the domain of the exchange function applied to a Steiner tree T into P subdomains and let each processor evaluate such a subdomain.

(2) Communicate profitable exchanges to other processors.

(3) Effectuate a subset of the profitable exchanges found in step (1), which results in a new Steiner tree T'.

(4) Replace T by T', and repeat steps (1) – (3) until no improvement is found.

Next, we discuss a distributed neighborhood structure $\mathcal{D}_1$ that can be employed in the above algorithm. We restrict our discussion of the time complexity associated with $\mathcal{D}_1$ to synchronous multiple-step parallelism as our target machine is a message-passing MIMD machine. Synchronization between the steps of the above multiple-step parallel local search algorithm is typically necessary on this machine because it has a decentralized architecture in which there is no central processor that governs accesses to the current Steiner tree.

Domain distribution. The distribution structure of $\mathcal{D}_1$ assigns the entire current Steiner tree to all processors. To define the domain distribution of $\mathcal{D}_1$, we note that the additional arguments for applying the exchange function τ associated with the neighborhood $\mathcal{N}_1$ to a given Steiner tree T are single key paths that are to be removed from T. So the set of arguments of τ that determines the neighborhood $\mathcal{N}_1(T)$ consists of all pairs (T, l), for key paths $l \in T$. This set of arguments is partitioned into P (almost) equally sized subdomains that specify local neighborhoods. The domain distribution of $\mathcal{D}_1$ defined in this way partitions the neighborhood $\mathcal{N}_1(T)$. If T consists of K key paths, then each processor evaluates at most $\lceil K/P \rceil$ exchanges. So the size of a local neighborhood is $\mathcal{O}(\frac{|X|}{P})$, considering that K is at most $2|X| - 3$.

Combination function. The combination function of the distributed neighborhood structure $\mathcal{D}_1$ specifies how proposed exchanges are combined to form a new Steiner tree. It is not always possible to effectuate all proposed exchanges since some simultaneous key path replacements can split a Steiner tree into disconnected components. So the combination function may only effectuate a subset of the proposed exchanges. Essential for the speed-up of the algorithm is that the total gain that results from effectuating exchanges is as large as possible. The Steiner tree combination problem (STCP) of effectuating a subset of q proposed exchanges such that the resulting gain is maximized, is formulated as follows.

Definition 6.6 (STCP). Given a Steiner tree T with key paths $l_1, \ldots, l_q \in T$ for $q \in \mathbb{N}$. Let $l'_1, \ldots, l'_q$ be paths such that $T \setminus l_i \cup l'_i \in \mathcal{N}_1(T)$ for all $1 \le i \le q$, and let $L = \{l_1, \ldots, l_q, l'_1, \ldots, l'_q\}$. Then, the problem is to find a subset L' of L such that the cost of Steiner tree $T' = T \setminus \{l_1, \ldots, l_q\} \cup L'$ is minimal. □

The STCP is closely related to the problem of finding a minimum spanning tree in which components instead of vertices are to be connected. In the STCP a minimum spanning tree is to be found that connects $q + 1$ components that remain after removal of $l_1, \ldots, l_q$ from T using new key paths $l'_1, \ldots, l'_q$ or removed key paths $l_1, \ldots, l_q$. If key path l'_i is inserted into T' and l_i is removed, a proposed replacement of l_i with l'_i is accepted. In the classical minimum spanning tree problem all edges are connections between vertices. In the STCP, however, inserted key paths l'_i are not necessarily connections between components because an inserted path l'_i may end in a removed key path l_j in which case l'_i is not a direct connection between components.

The STCP can be solved polynomially by generalizing the algorithm for finding a minimum spanning tree of Kruskal [1956]. Kruskal's algorithm is based on the observation that an edge (u, v) has to be included in a minimum spanning tree for a graph $G = (V, E)$ if there exists a subset $U \subseteq V$ with $u \in U$ and $v \in V \setminus U$ such that (u, v) is a shortest edge between U and $V \setminus U$. Let T be a Steiner tree, and let L be defined as in Definition 6.6. The STCP can then be solved as follows.

(1) Remove $\{l_1, \ldots, l_q\}$ from T, which results in $q + 1$ components $C_0, \ldots, C_q$. Let $T' = \{C_0, \ldots, C_q\}$ be this set of components.

(2) Select a smallest length path l from L whose addition to T' does not give a cyclic component or else intersects with a key path l' in L. Remove l from the set L and add it to T'. If l connects two components, merge these components to one. l is not necessarily a connection between two components in T' since it is possible that l ends in a vertex of a key path removed from T. Addition of l can give a cycle in a component C in T' if l intersects with a key path l' removed from T. In this case, remove a maximum length key path from C such that C becomes non-cyclic.

(3) Repeat step (2) until all terminals are contained in a single component, i.e., T' includes a Steiner tree. It may occur that T' includes components without terminals or that T' contains paths that end in a non-terminal with degree one. These components or paths can be removed from T'. After this, T' is a minimum weight tree that spans the components $\{C_0, \ldots, C_q\}$ using paths from L, so T' is an optimal solution for the STCP.

Next, we discuss the complexity of the above algorithm to effectuate a subset of q proposed exchanges. The complexity of selecting the smallest length path from L is constant if L is sorted in a pre-processing phase. Step (2) requires at most $\mathcal{O}(q)$ time in case a cycle is introduced. It is repeated at most $2q$ times. The overall time complexity of the algorithm is therefore $\mathcal{O}(q^2)$. Thus, it is possible to determine in polynomial time a subset of q proposed exchanges whose effectuation gives maximum cost decrease of the current Steiner tree T. In particular, at least one proposed exchange is effectuated by this algorithm, so in each iteration of the parallel algorithm the cost of the current Steiner tree decreases, which guarantees termination of the parallel local search algorithm. Moreover, in each iteration a Steiner tree T' is constructed whose cost is at most that of the lowest-cost neighbor of the current Steiner tree T, which would be accepted by a sequential algorithm with best improvement.

Multi improvement. In sequential local search algorithms, two pivoting rules are distinguished to pick from profitable exchanges. In first-improvement algorithms the first profitable exchange found is the one that is accepted, whereas in best-improvement algorithms all exchanges are evaluated and the one with the largest gain is accepted. A disadvantage of first improvement in a synchronous multiple-step parallel algorithm is that it can lead to load imbalance, because synchronization takes place after the proposal phase. Some processors may find a profitable exchange quickly, while others have to search their entire local neighborhood. Moreover, even the running times required by single exchanges can differ substantially, as the number of vertices explored in the shortest path algorithm can vary significantly for different exchanges. Once a profitable exchange has been found or the entire local neighborhood has been examined, a processor has to wait until other processors have finished their proposal phase since effectuating proposed exchanges requires all-to-all broadcasting. Hence, large load imbalance may occur when first improvement is used as pivoting rule. On the other hand, best improvement is also not efficient because in each step local neighborhoods are searched entirely, although several steps with equal total gain can often be made with less computational effort using first improvement.

Therefore, we use a different pivoting rule in our parallel local search algorithm for the STPG in which the entire local neighborhood is explored, as is the case in best-improvement algorithms, but instead of selecting only the best exchange, all profitable exchanges are memorized and proposed for effectuation. This pivoting rule is called *multi improvement*. Through multi improvement more exchanges are proposed, which leads to a better speed-up, provided that a large proportion of the proposed exchanges can be effectuated. An additional advantage of this approach over first improvement is that a better load balance is obtained because the entire local neighborhood is always explored, regardless of observed profitable exchanges. This leads to smaller differences in running times of the proposal phase for processors since the running time for evaluating neighbors is summed over the entire local neighborhood. The effectiveness of an algorithm with multi improvement of course strongly depends on the ratio of proposed and effectuated exchanges, but this holds for best or first improvement as well. Note that for multi improvement the set of proposed exchanges is independent of the number of employed processors, so the course of the algorithm does not depend on the number of employed processors.

Next, we discuss an upper bound for the speed-up of a parallel algorithm with multi improvement over a sequential algorithm with first improvement. For this we assume that the amount of time needed to communicate and combine proposed exchanges is negligible compared to the time needed for evaluation of local neighbors. Let ρ be the number of exchanges effectuated in the combination step and let α be the average number of exchanges evaluated by a first-improvement algorithm to find a profitable exchange. ρ can be larger than the number of processors P since all profitable exchanges are proposed, but ρ is bounded by K, the number of key paths in the current Steiner tree T. The average running time of a sequential algorithm with first improvement to find ρ profitable exchanges is $\alpha\gamma\rho$, where γ is the average time for evaluating an exchange. A lower bound for the average running time of a parallel multiple-step algorithm on P processors to find ρ profitable exchanges is $\frac{K}{P}\gamma$. So the speed-up is at most $P\frac{\alpha\rho}{K}$. If $\alpha\rho \geq K$ this would imply that a super-linear speed-up might be achieved with multi improvement, in which case it would be beneficial to use multi improvement in a sequential algorithm instead of first improvement. This observation emphasizes the importance of adequate neighborhood enumeration, because in a neighborhood for which $\alpha\rho \geq K$, it is better to first search an entire neighborhood for profitable exchanges, which are subsequently effectuated, than to effectuate exchanges immediately after finding them. In a local search algorithm neighbors are enumerated according to some order imposed on the neighborhood. This order can change through effectuation of exchanges, which can infer that exchanges applied to T and T' that remove the same key paths are evaluated in different orders for neighboring solutions T and T'. If neighbors of T' that are not likely to have lower cost are examined first through this order change, it might be beneficial to examine several neighbors of T first before effectuating profitable exchanges.

In our neighborhoods for the STPG neighbors are enumerated according to a tree traversal order. This order can change through removal and insertion of key paths. In Section 6.1.2 we have introduced a boolean function that is used to adapt the enumeration order of neighborhoods in such a way that exchanges that are not likely to result in lower-cost neighbors, are selected last. Hence, first improvement in combination with a boolean function to guide enumeration of neighborhoods, is likely to be more effective than multi improvement as pivoting rule in a sequential local search algorithm.

Global communication. When a processor has evaluated its local neighborhood and collected the profitable exchanges in this neighborhood, it has to wait until other processors have finished evaluating their local neighborhoods. As soon as all processors have explored their local neighborhoods, all-to-all broadcasting has to take place to communicate all profitable exchanges to all processors. The algorithm for all-to-all broadcasting depends on the target machine on which it is implemented, which is in our case a message-passing MIMD machine configured in a two-dimensional torus.In this machine processors can only be involved in a single communication action at a time, either a send or receive.

Each row and column in a two-dimensional torus is configured as a ring network. Therefore, we first discuss an algorithm to perform all-to-all broadcasting in a ring network with P processors, where $P \bmod 2 = 0$. The idea of this algo-

```
proc Broadcast_Ring(p)
  { $0 \leq p < P \wedge P \bmod 2 = 0$ }
  var $m$ : array $[0, P)$ ;
  begin
    $i := 0$ ; $m(p)$ = proposed exchanges processor $p$ ;
    { $m(p)$ = proposed exchanges proc. $p \wedge \forall_{0 \leq q < P \wedge q \neq p}\ m(q) = \text{nil}$ }
    while $i < P$ do
      if $p \bmod 2 = 0$ then $(p+1)\ !\ m(p-i),\ m(p+1-i)$ ;
                              $(p-1)\ ?\ m(p-2-i),\ m(p-1-i)$
      else$(p-1)\ ?\ m(p-1-i),\ m(p-i)$ ;
          $(p+1)\ !\ m(p-1-i),\ m(p-i)$
      fi
      $i := i + 2$
    od { $\forall_{0 \leq q < P}\ m(q)$ = proposed exchanges of processor $q$ }
  end
```

Figure 6.2: Algorithm for all-to-all broadcasting in a ring for processor p.

rithm is as follows. Let each $2i$-th processor in a ring, with $0 \leq 2i < P$, send a message that contains proposed exchanges to the $2i+1$-th processor in this ring. Subsequently, each $2i+1$-th processor sends a message that contains not already sent exchanges to the $2i+2$-th processor. Figure 6.2 presents the algorithm for all-to-all broadcasting on a ring with P processors. In this algorithm all indexing is done modulo P. Expression $p!m, m'$ denotes that variables m and m' are consecutively sent to processor p, and $p?m, m'$ denotes that variables m and m' are consecutively received by p.

