LOCAL search

in combinatorial optimization

EMILE AARTS & JAN KAREL LENSTRA, EDITORS

Local Search in Combinatorial Optimization
Edited by E. Aarts and J. K. Lenstra
© 1997 John Wiley & Sons Ltd.

1

Introduction

Emile H. L. Aarts
Philips Research Laboratories, Eindhoven; Eindhoven University of Technology

Jan Karel Lenstra
Eindhoven University of Technology; CWI, Amsterdam

1 HISTORICAL PERSPECTIVE 1
2 FORMULATION AND MODELING 3
3 NEIGHBORHOODS 4
3.1 Simple exchange neighborhoods 4
3.2 More intricate exchange neighborhoods 7
4 ANALYSIS AND COMPLEXITY 10
4.1 Empirical results 10
4.2 Theoretical results 11
5 ADVANCED SEARCH STRATEGIES 13
5.1 Simulated annealing 14
5.2 Tabu search 14
5.3 Genetic algorithms 15
5.4 Neural networks 15
6 CONCLUSION 16

1 HISTORICAL PERSPECTIVE

In this book we are concerned with problems in combinatorial optimization. Such problems arise in situations where discrete choices must be made, and solving them amounts to finding an optimal solution among a finite or countably infinite number of alternatives. Optimality relates to some cost criterion, which provides a quantitative measure of the quality of each solution. This area of discrete mathematics is of great practical use and has attracted much attention over the years. For detailed introductions into the field the reader is referred to the books by Lawler [1976], Papadimitriou & Steiglitz [1982], Schrijver [1986], and Nemhauser & Wolsey [1988]. Useful collections of annotated bibilographies are given by O'hEigeartaigh, Lenstra & Rinnooy Kan [1985] and Dell'Amico, Maffioli & Martello [1997].

Many combinatorial optimization problems are *NP-hard*; see Garey & Johnson [1979]. It is generally believed that NP-hard problems cannot be solved to optimality within polynomially bounded computation times. Consequently,

there is much interest in approximation algorithms that can find near-optimal solutions within reasonable running times. The literature distinguishes between two broad classes of approximation algorithms: *constructive* and *local search* methods. This book concentrates on the latter. They are typically instantiations of various general search schemes, but all have the same feature of an underlying neighborhood function, which is used to guide the search for a good solution [Papadimitriou & Steiglitz, 1982; Yannakakis, 1990].

The use of local search in combinatorial optimization has a long history that reaches back to the late 1950s and early 1960s, when the first edge-exchange algorithms for the traveling salesman problem (TSP) were introduced; see the work of Bock [1958a,b], Croes [1958], Lin [1965], and Reiter & Sherman [1965]. In the subsequent years the scope of local search has been broadened, and the basic concept of an exchange algorithm has been applied with some success to a variety of problems. We mention scheduling [Page, 1965; Nicholson, 1965] and graph partitioning [Kernighan & Lin, 1970]. Nicholson [1971] extended the concept of exchange strategies to a more general class of permutation problems, and discussed their application to network layout design, scheduling, vehicle routing, and cutting stock problems. Despite the early successes, local search was not considered a mature technique for a long time. This was mainly because the success could be attributed to its practical usefulness only, and no major conceptual progress was made at that time. The past decade shows a strong renewed interest in the subject, which is due to the following three developments:

- *Appeal.* Many variations of local search algorithms have been proposed based on analogies with processes in nature, relating the subject to other disciplines such as statistical physics, biological evolution, and neurophysiology. This has generated new algorithms or paradigms. Well-known examples are *simulated annealing*, *genetic algorithms* and some variants of *neural networks*. The analogies are sometimes taken quite far, leading to variants with extravagant names such as the *great deluge* algorithm of Dueck [1993], the *hide and seek* game of Bélisle, Romeijn & Smith [1990], or the *roaming ants* approach of Colorni, Dorigo & Manniezzo [1992].
- *Theory.* Some of the newly proposed local search algorithms have been mathematically modeled, yielding theoretical results on their performance. Probably the best-known example is the modeling of the simulated annealing algorithm by Markov chains [Aarts & Korst, 1989a]. Furthermore, Johnson, Papadimitriou & Yannakakis [1988] introduced a complexity theory of local search, which provided more theoretical insight, not only into the complexity of local search but also into the combinatorial structure of discrete optimization problems.
- *Practice.* The large increase in computational resources together with the use of sophisticated data structures have made local search algorithms strong competitors within the class of algorithms designed to handle large problem instances [Bentley, 1992; Reinelt, 1992; Fredman et al., 1995]. Furthermore,

the flexibility and ease of implementation of local search algorithms have led to the successful handling of many complex real-world problems.

2 FORMULATION AND MODELING

The use of a local search algorithm presupposes definitions of a problem and a neighborhood. These can be formulated as follows.

Definition 1 *A* combinatorial optimization problem *is specified by a set of problem instances and is either a* minimization *problem or a* maximization *problem.*

Unless stated otherwise, we consider in this chapter only minimization problems.

Definition 2 *An* instance *of a combinatorial optimization problem is a pair* $(\mathscr{S}, f)$, *where the* solution set $\mathscr{S}$ *is the set of feasible solutions and the* cost function f *is a mapping* $f: \mathscr{S} \to \mathbb{R}$. *The problem is to find a* globally optimal *solution, i.e., an* $i^* \in \mathscr{S}$ *such that* $f(i^*) \leqslant f(i)$ *for all* $i \in \mathscr{S}$. *Furthermore,* $f^* = f(i^*)$ *denotes the optimal cost, and* $\mathscr{S}^* = \{i \in \mathscr{S} \mid f(i) = f^*\}$ *denotes the set of optimal solutions.*

The instance $(\mathscr{S}, f)$ is generally not given explicitly, i.e., by a listing of all solutions and their costs. Usually