The time needed for a single communication action depends on the start-up time t_0 and the message length in bytes multiplied by the time t_1 to transfer a byte, where the start-up time t_0 is usually much larger than the time t_1 for transferring a single byte. In the algorithm of Figure 6.2, P times a communication is initiated and messages have length $2M$, except for the first and the last communication action in which a message with length M is sent, where M is the maximum length for encoding proposed exchanges of a processor. So the time complexity of our broadcasting algorithm is $P(t_0 + 2Mt_1) - 2Mt_1$. Any one-to-all broadcasting algorithm on a ring in which a processor cannot send and receive simultaneously requires at least $\frac{P}{2}$ communications. An all-to-all broadcasting algorithm on such a ring requires at least twice as much communications since at a given time stamp only one of two adjacent processors can send a message through which a one-to-all broadcast for some processor proceeds. The total time needed for message transfer is at least $2(P-1)Mt_1$. Thus, our all-to-all broadcasting algorithm on a ring without simultaneous transmission is optimal.

All-to-all broadcasting on a two-dimensional $2x \times 2y$ torus can then be done by simultaneously broadcasting proposed exchanges in each row using the algorithm of Figure 6.2, followed by column-wise broadcasting using the same algorithm. Row-wise and column-wise broadcasting requires $\mathcal{O}(x)$ and $\mathcal{O}(y)$ time, respectively. The message sizes for broadcasting column-wise are of course much larger since each message then contains all proposed exchanges in a given row. The total complexity of all-to-all broadcasting to communicate all proposed exchanges to all processors in a $2x \times 2y$ torus is $\mathcal{O}(x+y)$. This is equal to $\mathcal{O}(\sqrt{P})$ if the network topology is a $\sqrt{P} \times \sqrt{P}$ torus, where P is the number of processors.

The total number of executions of this broadcasting algorithm equals the number of iterations of the parallel algorithm, which is independent of the number of employed processors since in each iteration all profitable exchanges are collected regardless of the number of employed processors.

6.2.1 Computational results

We have implemented the parallel iterative multi-improvement algorithm outlined in the previous section in C on a Parsytec PowerXplorer consisting of 32 processing units that are based on the PowerPC 601 microprocessor. Each processing unit is running the Parix operating system. The interconnection network topology of this message-passing machine is a two-dimensional torus.

We have tested the parallel algorithm on the same instances as the sequential algorithm. Tables 6.3 and 6.4 give the speed-ups obtained on the PowerXplorer machine. The results are averages over ten runs of the algorithm starting from different initial Steiner trees. Each table presents the average running time in seconds $t(s)$ of the sequential algorithm, the average speed-up of the parallel algorithm for a given number of processors P, and the average relative excess ϵ_{avg} of the cost of final solutions over the optimal cost for the D and E series or over the best known upper bounds for the G series. Our sequential algorithm differs from the parallel algorithm since it uses first improvement, whereas the parallel algorithm uses multi improvement as pivoting rule. However, as we have argued in the previous section, first improvement is more efficient in a sequential algorithm than multi improvement, provided that neighborhoods are enumerated as outlined in Section 6.1.2. Therefore, we compute speed-up using first improvement in the sequential algorithm and multi improvement in the parallel algorithm. Multi improvement furthermore ensures that the course of the algorithm is independent of the number of employed processors, and consequently the same final solutions are found regardless of the number of processors.

From the tables we observe that we obtain an acceptable speed-up that scales

P	1	4	8	16	24	32	
instance	$t(s)$	speed up(P)					ϵ_{avg}
d18	8.8	1.8	3.0	5.5	8.8	11.0	7.17
d19	15.4	1.5	2.4	4.7	8.1	9.1	9.12
d20	35.2	2.0	3.4	6.2	9.3	13.0	4.45
e18	65.7	1.6	2.8	5.2	7.1	8.6	8.67
e19	106.7	1.7	3.3	6.0	7.9	9.5	7.51
e20	249.9	1.9	3.6	6.8	9.4	11.6	4.62

Table 6.3: Some computational results for series D and E on PowerXplorer.

P	1	4	8	16	24	32	
instance	$t(s)$	speed up(P)					ϵ_{avg}
g1	111.1	1.5	2.9	4.9	6.2	7.5	0.67
g2	191.3	1.7	3.3	5.8	7.6	9.2	0.46
g3	245.8	2.0	3.7	6.6	8.2	10.5	0.58
g4	196.3	1.9	3.5	6.3	8.5	9.9	0.43
g5	317.6	1.6	3.0	5.2	7.2	8.8	0.47
g6	432.4	1.9	3.5	6.2	8.7	10.7	0.49
g7	705.6	1.8	3.2	5.8	8.4	10.4	0.47
g8	983.3	1.8	3.5	6.1	8.6	11.0	0.46
g9	1458.9	1.8	3.3	6.0	8.7	11.0	0.64
g10	1674.3	1.4	2.7	4.8	6.9	9.0	0.63
g11	2809.1	1.5	2.8	5.2	7.6	9.6	0.72
g12	3715.0	1.3	2.5	4.7	6.8	8.7	0.43
g13	2061.1	1.4	2.6	4.7	6.9	9.0	0.44
g14	3331.3	1.4	2.6	4.7	6.8	8.7	0.74
g16	3623.2	1.6	3.0	5.4	7.6	10.0	0.59
g17	5543.8	1.3	2.5	4.5	6.4	8.2	0.69

Table 6.4: Computational results for series G on PowerXplorer.

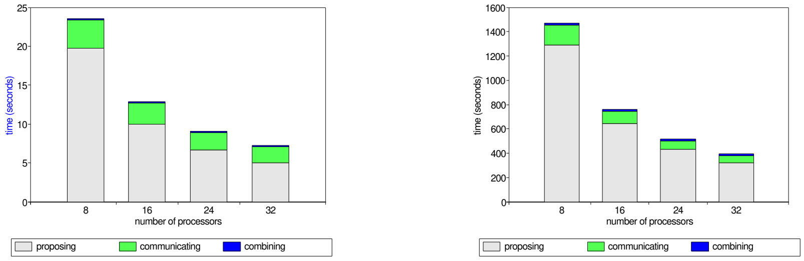

Figure 6.3: Running time profile for the instances g1 (left) and g10 (right).

with the number of processors and the size of the instance. We typically obtain better speed-ups for the random instances in the series D and E than for the Euclidean instances in the series G. Furthermore, we observe that the quality of the final solutions found by the parallel algorithm is equal to the quality that is obtained with the sequential algorithm. This can be explained by the isomorphism of the employed neighborhoods. Moreover, this indicates that the probability of finding given local minima is equal for multi improvement and first improvement algorithms.

Next, we investigate the behavior of our algorithm in more detail. Our parallel algorithm consists of three stages, viz., proposing exchanges, communicating, and combining proposed exchanges. Figure 6.3 gives the total amount of running time spent in each of these stages for two instances and different numbers of processors. We observe that the amount of time spent for proposing exchanges accounts for the bulk of the total running time. This amount of time decreases almost linearly with increasing number of processors. The communication overhead of the parallel algorithm, which includes idle time as well, decreases with increasing number of processors. This implies that the amount of idle time decreases with an increasing number of processors since the time needed for broadcasting increases when more processors are employed. Furthermore, we observe that the amount of time needed to combine and effectuate proposed exchanges is negligible compared to the time needed for proposing and communicating exchanges.

Essential for obtaining a good speed-up with our parallel algorithm is that a large proportion of proposed exchanges is effectuated by the combination function. Figure 6.4 gives for two instances the number of proposed and effectuated exchanges as function of the iteration number. We observe that initially roughly 30% – 40% of the proposed exchanges can be effectuated. This proportion increases rapidly in the course of the algorithm. This shows that the combination function outlined in the previous section is quite effective.

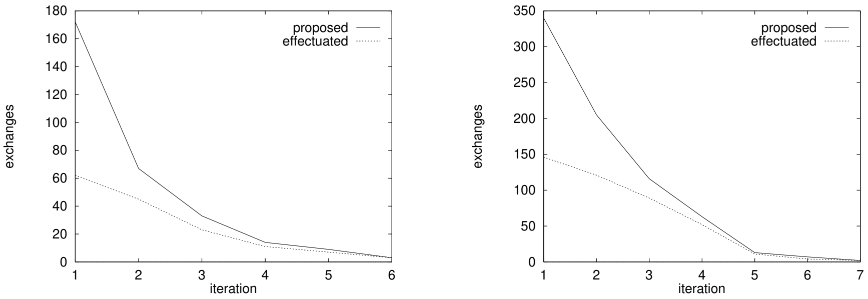

Figure 6.4: Proposed and effectuated exchanges for g1 (left) and g10 (right).

Another interesting observation is that typically only a few iterations of the parallel algorithm are needed to find a local minimum. This is due to the multi improvement pivoting rule that proposes a large number of exchanges in each iteration, many of which can be effectuated. A small number of iterations implies that only a few all-to-all broadcasts occur in the course of the algorithm, since in each iteration only a single all-to-all broadcast is necessary. This accounts for the small communication overhead of the algorithm, which enables a good speed-up of our parallel local search algorithm for the STPG.

In this chapter we have discussed a parallel local search algorithm for the STPG based on multiple step parallelism. Our parallel algorithm uses a new pivoting rule for proposing exchanges which is better suited for a parallel algorithm. Furthermore, we have presented an optimal polynomial-time algorithm to implement the combination function. Computational results show that good speed-ups can be obtained without any loss in quality of final solutions, which shows that the proposed algorithm is one of the most effective algorithms to handle large instances of the STPG.

7

Scheduling

Scheduling problems arise in situations where a set of activities has to be processed using a limited number of resources. Applications can be found in production planning, time tabling, or real-time system controlling. An introduction to scheduling can be found in [Pinedo, 1995]. Job shop scheduling is an important model in scheduling theory, which serves as a testbed for new algorithmic ideas and provides a starting point for the more complicated practically relevant scheduling models [Lawler, Lenstra, Rinnooy Kan & Shmoys, 1993].

In this chapter we study the applicability of multiple-step parallelism in local search for the job shop scheduling problem. Furthermore, we present a local search algorithm for a generalization of the job shop scheduling problem in which machines are capable of processing more than one operation at a time and in which operations may require machine sets for processing and may have several alternative machine sets on which they can be processed. Finally, we study parallel local search approaches to this problem.

7.1 Job shop scheduling

An instance of the job shop scheduling problem (JSSP) consists of a set of jobs and a set of machines. Each machine can handle at most one job at a time. Each job consists of a chain of operations, which need to be processed in that order during an uninterrupted time period of a given length on a given machine. The problem is to find a schedule, which is defined as an assignment of the operations to time intervals, such that the total length of the schedule is minimal [French, 1982]. More formally, the problem can be defined as follows.

Definition 7.1 (JSSP). Given are a set J of jobs, a set M of machines, and a set O of operations. For each operation $a \in O$, there is a job $j(a) \in J$ to which it belongs, a machine $m(a) \in M$ on which it must be processed, and a processing time $d(a) \in \mathbb{N}$. There is a binary relation $\prec$ on O that decomposes O into chains corresponding to the jobs; more specifically, if $a \prec b$, then $j(a) = j(b)$ and there is no $c \in O \setminus \{a, b\}$ with $a \prec c$ and $c \prec b$. The problem is to find a non-negative start time $s(a)$ for each operation $a \in O$ such that the schedule length

$$\max_{a \in O}(s(a) + d(a))$$

is minimal, subject to

(1) $s(b) - s(a) \geq d(a)$ if $a \prec b$, and
(2) $s(b) - s(a) \geq d(a) \ \vee \ s(a) - s(b) \geq d(b)$ if $m(a) = m(b)$,

for all $a, b \in O$. □

The constraints corresponding to (1) are job precedence constraints, and those corresponding to (2) are machine capacity constraints.

The job shop scheduling problem is NP-hard [Garey & Johnson, 1979] and local search, in particular tabu search [Nowicki & Smutnicki, 1995], has proved to belong to the best approximation algorithms for it [Vaessens, 1995]. To apply local search to the job shop scheduling problem, it is most appropriate to use the disjunctive graph representation of Roy & Sussmann [1964].

Definition 7.2. A JSSP instance is represented by a vertex weighted disjunctive graph $G = (V, A, E)$ with vertex set $V = O$, arc set $A = \{(a, b) \mid a \prec b\}$, and edge set $E = \{\{a, b\} \mid m(a) = m(b)\}$. A weight $d(a)$ is associated with each vertex a in V. A feasible solution is represented as a minimal subset E' of orientations of the edges in E, such that E' gives for each machine a complete order of the operations that have to be processed on it, and such that the resulting digraph $D = (V, A \cup E')$ is acyclic. The start time of an operation $v \in V$ is the length of the longest path up to and without v, and the cost $f(D)$ of a feasible solution D is the longest path in D. □

The (directed) arcs represent the job precedence constraints and the (undirected) edges represent the machine capacity constraints. The length of a path is the sum of the weights of vertices on this path. A digraph D corresponding with a feasible solution contains only oriented edges between immediate successors and predecessors on a machine, since the orientation of the remaining edges in E is uniquely determined by E'. So each vertex in D has at most two incoming edges and at most two outgoing edges, viz., at most one successor and at most one predecessor in its job and on its machine. Given any such orientation E', we can determine feasible start times by setting each start time $s(a)$ equal to the length of a longest path in D up to a. The cost of a solution with $|O|$ operations can be computed in $\mathcal{O}(|O|)$ time with Bellman's labeling algorithm [Lawler, 1976], since the degree of each vertex in D is at most four. The problem is then to find an orientation of the edges in E that minimizes the longest path in D.

Definition 7.3. Given are a digraph $D = (V, E)$ and a vertex $a \in V$. Then, the immediate successor and predecessor on the machine and in the job of operation a are given by sm_a, pm_a, sj_a, and pj_a, respectively. $st(a, D)$ is the length of the longest path in D up to and without a. $rt(a, D)$ is the length of the longest path in D starting from and including a. $l(a, D)$ is the length of the longest path in D through a. The start time $s(a)$ is equal to $st(a, D)$. □

Most neighborhoods for the job shop scheduling problem are based on reversing machine capacity arcs in the digraph representing a schedule—that is, reversing the order in which operations are processed on a machine. Van Laarhoven, Aarts & Lenstra [1992] give the following 1-opt neighborhood $\mathcal{N}_1$. Given a solution $s \in \mathcal{S}$ represented by a digraph D, a neighboring solution is obtained by selecting two operations a and b, with $j(a) \neq j(b)$, that are adjacent on some machine m and for which the arc (a, b) is on a longest path in D, and reversing (a, b). More formally, this leads to the following definition.

Definition 7.4. Given are a schedule $s \in \mathcal{S}$ represented by digraph $D = (V, E)$, and $a, b \in V$ with $b = sm_a$. Let the exchange function τ be given by $\tau(D, a, b) = (V, (E \setminus \{(a, b), (pm_a, a), (b, sm_b)\}) \cup \{(pm_a, b), (b, a), (a, sm_b)\})$. Then, the neighborhood $\mathcal{N}_1(D) = \{\tau(D, a, sm_a) \mid a, sm_a \in V \wedge (a, sm_a)$ on a longest path in $D\}$. □

The size of the neighborhood $\mathcal{N}_1(D)$ for any digraph D is $\mathcal{O}(|O|)$. Van Laarhoven, Aarts & Lenstra [1992] showed the following properties for this neighborhood.

(1) The reversal of (a, sm_a) on a longest path in D results in an acyclic digraph D' that again corresponds to a feasible solution s'.

(2) Reversals of arcs on a longest path are the only arc reversals that can —but need not— result in a digraph with a shorter longest path. Furthermore, a reversal of an arc not on a longest path can lead to an infeasible solution.

(3) $\mathcal{N}_1$ is sufficiently connected.

Furthermore, we have the following result, which generalizes an observation made by Taillard [1994].

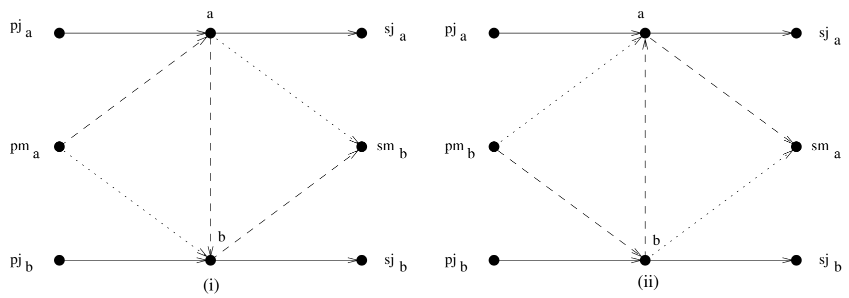

Figure 7.1: Part of a digraph before (i) and after (ii) reversal of an arc (a,b). Dotted arcs are redundant machine capacity arcs.

Theorem 7.1. *Given are a digraph D and functions st and rt as in Definition 7.3. Let digraph $D' = \tau(D,a,b)$, with $b = sm_a$ and (a,b) on a longest path in D, and let $mx(D',a,b) = \max\{l(a,D'), l(b,D')\}$. Furthermore, let l_2 be the length of the second longest path in D, which can be equal to $f(D)$, and l' the length of the longest path in D not through a or b. Then,*

$$\begin{array}{lll} (1) & mx(D',a,b) \geq f(D) & \Rightarrow f(D') = mx(D',a,b), \\ (2) & l_2 < mx(D',a,b) < f(D) & \Rightarrow f(D') = mx(D',a,b), \text{ and} \\ (3) & l_2 \geq mx(D',a,b) & \Rightarrow f(D') = \max\{l', mx(D',a,b)\}. \end{array}$$

Moreover, $mx(D',a,b)$ is computable in constant time.

Proof. Only paths through a or b in D may no longer exist in D' as a consequence of the reversal of (a,b). So in case (1) when $mx(D',a,b) \geq f(D)$, then $f(D') = mx(D',a,b)$ because all paths in D' not through a or b have lengths of at most $f(D)$. In case (2) when $l_2 < mx(D',a,b) < f(D)$, then $f(D') = mx(D',a,b)$ because all paths in D' not through a or b have lengths of at most l_2 with $l_2 < mx(D',a,b)$, and paths in D' through a or b have lengths of at most $mx(D',a,b)$ and at least one path has length $mx(D',a,b)$. In case (3) when $mx(D',a,b) \leq l_2$, then $f(D') = \max\{l', mx(D',a,b)\}$ because the longest path in D' is the longest path through a or b in D' or the longest path not through a or b in D', which is equal to that in D. From Figure 7.1 we infer that $st(b,D') = \max\{st(pm_a,D) + d(pm_a), st(pj_b,D) + d(pj_b)\}$ and $st(a,D') = \max\{st(b,D') + d(b), st(pj_a,D) + d(pj_a)\}$. Also, $rt(a,D') = \max\{rt(sj_a,D), rt(sm_b,D)\} + d(a)$, and $rt(b,D') = \max\{rt(a,D'), rt(sj_b,D)\} + d(b)$. Hence, $l(a,D') = st(a,D') + rt(a,D')$ and $l(b,D') = st(b,D') + rt(b,D')$. This gives $mx(D',a,b) = \max\{l(a,D'), l(b,D')\}$. So $mx(D',a,b)$ is given by expressions that are all computable in constant time. □

7.1.1 Parallel local search for the job shop scheduling problem

In this section we outline a local search algorithm with multiple-step parallelism for the job shop scheduling problem. Other approaches discussed in Chapter 3 to introduce parallelism in local search are also valuable since most tabu search algorithms for the job shop scheduling problem use best improvement as pivoting rule, in which case good speed-ups can be achieved with single-step parallelism, provided that neighborhoods are sufficiently large; see Section 3.2.1. Moreover, probabilities of finding optimal solutions using tabu search fit well with exponential distributions [Taillard, 1994], and in that case good speed-ups can be obtained with multiple independent walks; see Chapter 4. Multiple-step parallelism can be incorporated in the above approaches, which may result in even more scalable hybrid approaches.

In the previous section we have argued that only reversals of arcs on a longest path can lead to digraphs with shorter longest paths. Furthermore, infeasible digraphs can be constructed when arcs not on a longest path are reversed. So a parallel local search algorithm must be based on reversals of arcs on longest paths. However, a reversal of an arc on a longest path leads to a digraph that has a different longest path, and consequently only a single arc can be reversed on a given longest path.

Our parallel local search algorithm is based on a distributed neighborhood structure in which local neighborhoods for each processor p comprise arc reversals on the p-th longest path in a digraph.

Definition 7.5. Let p be an integer between 1 and P, and $D = (V, E)$ a digraph. Then, for a vertex $c \in V$, $st_p(c, D)$ gives the length of the p-th longest path in D up to and without c, and $rt_p(c, D)$ gives the length of the p-th longest path in D from and including c. Furthermore, $\lambda_p(D)$ denotes the p-th longest path in D. Let $a_p, b_p \in V$ with (a_p, b_p) on $\lambda_p(D)$. Then, we recursively define D_p as $D_1 = D$, and $D_{p+1} = \tau(D_p, a_p, b_p)$ for $1 \leq p < P$. □

Note that $st_p(c, D)$ and $rt_p(c, D)$ are descending in p and that the paths specified by λ are not necessarily disjoint. D_p is the graph obtained by effectuating the reversals on the paths $\lambda_q(D)$ with $1 \leq q < p$. Theorem 7.1 suggests that $\lambda_p(D)$ is often the longest path in D_p. This gives rise to the following local search algorithm with multiple-step parallelism for the job shop scheduling problem. Given are a digraph D and P processors.

(1) Compute the P longest paths in D and assign $\lambda_p(D)$ to processor p for $1 \leq p \leq P$.

(2) Processor p evaluates arc reversals $\tau(D, a_p, b_p)$ with (a_p, b_p) on $\lambda_p(D)$. A reversal is proposed if $mx(D', a_p, b_p) < \text{length}(\lambda_p(D))$.

(3) Effectuate the proposed arc reversals by successively constructing each digraph D_{p+1}, which is obtained by reversal of (a_p, b_p) in D_p, provided that D_{p+1} is feasible.

(4) Replace D with D_ρ for ρ such that $\min_{1\leq p\leq P} f(D_p) = f(D_\rho)$. Repeat the above steps until no improvement is found.

The above algorithm is based on the following distributed neighborhood $\mathcal{D}$. The distribution structure of $\mathcal{D}$ assigns to each processor p, with $1 \leq p \leq P$, the p-th longest path in a digraph D. The local neighborhood for a processor, specified by the domain distribution of $\mathcal{D}$, is obtained by reversing arcs on the path assigned to a processor. Note that the P longest paths in an acyclic digraph can be computed by a generalization of Bellman's labeling algorithm [Lawler, 1976] that computes the longest path in an acyclic digraph.

An arc (a_p, b_p) whose reversal is proposed may not on a longest path in the graph D_p if the longest path through (a_p, b_p) in D also passes through (a_q, b_q) in D whose reversal is proposed by processor q with $1 \leq q < p$. This implies that effectuating $\tau(D_p, a_p, b_p)$ might lead to an infeasible solution. Furthermore, effectuating a proposed exchange $\tau(D_p, a_p, b_p)$ may result in a digraph with larger cost if st or rt needed to compute $mx(D_{p+1}, a_p, b_p)$ are not equal to the values of st or rt used to compute $mx(D, a_p, b_p)$, i.e., reversals of other edges have resulted in a longer path to or from a vertex that precedes or succeeds a_p or b_p in D_p. In order to deal with such situations, it has to be checked whether proposed exchanges lead to feasible digraphs with lower costs, before they are effectuated. This is done by the combination function of the distributed neighborhood $\mathcal{D}$ as follows. Each processor p effectuates proposed reversals (a_q, b_q) with $1 \leq q \leq p$ and checks whether the resulting digraph D_p defines a feasible schedule with lower cost than the current schedule. The digraph D' that is returned by the combination function is the digraph D_p with minimal cost found in this way. The time complexity of this combination function is $\mathcal{O}(P + |O|)$, since each digraph D_p is constructed in $\mathcal{O}(P)$ time (cf. Definition 7.5) and the time complexity to check whether D_p is a feasible schedule with lower cost is $\mathcal{O}(|O|)$.

Discussion. We have implemented a sequential algorithm that simulates an algorithm with multiple-step parallelism based on the distributed neighborhood $\mathcal{D}$ for the job shop scheduling problem. Experiments show that the maximum speed-up is limited since the number of proposed exchanges that can be effectuated by the combination function is typically quite small, e.g., for instances with up to 300 operations the maximum speed-up is at most four. Although this picture is slightly better for larger instances, in particular for instances with many jobs and few machines, it is fair to say that the application of multiple-step parallelism to local search for job shop scheduling does not give satisfactory results.

7.2 Resource-constrained scheduling

In the job shop scheduling problem it is assumed that each machine can process only one operation at a time and that each operation is processed by a prespecified machine. However, in many practical situations machines are capable of processing more than one operation at a time and processing of operations may require sets of machines for which several alternative choices exist.

In this section we present a scheduling model that incorporates the above extensions. In this model each operation can be processed by *resource sets* for which *alternatives* may exist, and an operation's processing time on a particular resource may depend on the resource set by which it is processed and on the resource at hand. Furthermore, resources are able to process several operations simultaneously—that is, resources are *multiple capacitated.* For this purpose, each resource is given an integer *capacity*, and each operation is given an integer *size* which may depend on the resource by which it is processed. A resource can simultaneously process only those sets of operations whose sizes do not exceed the capacity of the resource. Finally, arbitrary precedences between operations are allowed. The objective is to minimize the maximum completion time over all operations. Other extensions, such as resource availability constraints that prohibit usage of resources during given time intervals, and minimum delay time constraints that force time gaps between operations, can easily be integrated in this model. One important type of constraint we exclude from our model is the maximum delay constraint that binds operations in time by stating that an operation must be started within a given time interval after another operation is completed.

Our model generalizes the resource-constrained single project scheduling problem, which is one of the few problems in the area of more general scheduling problems, an area usually referred to as resource-constrained scheduling, that received considerable attention in the literature [Lawler, Lenstra, Rinnooy Kan & Shmoys, 1993]. Our model also generalizes the models presented by Hurink, Jurisch & Thole [1994] and Vaessens [1995]. It resembles the model of Nuijten [1994] except for the possibility of adding additional constraints by means of a constraint language. Our resource-constrained scheduling problem (RCSP) is described as follows.

Definition 7.6 (RCSP). Instances of the resource-constrained scheduling problem consist of a set of operations O and a set of resources R. Furthermore, $fr : O \to \mathcal{P}(\mathcal{P}(R))$ gives for each operation a set of feasible resource sets, $cp : R \to \mathbb{N}$ gives for each resource its capacity. The functions $pt : O \times R \times \mathcal{P}(R) \mapsto \mathbb{N}$ and $sz : O \times R \times \mathcal{P}(R) \mapsto \mathbb{N}$ give for each operation $o \in O$ and resource $r \in R$ in a resource set in $fr(o)$ the processing time and size, respectively, required by o for processing on r. A binary relation $\prec$ defines a partial order on O.

A *schedule* is a pair (ra, st) where $ra : O \to \mathcal{P}(R)$ is mapping that gives for each operation $o \in O$ a resource assignment $ra(o) \in fr(o)$ and $st : O \to \mathbb{N}$ is a mapping that gives for each operation o a start time such that for all $o, o' \in O$ with $o \prec o'$

$$\max_{r \in ra(o)} st(o) + pt(o, r, ra(o)) \leq st(o'), \tag{7.1}$$

and for all $r \in R$ and $t \in \mathbb{N}$

$$\sum_{o \in O | r \in ra(o) \,\wedge\, st(o) \leq t < st(o) + pt(o,r,ra(o))} sz(o, r, ra(o)) \;\leq\; cp(r). \tag{7.2}$$

The problem is to find a schedule (st, ra) such that its length, or *makespan*,

$$\max_{o \in O \,\wedge\, r \in ra(o)} st(o) + pt(o, r, ra(o))$$

is minimized. □

We use $ct(o, r, ra(o))$ to abbreviate $st(o)+pt(o, r, ra(o))$. The constraints given in (7.1) are called precedence constraints and those in (7.2) are capacity constraints. The RCSP is NP-hard since it contains the job shop scheduling problem as a subproblem.

A schedule is *left-justified* if it is not possible to complete any operation earlier without changing the resource assignment and the order of operations on resources that is determined by their start times. Clearly, there is a left-justified optimal schedule that can be obtained from an optimal schedule by adjusting its start times such that each operation is started as soon as possible without changing the order of operations on resources. So we can restrict the solution space to left-justified schedules, since such a solution space still contains an optimal solution.

7.2.1 Neighborhoods for the resource-constrained scheduling problem

Most of the literature on resource-constrained scheduling concentrates on exact algorithms or constructive heuristics [Slowinski & Weglarz, 1989; Blazewicz, Ecker, Schmidt & Weglarz, 1994]. Local search algorithms belong to the best approximation algorithms for job shop scheduling, and therefore one may expect that local search also gives good-quality solutions for more general scheduling problems such as the resource-constrained scheduling problem.

Only a few attempts to apply local search in this problem area are known to us. Sampson & Weiss [1993] present a local search algorithm for a, not explicitly stated, resource-constrained scheduling problem in which irregular cost functions are allowed. In the neighborhood they propose, neighbors are obtained by incrementing of decrementing the delay from the earliest possible start time of any operation. Yoo, Yang & Ignizio [1995] consider a model in which operations may require resource sets for which alternatives exist and in which resources are multiple capacitated. The objective is to minimize the makespan. They propose a neighborhood in which neighbors are obtained by pairwise interchanging any two operations on a machine. A drawback of this neighborhood is the large proportion of non-improving neighbors. Leon & Balakrishnan [1995] consider a similar model extended with the possibility to utilize other cost functions, such as mean tardiness, but without the possibility of alternative resource sets for processing an operation. They represent solutions by a vector that associates values with operations. Schedules are constructed using some heuristic that utilizes this vector. Neighbors are obtained by adapting the values in this vector. A drawback of this approach is that solutions are manipulated only indirectly without exploiting the problem structure. Hurink, Jurisch & Thole [1994] and Dauzère-Pérès & Paulli [1995] present local search algorithms for the job shop scheduling problem with resource alternatives, which is a subclass of the RCSP.

To apply local search to the RCSP, we represent schedules by feasible resource assignments for operations and by specifying the order in which operations are started on resources. Start times of operations can then be determined by computing a left-justified schedule.

Definition 7.7. Let an RCSP instance be given. A schedule (ra, st) is represented by resource assignment ra and an acyclic directed graph $G = (O, E)$ with $E = \{(o, o') \mid o \prec o'\} \cup E'$, where $E' \subseteq \{(o, o') \mid ra(o) \cap ra(o') \neq \emptyset \wedge st(o) \leq st(o')\}$ such that E' is a maximal edge set that gives for each resource r a complete order for operations o with $r \in ra(o)$. □

Given an acyclic graph G as specified in Definition 7.7, start times of operations in a left-justified schedule corresponding with G can be computed using the algorithm of Figure 7.2 that is based on Bellman's labeling algorithm [Lawler, 1976]. In this algorithm $S(o)$ is the set of immediate successors of an operation o in G. This set contains operations o' with $(o, o') \in E$ for which no o'' exists with $(o, o'') \in E$ and $(o'', o') \in E$. The complexity of this algorithm is $\mathcal{O}(|O|(CK + J))$, where C is the maximum capacity of any resource, which is an upper bound on the number of operations that can be processed simultaneously on a resource, K the maximum cardinality of a feasible resource set of any operation, and J the maximum number of successors of any operation in the precedence graph specified by $\prec$.

A neighborhood structure for the RCSP must be capable of changing resource assignments of operations and modifying the order in which operations are processed by resources. This can be done by an exchange function that removes operations from a schedule and reinserts them in such a way that feasible schedules

```
proc Compute_Start_Time
  begin
    for $o \in O$ do $n(o) := 0$; $st(o) := 0$ od
    for $o \in O$ do for $o' \in S(o)$ do $n(o') := n(o') + 1$ od od
    $Y := \{o \in O \mid n(o) = 0\}$;
    while $Y \neq \emptyset$ do
      $o \in Y$; { $st(o)$ satisfies (7.1) }
      increase $st(o)$ until (7.2) is met and $st(o') \leq st(o)$ for $(o', o) \in E$;
      for $o' \in \{o'' \mid o \prec o''\}$ do update $st(o')$ od
      for $o' \in S(o)$ do $n(o') := n(o') - 1$;
        if $n(o') = 0$ then $Y := Y \cup \{o'\}$ fi
      od
      $Y := Y \setminus \{o\}$;
    od { $st(o)$ is start time of $o \in O$ subject to (7.1) and (7.2) }
  end
```

Figure 7.2: An algorithm to compute start times of operations.

are constructed in which either resource assignments have been changed or orders of operations on resources have been modified. To reduce the size of such a neighborhood it is desirable to exclude exchanges that cannot result in neighbors with lower cost, as is done for the job shop by restricting exchanges to those that modify the order of operations on a longest path. For the RCSP *critical* operations, operations whose removal from a schedule may affect a schedule's makespan, are defined as follows.

Definition 7.8. Given are an RCSP instance and a feasible schedule (ra, st) for it, represented by $G = (O, E)$. Add a dummy vertex $o^\star$ to G that succeeds all operations in O, i.e., $o \prec o^\star$ for all $o \in O$. Define a mapping $\lambda : O \to \mathcal{P}(O)$ that gives for each operation o a set of operations whose removal from G may affect the start time of o, as follows. $\lambda(o) = \emptyset$ if $\{(o', o) \in E \mid o' \not\prec o\} = \emptyset$, otherwise $\lambda(o) = \cup_{o' \in O \mid (o', o) \in E \,\land\, R(o, o')} (\lambda(o') \cup \{o'\})$ where $R(o, o') \equiv$

$$\exists_{r \in ra(o) \cap ra(o')} st(o) \leq ct(o', r, ra(o')) \ \lor$$

$$(o' \prec o \ \land \ st(o) = \max_{r \in ra(o')} ct(o', r, ra(o'))).$$

Operation o is *critical* in G, if and only if $o \in \lambda(o^\star)$. □

If we consider RCSP instances in which all operation sizes and resource capacities are one, then the set of critical operations defined by Definition 7.8 coincides with the set of operations on a longest path in a graph, but in general critical operations are not situated on a single path in a graph.

Removal of non-critical operations does not affect the makespan of a schedule, and therefore removal and subsequent reinsertion of non-critical operations cannot result in lower-cost schedules. This gives rise to the following neighborhood in which only critical operations are reinserted. First, we remark that an operation o is an *active* predecessor of o' if o is just finished or still being processed on a resource when o' is started on this resource. Neighbors are obtained by removing a critical operation from a schedule and inserting it before an active predecessor on a resource while maintaining the same resource assignment, or by choosing a new resource set for this operation and reinserting the operation on the resources in this set. Formally, this neighborhood is defined as follows.

Definition 7.9. Let $G = (O, E)$ be a graph that represents a schedule (ra, st). Let τ_1, τ_2 be exchange functions such that $\tau_1(G, o', o) = (O, E')$ is an acyclic graph with $(o, o') \in E'$ and $(o', o) \in E$ that represents a schedule (ra, st'), and such that $\tau_2(G, o, a)$ is an acyclic graph corresponding with a schedule (ra', st') with $ra'(o) = a$ for $a \in fr(o)$ and $a \neq ra(o)$. Then, the neighborhood $\mathcal{N}_1$ is defined by

$$\begin{aligned}\mathcal{N}_1(G) = \{\tau_1(G, o', o) \mid (o', o) \in E \wedge \exists_{r \in ra(o) \cap ra(o')}\, st(o) \leq ct(o', r, ra(o'))\\ \wedge\, o \text{ is critical in } G\} \cup \{\tau_2(G, o, a) \mid a \in fr(o) \wedge o \text{ is critical in } G\}.\end{aligned}$$

□

Next, we show that this neighborhood is sufficiently connected, which implies that an optimal solution can be reached starting from an arbitrary solution by a finite sequence of exchanges. For this we need the following lemma.

Lemma 7.1. *Let (ra, st) be a non-optimal schedule represented by G, and let (ra^*, st^*) be an optimal schedule represented by $G^* = (O, E^*)$. Then, the set*

$$\begin{aligned}&\{o \in O \mid o \textit{ is critical in } G \wedge (\, ra(o) \neq ra^*(o) \ \vee\\ &\exists_{o' \in O} \exists_{r \in ra(o) \cap ra(o')}\, st(o) \leq ct(o', r, ra(o')) \wedge (o', o) \in E \wedge (o, o') \in E^* \,)\}\end{aligned}$$

is non-empty.

Proof. Assume that for all critical operations o in G holds that $ra(o) = ra^*(o)$ and for all $(o', o) \in E$ with $st(o) \leq ct(o', r, ra(o'))$ and $r \in ra(o')$ holds $(o', o) \in E^*$. Then, all edges in G that contribute to $st(o^\star)$ are also included in G^*, and since other edges (o', o) in G^* can only increase $st^*(o^\star)$, we have $st^*(o^\star) \geq st(o^\star)$. But we already know that $st^*(o^\star) \leq st(o^\star)$ and therefore G is optimal. □

Theorem 7.2. *The neighborhood $\mathcal{N}_1$ is sufficiently connected.*

Proof. Let (ra, st) be a non-optimal schedule represented by G, and (ra^*, st^*)

an optimal schedule represented by $G^* = (O, E^*)$. We construct a sequence of exchanges leading from G to G^* via G_i as follows.

(1) $G_0 = G$.

(2) If there is a critical operation o in G_i with $ra(o) \neq ra^*(o)$, then $G_{i+1} = \tau_2(G_i, o, ra^*(o))$.

(3) If for all critical operations o $ra(o) = ra^*(o)$, then $G_{i+1} = \tau_1(G_i, o', o)$, where o is a critical operation in G_i and $o' \in O$ is such that $(o', o) \in E_{G_i}$ and $\exists_{r \in ra(o) \cap ra(o')}\, st(o) \leq ct(o', r, ra(o'))$ and $(o, o') \in E^*$.

It holds that $G_{i+1} \in \mathcal{N}_1(G_i)$. According to Lemma 7.1 operations o or o' that meet the conditions in (2) or (3) always exist, unless G_i is optimal. Furthermore, for (3) holds that edges $(v', v) \in G_i$ exist for which $(v, v') \in G^*$ and $(v, v') \in G_{i+1}$. Reversal of (v', v), in addition to reversal of (o', o), is necessary for achieving acyclicity of G_{i+1}. These edges (v', v) exist because if all edges $(v', v) \in G_i$ on a path from o' to o in G_i also occur in G^*, then G^* would contain a cycle since $(o, o') \in G^*$.

It remains to show that this sequence is finite. For this, define $A(G, G^*) = \{o \in O \mid ra(o) \neq ra^*(o)\}$ and $R(G, G^*) = \{(o', o) \in E \mid (o, o') \in E^*\}$ for a graph $G = (O, E)$. It holds that $|A(G_{i+1}, G^*)| < |A(G_i, G^*)|$ if G_{i+1} is constructed in (2) and that $|R(G_{i+1}, G^*)| < |R(G_i, G^*)|$ if G_{i+1} is constructed in (3). So there exists a $k_1 \leq |O| \cdot A(G_0, G^*)$ such that the condition in (2) can no longer be satisfied for all $k > k_1$ in which case G_{k+1} is constructed from G_k by applying τ_1 in (3) only. Moreover, for some $k_2 \leq k_1 + |R(G_{k_1}, G^*)|$ holds that the condition in (3) can also no longer be satisfied. According to Lemma 7.1 then holds that G_{k_2} is an optimal schedule. □

A neighbor in $\mathcal{N}_1$ is any acyclic graph that can be obtained by placing a critical operation before an active predecessor or by changing its resource assignment. Several feasible graphs may be constructed in which a given resource set is assigned to a critical operation. All these graphs are comprised in $\mathcal{N}_1$, which results in large neighborhood sizes. Moreover, determining feasibility of a graph obtained by reversing some edges or by changing the resource assignment of a critical operation cannot be done in constant time, as opposed to the job shop in which graphs obtained by reversing edges on a longest path are always feasible. These issues limit the practical usefulness of $\mathcal{N}_1$. Therefore, we restrict the neighborhood $\mathcal{N}_1$ such that it contains less neighbors, which can be constructed efficiently.

Definition 7.10. Given is a graph $G = (O, E)$ representing a schedule (ra, st). An *insert neighbor* $G' = (O, E')$ with given resource assignment ra' is constructed as follows. Remove operation $a \in O$ from G and remove all edges incident with a, resulting in a graph $G^- = (O \setminus \{a\}, E^-)$. Choose an operation $b \in O$. Determine, for all $r \in ra'(a)$, operation $o_r \in O$ with $r \in ra'(o_r)$ such that $st(o_r) < st(b)$ and $st(o') \leq st(o_r)$ for all $o' \in O$ for which $r \in ra(o')$ and $st(o') < st(b)$. Add to G^- edge (o_r, a), edges (a, o') for all $(o_r, o') \in E^-$, and edges (o', a) for all $(o', o_r) \in E^-$, i.e., o_r directly precedes a on resource r. □

Theorem 7.3. *An insert neighbor G' as specified in Definition 7.10 is acyclic and represents a feasible schedule.*

Proof. Let a, b, o_r, G, G^-, G' be given by Definition 7.10. First note that the existence of a path in G from $o \in O$ to $o' \in O$ in G implies that $st(o) \leq st(o')$ by definition of a graph G. Let o'_r denote a direct successor of operation o_r in G. Assume that G' contains a cycle. This cycle must pass through edge $(o_{r'}, a)$ and (a, o'_r) for some $r, r' \in R$, since G and thus G^- are acyclic. So there must be a path from $\{o_r \mid r \in R\}$ to $\{o'_r \mid r \in R\}$ in G^-. However, no such path can exist, because for all $r, r' \in R$ holds $st(o'_r) \geq st(b) > st(o_r)$ which implies that no such path exists in G and therefore neither in G^-. □

Insert neighbors G' specified by Definition 7.10 can be computed in $\mathcal{O}(K \log L)$ time, where K is the maximum cardinality of a feasible resource set of any operation and L the maximum number of operations assigned to any resource, because operations o_r that directly precede a in G' can be found using binary search.

Using the above observations, we can define a neighborhood $\mathcal{N}'_1$ that is similar to $\mathcal{N}_1$, except for the exchange functions τ_1 and τ_2 that construct only insert neighbors as given in Definition 7.10. $\mathcal{N}'_1$ has a much smaller computational complexity for verifying local optimality than $\mathcal{N}_1$, but the price we pay for this is that $\mathcal{N}'_1$ is no longer sufficiently connected, as can be shown by an example. The size of the neighborhood $\mathcal{N}'_1(G)$ for a graph G is $\mathcal{O}(\lambda(o^\star)F)$, where F is an upper bound on the number of feasible resource sets for any operation.

The effectiveness of the neighborhood $\mathcal{N}'_1$ can be further improved by restricting the solution space to active schedules. A schedule is *active* if it is not possible to complete any operation earlier without changing the resource assignment of operations or postponing the completion time of any of the other operations. The construction of active schedules is based on the following observation. If $(o, o') \in E$ for a graph G, then according to Definition 7.7 $st(o) \leq st(o')$. If for some $t < st(o)$ the precedence constraints (7.1) and capacity constraints (7.2) for o' are satisfied without postponing o when o' is started on t, then reversing (o, o') does not delay any operation and can only decrease the makespan of G. Active schedules can be constructed with the algorithm of Figure 7.2 by satisfying only the precedence constraints (7.1) while assigning the earliest possible start time to operations at which (7.2) is satisfied, in the order imposed by E. So while computing a schedule's makespan it is possible to construct, with some additional computational overhead, an active schedule by reversing some edges, which may have a shorter makespan than the original graph. The time complexity to compute the makespan of such an active schedule, derived from G, is $\mathcal{O}(|O|(KL + J))$, where K is the maximum cardinality of a feasible resource set of any operation, L is the maximum number of operations assigned to any resource, and J is the maximum number of successors of any operation in the precedence graph.

Construction of active neighboring schedules may give a larger cost decrease in a single exchange, but it also results in a larger computational complexity for each local search step.

7.2.2 Tabu search for the resource-constrained scheduling problem

Tabu search is discussed in Section 2.2. In this section, we outline how tabu search can be adapted to the RCSP. The tabu list can be implemented for the neighborhood $\mathcal{N}_1'$ as follows. Introduce mappings $ts : O \to O \times \mathbb{N}$ and $ta : O \times \mathcal{P}(R) \mapsto \mathbb{N}$. Let I denote the total number of steps performed by a tabu search algorithm and T the tabu tenure. Then, a proposed reinsertion $\tau_1'(G, o', o)$ is tabu, if $ts(o) = (o', i)$ with $i + T \geq I$, and a proposed resource assignment $\tau_2'(G, o, a)$ is tabu, if $ta(o, a) = i$ with $i + T \geq I$. After effectuating a proposed reinsertion $\tau_1'(G, o', o)$, $ts(o')$ is set equal to (o, I), and if a resource assignment $\tau_2'(G, o, a)$ is effectuated, $ta(o, a)$ is set equal to I. Note that the memory requirements to implement the tabu list are quite limited, viz., $\mathcal{O}(|O|F)$ where F is the maximum number of feasible resource sets for any operation.

The basic tabu search algorithm, outlined above, can be extended in several ways; see [Glover, Taillard & De Werra, 1993]. Nowicki & Smutnicki [1995] present a tabu search algorithm for the job shop scheduling problem that is one of the most effective approximation algorithms for this problem. In this algorithm restarting from one of the five best solutions found so far takes place if no improvement of the overall best solution is found for a given number of steps. We incorporate this approach for intensification of the search in our tabu search algorithm for the RCSP as follows.

(1) Perform the basic tabu search step by selecting a non-tabu neighbor of the current solution, while storing the B lowest-cost local minima G_i in a list $\mathcal{L}$. Also the tabu lists $tl(G_i)$ associated with solutions G_i are stored.

(2) Repeat step (1) until no solution with lower cost than the overall best is found for a given number of steps L.

(3) Select the lowest-cost solution G_i in $\mathcal{L}$ for which the set of non-tabu neighbors in $\mathcal{N}_1'(G_i) \setminus W(G_i)$ is non-empty. Select $G' \in \mathcal{N}_1'(G_i) \setminus W(G_i)$ and add it to $W(G_i)$. Go to step (1) using G' as current solution and tabu list $tl(G_i)$. If no such G' exists for any solution G_i in $\mathcal{L}$, the algorithm stops.

In the above algorithm, the search is intensified around previously found good solutions by exploring different paths from there. Memory requirements for this approach can be reduced by storing the exchanges that lead to the examined neighbors in $W(G_i)$, instead of the actual neighbors themselves.

7.2.3 Computational results

We have implemented the tabu search algorithm with the neighborhood $\mathcal{N}_1'$ outlined in the previous sections. The parameters of the tabu search algorithm are chosen as follows. The tabu tenure is randomly chosen in the interval $[A, 2A]$ where A is equal to the sum of $|fr(o)|$ over all $o \in O$ divided by the total number of resources in an instance. A new value for the tabu tenure is chosen after every $2A$ steps. In this way the probability that the tabu search algorithm starts cycling in the neighborhood graph is reduced. Furthermore, the length of the list $\mathcal{L}$ in which the best local minima found are stored, is set to five. The number of steps L in which the overall best solution must be improved since otherwise restarting from a previously found local minimum takes place is set equal to $2000/A$ at first, and it is set equal to $1000/A$ once restarting has occurred. The algorithm is terminated when either all restarting possibilities are exhausted or when the total number of examined solutions exceeds $10^7/A$.

We have tested the algorithm on instances due to Nuijten [1994], who transformed well-known job shop instances into instances of two subproblems of the resource-constrained scheduling problem, viz., the multiple capacitated job shop scheduling problem (MCJSSP) and the job shop scheduling problem with resource sets and alternatives (RSAJSSP).

The MCJSSP is a generalization of the job shop in which machines can process several operations simultaneously. Each machine has an integer capacity and each operation has an integer size. A machine can process only those subsets of operations simultaneously whose summed sizes do not exceed its capacity. Instances of the MCJSSP are devised from job shop instances by duplicating all operations including processing times, precedences and machine assignments. All operations have size one, and the machines have capacity two. Upper bounds on the makespan of the original job shop instances are then upper bounds for the corresponding MCJSSP instances. Triplicated instances, in which machines have capacity three, are constructed similarly.

The RSAJSSP is a generalization of the job shop in which operations may require several machines for processing. Furthermore, an operation can be processed by several alternative machine sets. Machines can only process one operation at a time. Instances of the RSAJSSP are constructed from job shop instances as follows. Each operation requires one additional machine for which three alternatives exist. The processing times are either the original processing time or this

	$\|O\|$	$\|R\|$	cap	lb/ub	rcs	best	ϵ_{best}	avg	$t(s)$
f1d	100	5	2	666	666	669	0.45	682	21
f2d	100	5	2	655	683	669	2.14	688	35
f3d	100	5	2	593/597	638	645	8.77	661	25
f4d	100	5	2	572/590	590	601	5.07	630	65
f5d	100	5	2	593	593	593	0	596	12
f1t	150	5	3	666	671	671	0.75	688	35
f2t	150	5	3	655	704	694	5.95	716	27
f3t	150	5	3	590/597	647	636	7.80	660	36
f4t	150	5	3	570/590	592	615	7.89	645	63
f5t	150	5	3	593	593	593	0	599	12
g1d	150	5	2	926	926	926	0	932	49
g2d	150	5	2	890	890	893	0.33	925	75
g3d	150	5	2	863	863	866	0.35	895	73
g4d	150	5	2	951	951	951	0	967	53
g5d	150	5	2	958	958	958	0	961	90
g1t	225	5	3	926	926	926	0	931	59
g2t	225	5	3	890	913	900	1.12	913	101
g3t	225	5	3	863	866	873	1.16	888	135
g4t	225	5	3	951	952	952	0.11	956	132
g5t	225	5	3	958	960	958	0	962	69
a1d	200	10	2	888/935	935	962	8.33	993	113
a2d	200	10	2	750/765	765	783	4.40	804	74
a3d	200	10	2	783/844	844	856	0.93	897	101
a4d	200	10	2	730/840	840	849	16.30	872	106
a5d	200	10	2	829/902	902	907	9.41	960	93
fi06d	72	6	2	53/55	55	57	7.55	58	5
fi10d	200	10	2	835/930	963	989	18.44	1021	132
fi20d	200	5	2	1165	1319	1353	16.14	1409	291

Table 7.1: Results for multiple capacitated job shop instances.

	$\lvert O \rvert$	$\lvert R \rvert$	F	lb/ub	rcs	best	ϵ_{best}	avg	$t(s)$
f1ra	50	5	3	950	956	950	0	954	77
f2ra	50	5	3	881/883	893	883	0.23	887	75
f3ra	50	5	3	795/796	810	796	0.13	807	73
f4ra	50	5	3	836	840	836	0	840	74
f5ra	50	5	3	761	764	761	0	764	76
g1ra	75	5	3	1331	1331	1334	0.23	1342	130
g2ra	75	5	3	1249/1251	1251	1251	0.16	1278	141
g3ra	75	5	3	1275/1276	1276	1277	0.16	1295	134
g4ra	75	5	3	1421/1424	1424	1426	0.35	1432	131
g5ra	75	5	3	1340/1341	1341	1341	0.07	1350	133
a1ra	100	10	4	901/1082	1122	1082	20.09	1132	226
a2ra	100	10	4	780/901	946	901	15.51	955	212
a3ra	100	10	4	865/1003	1086	1003	15.95	1053	243
a4ra	100	10	4	891/1019	1030	1019	14.37	1065	250
a5ra	100	10	4	908/1078	1127	1078	18.72	1126	230
b1ra	150	10	4	1333/1478	1489	1478	10.88	1534	284
b2ra	150	10	4	1221/1365	1368	1365	11.80	1416	283
b3ra	150	10	4	1348/1498	1498	1506	11.72	1530	272
b4ra	150	10	4	1288/1401	1401	1437	11.57	1452	288
b5ra	150	10	4	1252/1380	1380	1396	11.50	1421	282
c1ra	200	10	4	1753/1884	1884	1925	9.81	1988	323
c2ra	200	10	4	1806/2001	2018	2001	10.79	2058	302
c3ra	200	10	4	1781/1968	1975	1968	10.50	2026	304
c4ra	200	10	4	1655/1858	1858	1876	13.35	1957	301
c5ra	200	10	4	1780/1942	1942	1963	10.28	2040	305
fi06ra	36	6	3	66	68	66	0	69	27
fi10ra	100	10	4	954/1114	1202	1114	16.77	1198	233
fi20ra	100	5	3	1703/1724	1724	1726	1.35	1798	175

Table 7.2: Results for job shop instances with resource sets and alternatives.

time multiplied by 1.2. Lower bounds for the original job shop instance are then lower bounds for the corresponding RSAJSSP instances.

Tables 7.1 and 7.2 list the results. In these tables, $|O|$ is the number of operations in an instance, $|R|$ is the number of resources, and the column "lb/ub" gives the best known lower and upper bounds. For the MCJSSP instances the upper bounds are derived from the original job shop instance, except for the instances a1d, a2d, a3d, and a4d for which these upper bounds could be improved. All lower bounds are computed by Nuijten [1994]. The column labeled "cap" gives the capacity of the machines in the MCJSSP instances, in which each operation has size one. For the RSAJSSP instances F is the number of machine set alternatives for an operation, where each machine set consists of two machines. The column "rcs" gives the results obtained by the randomized constraint satisfaction algorithm of Nuijten [1994]. The columns labeled "best" and ϵ_{best} give the best solution found in ten runs and its relative excess over the lower bound, the column "avg" gives the average cost of final solutions, and $t(s)$ is the average running time in seconds on a Sparc 5 workstation.

The tables show that the tabu search algorithm is slightly better than the randomized constraint satisfaction algorithm of Nuijten [1994] for the RSAJSSP instances, but it is outperformed by this algorithm for the MCJSSP instances. So whereas tabu search algorithms based on similar conceptual ideas as incorporated in our algorithm clearly outperform constraint satisfaction algorithms for job shop scheduling [Vaessens, 1995], this no longer seems to hold for the more general resource-constrained scheduling problem. This may be explained by the substantial increase of the computational complexity to evaluate the neighborhoods of schedules.

7.2.4 Parallel tabu search for the resource-constrained scheduling problem

In Section 7.1 we have attempted to design a parallel local search algorithm for the job shop scheduling problem based on multiple-step parallelism. This approach, however, turned out to be unsatisfactory for this problem. In this section we consider other options for applying parallelism in tabu search for the resource-constrained scheduling problem.

Multiple independent walks. In Chapter 4 we have argued that it is possible to obtain good speed-ups with multiple independent walks of a tabu search algorithm, provided that some rather mild conditions are met. First, the desired final solution quality should be much lower than the average relative excess of local minima. Secondly, the probability to find these (sub)optimal solutions should be given by a geometrical distribution. This condition is typically satisfied when the time needed to find a local minimum starting from an initial solution is equal to

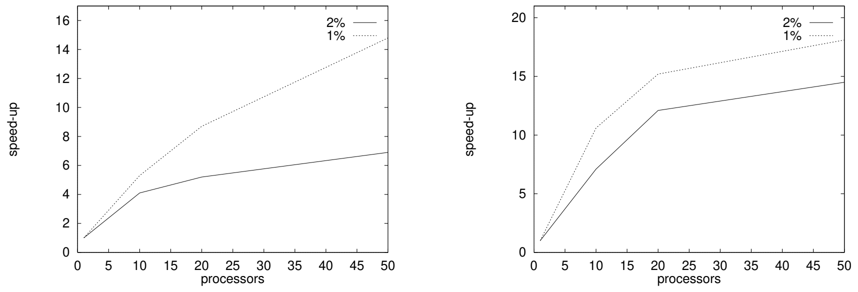

Figure 7.3: Speed-up with multiple independent walks for the instances f1t (left) and fish10ra (right) for finding solutions with a given relative excess.

the time needed to find other local minima starting from local minima.

Next, we study the amount of speed-up that can be obtained with multiple independent walks of our tabu search algorithm. For this, we perform runs of our tabu search algorithm in which the algorithm is halted when a solution with a given relative excess is found. The running time needed by P processors to find a solution with a given relative excess is then given by the minimum running time needed for this in P runs. In this way we can compute the average running time required by different numbers of employed processors and the resulting speed-ups for obtaining a given final solution quality. Figure 7.3 shows some typical results for the speed-up achieved with multiple independent walks of our tabu search algorithm for finding solutions with relative excesses of one and two percent over the best known upper bounds. We observe that this trivial approach to parallel tabu search for the RCSP results in good speed-ups and efficiencies —more than 50 percent— for as many as 20 processors, in particular when solutions with low relative excess are sought.

Single step parallelism. In Chapter 3 we have also argued that it is possible to obtain good speed-ups with single-step parallelism if best improvement is used for selecting neighbors. As this is the case in our tabu search algorithm, single-step parallelism may also be effective here. Single-step parallelism requires decomposition of a neighborhood into equally sized subsets containing neighbors that are explored simultaneously. Subsequently, one of the proposed neighbors replaces the current solution—that is, only a single proposed exchange is effectuated by the combination function; cf. Section 3.2.1. For this, we partition a neighborhood $\mathcal{N}'_1(G)$ of a solution G into equally sized subsets by imposing an order on the set $\lambda(o^\star)$ that determines the neighborhood of G. Each processor p, with $0 \leq p < P$, examines $|\mathcal{N}'_1(G)|/P$ neighbors whose rank in this order is between $p \cdot |\mathcal{N}'_1(G)|/P$ and $(p+1) \cdot |\mathcal{N}'_1(G)|/P$. The speed-up that can be obtained with

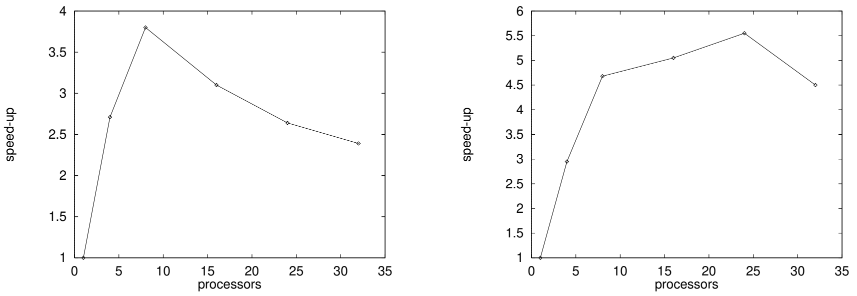

Figure 7.4: Speed-up with single-step parallelism for the instances f1t (left) and fish10ra (right).

single-step parallelism depends on the sizes of neighborhoods, the time needed to evaluate the cost of neighbors, and the time needed for all-to-all broadcasting. The sizes of neighborhoods rely also upon the status of the tabu list, in addition to the solution at hand. All-to-all broadcasting is necessary to communicate all proposed exchanges to all processors, after which processors can locally replace the current solution with the lowest-cost neighbor.

We have implemented a tabu search algorithm with single-step parallelism on a Parsytec PowerXplorer machine consisting of 32 PowerPC processing units configured in a torus. All-to-all broadcasting is done with the algorithm outlined in Section 6.2. Figure 7.4 shows the average speed-up using single-step parallelism for the instances f1t and fish10ra. The computational results show that good speed-ups and efficiencies can be obtained for moderate number of processors. The scalability, however, is rather limited due to the worse communication vs. computation ratio for larger number of processors.

An important observation is that it is possible to use more processors effectively, resulting in larger speed-ups, by combining single-step parallelism with multiple-walk parallelism since single-step parallelism can be used in each walk of a multiple-walk parallel local search algorithm. For example, performing 20 independent walks, each of which utilizes single-step parallelism using 8 processors, would result in a speed-up of 71 requiring 160 processors for the instance fish10ra. So large speed-ups can be obtained for the resource-constrained scheduling problem using a combination of concepts proposed in this thesis for parallel local search.

Bibliography

AARTS, E.H.L., F.M.J. DE BONT, J.H.A. HABERS, AND P.J.M. VAN LAARHOVEN [1986], Parallel implementations of the statistical cooling algorithm, *Integration* **4**, 209–238.

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

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

AARTS, E.H.L., AND M.G.A. VERHOEVEN [1996], Genetic local search for the traveling salesman problem, in: T. Bäck, D. Fogel, and Z. Michalewicz (eds.), *Handbook of Evolutionary Computing*, Oxford University Press.

ALLWRIGHT, J.R.A., AND D.B. CARPENTER [1989], A distributed implementation of simulated annealing for the traveling salesman problem, *Parallel Computing* **10**, 335–338.

APPLEGATE, D., R. BIXBY, V. CHVÁTAL, AND W. COOK [1995], *Finding cuts in the TSP*, Preliminary technical report.

AZENCOTT, R. (ed.) [1992], *Simulated Annealing: Parallelization Techniques*, Wiley, New York.

BACHEM, A., B. STECKEMETZ, AND M. WOTTAWA [1994], *An efficient parallel cluster heuristic for large traveling salesman problems*, Report 94-150, Zentrum für Paralleles Rechnen, Universität Köln, Germany.

BARBOSA, V., AND E. GAFNI [1989], A distributed implementation of simulated annealing, *J. of Parallel and Distributed Computing* **6**, 411–434.

BARR, R.S., AND B.L. HICKMAN [1993], Reporting computational experiments with parallel algorithms, *ORSA Journal on Computing* **5**, 2–18.

BATTITI, R., AND G. TECCHIOLLI [1992], Parallel biased search for combinatorial optimization: genetic algorithms and tabu, *Microprocessors and Microsystems* **16**, 351–367.

BAYER, R. [1972], Symmetric binary b-trees: data structures and maintenance algorithms, *Acta Informatica* **1**, 290–306.

BEASLEY, J.E. [1989], An SST-based algorithm for the Steiner problem in graphs, *Networks* **19**, 1–16.

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

BENTLEY, J.L. [1990], Experiments on traveling salesman heuristics, *Proc. 1st ACM-SIAM Symposium on Discrete Algorithms*, 91–99.

BERTSEKAS, D.P., AND J.N. TSITSIKLIS [1989], *Parallel and Distributed Computation*, Prentice-Hall, Englewood Cliffs.

BLAND, R.G., AND D.F. SHALLCROSS [1989], Large traveling salesman problems arising from experiments in x-ray crystallography, *Operations Research Letters* **8**, 123–133.

BLAZEWICZ, J., K.H. ECKER, G. SCHMIDT, AND J. WEGLARZ [1994], *Scheduling in Computer and Manufacturing Systems*, Springer-Verlag, Berlin.

BOESE, K.D., A.B. KAHNG, AND S. MUDDU [1994], A new adaptive multi-start technique for combinatorial global optimization, *Operations Research Letters* **16**, 101–113.

BOISSIN, N., AND J. LUTTON [1993], A parallel simulated annealing algorithm, *Parallel Computing* **19**, 859–872.

CASOTTO, A., F. ROMEO, AND A. SANGIOVANNI-VINCENTELLI [1987], A parallel simulated annealing algorithm for the placement of macro-cells, *IEEE Transactions on Computer-Aided Design* **6**, 838–847.

CHAKRAPANI, J., AND J. SKORIN-KAPOV [1993a], Connection machine implementation of a tabu search algorithm for the traveling salesman problem, *Journal of Computing and Information Technology* **1**, 29–36.

CHAKRAPANI, J., AND J. SKORIN-KAPOV [1993b], Massively parallel tabu search for the quadratic assignment problem, *Annals of Operations Research* **41**, 327–342.

CHANDRA, B., H. KARLOFF, AND C. TOVEY [1994], New results on the old k-opt algorithm for the TSP, *Proc. 5th ACM-SIAM Symposium on Discrete Algorithms*, 150–159.

CHEN, N.P. [1983], New algorithms for Steiner tree on graphs, *Proc. of the IEEE Int. Symposium on Circuits and Systems*, 1217–1219.

CRAINIC, T.G., M. TOULOUSE, AND M. GENDREAU [1993a], *Parallel asynchronous tabu search for multicommodity location-allocation with balancing requirements*, Publication 935, Centre de recherche sur les transports, Université de Montréal.

CRAINIC, T.G., M. TOULOUSE, AND M. GENDREAU [1993b], *Towards a taxonomy of parallel tabu search algorithms*, Publication 933, Centre de recherche sur les transports, Université de Montréal.

CRAINIC, T.G., M. TOULOUSE, AND M. GENDREAU [1995], Synchronous tabu search parallelization strategies for multicommodity location-allocation with balancing requirements, *OR Spektrum* **17**, 113–123.

CROES, G.A. [1958], A method for solving traveling salesman problems, *Operations Research* **6**, 791–812.

DAREMA, F., S. KIRKPATRICK, AND V.A. NORTON [1987], Parallel algorithms for chip placement by simulated annealing, *IBM Journal of Research and Development* **31**, 391–402.

DAUZÈRE-PÉRÈS, S., AND J. PAULLI [1995], *A global tabu search procedure for the general multiprocessor job shop scheduling problem*, Report 95/5, Institute of Mathematics, University of Aarhus.

DIEKMANN, R., R. LÜLING, AND J. SIMON [1993], Problem independent distributed simulated annealing and its applications, in: R.V.V. Vidal (ed.), *Applied Simulated Annealing*, LNEMS 396, Springer-Verlag, 18–44.

DODD, N. [1990], Slow annealing versus multiple fast annealing runs – an empirical investigation, *Parallel Computing* **16**, 269–272.

DOWSLAND, K.A. [1991], Hill-climbing, simulated annealing and the Steiner problem in graphs, *Engineering Optimization* **17**, 91–107.

DUIN, C.W. [1994], *Steiner's Problem in Graphs*, Ph.D. thesis, Amsterdam University.

DUIN, C., AND S. VOSS [1993], Steiner tree heuristics – a survey, in: H. Dyckhoff, U. Derigs, M. Salomon, and H.C. Tijms (eds.), *Proc. 22nd annual meeting DGOR / NSOR*, Springer-Verlag, Berlin, 485–496.

EIKELDER, H.M.M. TEN, M.G.A. VERHOEVEN, T.W.M. VOSSEN, AND E.H.L. AARTS [1996], A probabilistic analysis of local search, in: I.H. Osman and J.P. Kelly (eds.), *Meta-heuristics: Theory and Applications*, Kluwer Academic, Boston.

FELTEN, E., S. KARLIN, AND S.W. OTTO [1985], The traveling salesman problem on a hypercubic mimd machine, *Proc. Int. Conference on Parallel Processing*, 6–10.

FERREIRA, A.G., AND J. ZEROVNIK [1993], Bounding the probability of success of stochastic methods for global optimization, *Computers & Mathematics with Applications* **25**, 1–8.

FIECHTER, C.N. [1994], A parallel tabu search algorithm for large traveling salesman problems, *Discrete Applied Mathematics* **51**, 243–267.

FORTUNE, S. [1987], A sweepline algorithm for voronoi diagrams, *Algorithmica* **2**, 153–174.

FOX, B.L. [1993], Integrating and accelerating tabu search, simulated annealing, and genetic algorithms, *Annals of Operations Research* **41**, 47–67.

FREDMAN, M.L., D.S. JOHNSON, L.A. MCGEOCH, AND G. OSTHEIMER [1993], Data structures for the traveling salesman, *Proc. 4th ACM-SIAM Symposium on Discrete Algorithms*, 145–154.

FRENCH, S. [1982], *Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop*, Wiley, Chichester.

GARCIA, B., J. POTVIN, AND J. ROUSSEAU [1994], A parallel implementation of the tabu search heuristic for vehicle routing problems with time window constraints, *Computers and Operations Research* **21**, 1025–1033.

GAREY, M.R., AND D.S. JOHNSON [1979], *Computers and Intractability: A Guide to the Theory of NP-completeness*, Freeman, San Francisco.

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

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

GREENING, D.R. [1990], Parallel simulated annealing techniques, *Physica D* **42**, 293–306.

GREENLAW, R., H.J. HOOVER, AND W.L. RUZZO [1995], *Limits to Parallel Computation: P-completeness Theory*, Oxford University Press, Oxford.

HURINK, J., B. JURISCH, AND M. THOLE [1994], Tabu search for the job shop scheduling problem with multi-purpose machines, *OR Spektrum* **16**, 205–215.

HWANG, F.K., D.S. RICHARDS, AND P. WINTER [1992], *The Steiner Tree Problem*, Annals of Discrete Mathematics 53.

JOG, P., J.Y. SUH, AND D. VAN GUCHT [1991], Parallel genetic algorithms applied to the traveling salesman problem, *SIAM Journal on Optimization* **1**, 515–529.

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

JOHNSON, D.S., AND L.A. MCGEOCH [1996], The traveling salesman problem: a case study in local optimization, in: [Aarts & Lenstra, 1996].

JOHNSON, D.S., C.H. PAPADIMITRIOU, AND M. YANNAKAKIS [1988], How easy is local search?, *Journal of Computer and System Sciences* **37**, 79–100.

JONES, M.H., AND P. BANERJEE [1987], An improved simulated annealing algorithm for standard cell placement, *Proc. of the International Conference on Computer Design*, 83–86.

KAPSALIS, A., V.J. RAYWARD-SMITH, AND G.D. SMITH [1993], Solving the graphical Steiner tree problem using genetic algorithms, *Journal of the Operational Research Society* **44**, 397–406.

KERN, W. [1989], A probabilistic analysis of the switching algorithm for the Euclidian TSP, *Mathematical Programming* **44**, 213–219.

KERNIGHAN, B.W., AND S. LIN [1970], An efficient heuristic procedure for partitioning graphs, *Bell System Technical Journal* **49**, 291–307.

KINDERVATER, G.A.P., J.K. LENSTRA, AND M.W.P. SAVELSBERGH [1993], Sequential and parallel local search for the time constrained traveling salesman problem, *Discrete Applied Mathematics* **42**, 211–25.

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

KIRKPATRICK, S., AND G. TOULOUSE [1985], Configuration space analysis of travelling salesman problems, *Journal Physique* **46**, 1277–1292.

KRAVITZ, S.A., AND R.A. RUTENBAR [1987], Placement by simulated annealing on a multiprocessor, *IEEE Transactions on Computer Aided Design* **6**, 534–549.

KRUSKAL, C.P., L. RUDOLPH, AND M. SNIR [1990], A complexity theory of efficient parallel algorithms, *Theoretical Computer Science* **71**, 95–132.

KRUSKAL, J.B. [1956], On the shortest spanning tree of a graph and the traveling salesman problem, *Proc. Amer. Math. Soc.* **7**, 48–50.

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

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

LAWLER, E.L., J.K. LENSTRA, A.H.G. RINNOOY KAN, AND D.B. SHMOYS [1985], *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, Wiley, Chichester.

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

LEON, V.J., AND R. BALAKRISHNAN [1995], Strength and adaptability of problem-space based neighborhoods for resource-constrained scheduling, *OR Spektrum* **17**, 173–182.

LI, Y., AND P.M. PARDALOS [1992], Parallel algorithms for the quadratic assignment problem, in: P.M Pardalos (ed.), *Advances in Optimization and Parallel Computing*, Kluwer, 177–189.

LIN, S. [1965], Computer solutions of the traveling salesman problem, *Bell System Technical Journal* **44**, 2245–2269.

LIN, S., AND B.W. KERNIGHAN [1973], An effective heuristic algorithm for the traveling salesman problem, *Operations Research* **21**, 498–516.

MAHFOUD, S.W., AND D.E. GOLDBERG [1995], Parallel recombinative simulated annealing: a genetic algorithm, *Parallel Computing* **21**, 1–28.

MALEK, M., M. GURUSWAMY, AND M. PANDYA [1989], Serial and parallel simulated annealing and tabu search for the traveling salesman problem, *Annals of Operations Research* **21**, 59–84.

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

MICHALEWICZ, Z. [1992], *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag, Berlin.

MOSCATO, P. [1993], An introduction to population approaches for optimization and hierarchical objective functions: a discussion on the role of tabu search, *Annals of Operations Research* **41**, 85–122.

MÜHLENBEIN, H. [1992], Parallel genetic algorithms in combinatorial optimization, in: O. Balci (ed.), *Computer Science and Operations Research*, Pergamon Press.

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

NEMHAUSER, G.L., AND L.A. WOLSEY [1988], *Integer and Combinatorial Optimization*, Wiley, New York.

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

NUIJTEN, W.P.M. [1994], *Time and Resource Constrained Scheduling: A Constraint Satisfaction Approach*, Ph.D. thesis, Eindhoven University.

OSBORNE, L.J., AND B.E. GILLETT [1991], A comparison of two simulated annealing algorithms applied to the directed Steiner problem on networks, *ORSA Journal on Computing* **3**, 213–225.

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

PAPOULIS, A. [1965], *Probability, Random Variables, and Stochastic Processes*, McGraw-Hill, London.

PINEDO, M. [1995], *Scheduling: Theory, Algorithms, and Systems*, Prentice Hall, Englewood Cliffs.

RAVIKUMAR, C.P. [1992], Parallel techniques for solving large scale travelling salesperson problems, *Microprocessors and Microsystems* **16**, 149–158.

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

REINELT, G. [1991], TSPLIB: A traveling salesman problem library, *ORSA Journal on Computing* **3**, 376–384.

REINELT, G. [1992], Fast heuristics for large geometric traveling salesman problems, *ORSA Journal on Computing* **4**, 206–217.

REINELT, G. [1994], *The Traveling Salesman: Computational Solutions for TSP Applications*, LNCS 840, Springer-Verlag, Berlin.

ROMEO, F., AND A. SANGIOVANNI-VINCENTELLI [1991], A theoretical framework for simulated annealing, *Algorithmica* **6**, 302–345.

ROSE, J.S., W.M. SNELGROVE, AND Z.G. VRANESIC [1988], Parallel standard cell placement algorithms with quality equivalent to simulated annealing, *IEEE Transactions on Computer-Aided Design* **7**, 387–396.

ROUSSEL-RAGOT, P., AND G. DREYFUS [1990], A problem independent parallel implementation of simulated annealing: models and experiments, *IEEE Transactions on Computer-Aided Design* **9**, 827–835.

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

SAMPSON, S.E., AND E.N. WEISS [1993], Local search techniques for the generalized resource-constrained project scheduling problem, *Naval Research Logistics* **40**, 665–675.

SAVAGE, J.E., AND M.G. WLOKA [1991], Parallelism in graph-partitioning, *Journal of Parallel and Distributed Computing* **13**, 257–272.

SECHEN, C. [1988], *VLSI Placement and Global Routing using Simulated Annealing*, Kluwer Academic, Dordrecht.

SHAHOOKAR, K., AND P. MAZUNDER [1991], VLSI cell-placement techniques, *ACM Computing Surveys* **23**, 143–220.

SHONKWILER, R., AND E. VAN VLECK [1994], Parallel speed-up of Monte Carlo methods for global optimization, *Journal of Complexity* **10**, 64–95.

SLOWINSKI, R., AND J. WEGLARZ (eds.) [1989], *Advances in Project Scheduling*, Elsevier, Amsterdam.

STADLER, P.F., AND W. SCHNABL [1992], The landscape of the traveling salesman problem, *Physics Letters A* **161**, 337–344.

STUART, A., AND J.K. ORD [1987], *Kendall's Advanced Theory of Statistics*, Vol 1: Distribution Theory, Griffin & Company, London.

TAILLARD, E. [1990], Some efficient heuristic methods for the flow shop sequencing problem, *European Journal of Operational Research* **47**, 65–74.

TAILLARD, E. [1991], Robust taboo search for the quadratic assignment problem, *Parallel Computing* **17**, 443–455.

TAILLARD, E. [1993], Parallel iterative search methods for vehicle routing problems, *Networks* **23**, 661–673.

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

TAKAHASHI, H., AND A. MATSUYAMA [1980], An approximate solution for the Steiner tree problem in graphs, *Math. Japonica* **24**, 573–577.

TOVEY, C.A. [1985], Hill climbing with multiple local optima, *SIAM Journal of Discrete Mathematics* **6**, 383–393.

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

VERHOEVEN, M.G.A., E.H.L. AARTS, E. VAN DE SLUIS, AND R.J.M. VAESSENS [1992], Parallel local search and the travelling salesman problem (extended abstract), *Parallel Problem Solving from Nature 2*, North-Holland, Amsterdam, 543–552.

VERHOEVEN, M.G.A., AND E.H.L. AARTS [1994], A parallel Lin-Kernighan algorithm for the traveling salesman problem, in: G. Joubert, D. Trystram, F. Peters, and D. Evans (eds.), *Parallel Computing: Trends and Applications*, Elsevier Science, Amsterdam, 559–563.

VERHOEVEN, M.G.A., AND E.H.L. AARTS [1995a], Parallel local search, *Journal of Heuristics* **1**, 43–65.

VERHOEVEN, M.G.A., AND E.H.L. AARTS [1995b], Parallel local search and job shop scheduling, in: A. Ferreira and J. Rolim (eds.), *Parallel Algorithms for Irregular Problems: State of the Art*, Kluwer, Boston, 195–212.

VERHOEVEN, M.G.A., E.H.L. AARTS, AND M.E.M. SEVERENS [1995], Local search for the Steiner tree problem, *Proc. of Adaptive Decision Technologies (ADT-95)*, 345–352.

VERHOEVEN, M.G.A., E.H.L. AARTS, AND P.C.J. SWINKELS [1995], A parallel 2-opt algorithm for the traveling salesman problem, *Future Generation Computer Systems* **11**, 175–182.

VOSS, S. [1992], Steiner's problem in graphs: heuristic methods, *Discrete Applied Mathematics* **40**, 45–72.

VOSS, S. [1993], Tabu search: applications and prospects, in: D.Z. Du and P.M. Pardalos (eds.), *Network Optimization Problems*, World Scientific, 333–353.

WEINBERGER, E.D. [1991], Local properties of Kauffman's N-k model: A tunably rugged energy landscape, *Physical Review A* **44**, 6399–6413.

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

YANNAKAKIS, M. [1996], Computational complexity of local search, in: [Aarts & Lenstra, 1996].

YOO, J., T. YANG, AND J.P. IGNIZIO [1995], An exchange heuristic for resource constrained scheduling with consideration given to opportunities for parallel processing, *Production Planning & Control* **6**, 140–150.

Index

Samenvatting

Dit proefschrift behandelt parallelle lokale-zoekalgoritmen voor combinatorische optimaliseringsproblemen. Dit zijn problemen waarbij een optimale oplossing gevonden moet worden uit een extreem groot, maar eindig, aantal alternatieve oplossingen met bepaalde kosten. Een grote verscheidenheid aan problemen in planning- en ontwerpsituaties kunnen gemodelleerd worden als een combinatorisch optimaliseringsprobleem. Voorbeelden hiervan zijn het opstellen van produktieroosters in fabrieken en lesroosters in scholen, het berekenen van voertuigroutes en het berekenen van layouts voor geïntegreerde schakelingen. Instanties van deze problemen kunnen veelal niet optimaal opgelost worden in aanvaardbare tijd en daarom moet er volstaan worden met suboptimale oplossingen.

Lokaal zoeken is een benaderingstechniek voor lastige combinatorische optimaliseringsproblemen die in staat is goede kwaliteit oplossingen te vinden voor een grote klasse van problemen. Een lokaal-zoekalgoritme is gebaseerd op het herhaaldelijk doorzoeken van *buurruimten* van oplossingen. Een buurruimte van een oplossing bestaat uit een verzameling buuroplossingen. Deze oplossingen worden geconstrueerd door het toepassen van een verwisselingsfunctie die een aantal elementen uit een oplossing vervangt door andere elementen. Een buurruimte induceert een graaf op de oplosruimte, de buurruimtegraaf, waarin twee buuroplossingen verbonden zijn door een kant. Een lokaal-zoekalgoritme voert in deze graaf een wandeling uit die bestaat uit opeenvolgende *stappen* van oplossing naar buuroplossing. Het klassieke lokaal-zoekalgoritme is *iteratieve verbetering* waarin alleen stappen worden gemaakt naar buren met lagere kosten totdat een *lokaal minimum*, een oplossing zonder buren met lagere kosten, bereikt wordt. Een nadeel van iteratieve verbetering is dat het algoritme kan stoppen in een lokaal minimum van slechte kwaliteit. Om dit risico te verlagen kan een grotere buurruimte gekozen worden of een andere manier om de wandeling in de buurruimtegraaf uit te voeren. Voorbeelden van de eerste aanpak zijn variabele-diepte algoritmen waarin buren verkregen worden door reeksen verwisselingen. Varianten van lokaal zoeken gebaseerd op de tweede aanpak zijn herhaald lokaal zoeken, tabu search, simulated annealing en genetisch lokaal zoeken. Kenmerkend voor al deze varianten is dat ook stappen naar buren met hogere kosten toegelaten zijn.

Lokale-zoekalgoritmen vergen vaak lange rekentijden voor de grotere probleem instanties. In dit proefschrift onderzoeken we de mogelijkheden van parallelle computers om deze rekentijden te verkleinen, waardoor het tevens mogelijk wordt om grotere instanties te hanteren of om betere oplossingen te vinden in een gegeven tijdsbestek. Het doel van het onderzoek is het ontwerpen van parallelle lokale zoekalgoritmen die kunnen concurreren met de beste sequentiële algoritmen, zowel met betrekking tot rekentijd als kwaliteit van de gevonden oplossingen. Hiertoe onderzoeken we zowel algemene technieken die toepasbaar zijn op een brede klasse van problemen als technieken die toegesneden zijn op een specifiek probleem. Schaalbaarheid van algoritmen, het gedrag bij toenemend aantal processoren, is hierbij een belangrijk aspect. Rekenexperimenten op parallelle computers vormen daarom een belangrijk onderdeel van dit onderzoek.

Eerst presenteren we een overzicht van concepten voor parallelle lokale-zoekalgoritmen. Hierbij maken we onderscheid tussen *één-wandelings* en *meer-wandelings* parallellisme. In één-wandelings parallellisme worden meerdere processoren ingezet voor het uitvoeren van één wandeling en in meer-wandelings parallellisme worden verscheidene wandelingen tegelijkertijd door verschillende processoren uitgevoerd. Binnen één-wandelings parallellisme maken we verder onderscheid tussen *één-staps* en *meer-staps* parallellisme. Het idee van één-staps parallellisme is om buuroplossingen van een oplossing gelijktijdig te evalueren en vervolgens één stap naar een buuroplossing te maken. In algoritmen met meerstaps parallellisme worden meerdere opeenvolgende stappen van een wandeling tegelijkertijd uitgevoerd. In de klasse van meer-wandelings algoritmen maken we onderscheid tussen algoritmen die onafhankelijke wandelingen uitvoeren en algoritmen die interactie tussen wandelingen toelaten.

De eenvoudigste aanpak voor parallel lokaal zoeken is het gelijktijdig uitvoeren van een aantal onafhankelijke wandelingen. We bestuderen de *versnelling*, de factor waarmee de rekentijd afneemt, die behaald kan worden door deze aanpak. Dit gebeurt aan de hand van een studie voor het handelsreizigersprobleem, een probleem waarin de kortste route gevonden moet worden voor een rondreis waarbij een aantal steden bezocht dient te worden. Voor dit probleem wordt eerst het gemiddelde gedrag van een twee-verwisselingsbuurruimte bestudeerd. In deze buurruimte worden buren geconstrueerd door twee trajecten in een tour te vervangen door twee nieuwe trajecten. Met behulp van de analyse voor dit probleem wordt de versnelling bepaald die verkregen kan worden door een aantal onafhankelijke wandelingen parallel uit te voeren. Het belangrijkste resultaat is dat een goede versnelling verkregen kan worden voor het vinden van suboptimale oplossingen van hoge kwaliteit indien de tijden benodigd voor het vinden van een eerste lokale minimum en daaropvolgende lokale minima vergelijkbaar zijn.

Verder presenteren we een aantal studies van toegesneden aanpakken, gebaseerd op meer-staps parallellisme, voor een aantal klassieke combinatorische optimaliseringsproblemen, waaronder het handelsreizigersprobleem. We ontwerpen en analyseren parallelle twee-verwisselings- en drie-verwisselingsalgoritmen voor dit probleem. Deze algoritmen vinden gelijkwaardige kwaliteit eindoplossingen als sequentiële algoritmen en vertonen een goede versnelling. Verder presenteren we een parallel variabel-diepte algoritme dat gebaseerd is op de Lin–Kernighan buurruimte. Dit algoritme maakt gebruik van geavanceerde data-structuren en buurruimtereductietechnieken. Het algoritme is geïmplementeerd op een parallelle computer bestaande uit 32 processoren en vertoont een aanvaardbare versnelling. Het algoritme kan concurreren met geavanceerde sequentiële implementaties uitgevoerd op krachtige werkstations.

In het Steinerbomenprobleem moet een boom met minimaal gewicht geconstrueerd worden die ten minste een gegeven deelverzameling van knopen van een graaf omvat. We introduceren nieuwe buurruimten voor dit probleem. Rekenresultaten voor grote instanties tonen aan dat goede kwaliteit oplossingen in bescheiden rekentijden gevonden kunnen worden. Verder wordt aangetoond dat het mogelijk is om lokale-zoekalgoritmen met meer-staps parallellisme te ontwerpen voor dit probleem die, zonder verlies van kwaliteit, een goede versnelling vertonen op een parallelle computer.

In het werkplaatsroosterprobleem dient een rooster gevonden te worden voor een aantal reeksen van taken die elk een bepaalde machine vereisen. Het rooster dient zodanig te zijn dat alle uit te voeren handelingen binnen een zo kort mogelijke tijd voltooid zijn. We onderzoeken de toepasbaarheid van meer-staps parallellisme voor dit probleem. Verder presenteren we een lokaal-zoekalgoritme voor een veel algemener roosterprobleem. In dit probleem is het mogelijk dat een handeling meerdere hulpmiddelen vergt waarvoor alternatieve combinaties bestaan, en bovendien kunnen hulpmiddelen in meerdere handelingen tegelijkertijd gebruikt worden. Voor dit probleem ontwerpen we nieuwe buurruimten en buurruimtereductietechnieken. Tevens wordt aangetoond dat het mogelijk is het algoritme aanzienlijk te versnellen door middel van een combinatie van meerwandelings en één-staps parallellisme.

Samenvattend kan gesteld worden dat het mogelijk is om parallellisme effectief aan te wenden in lokale-zoekalgoritmen voor een brede klasse van combinatorische optimaliseringsproblemen.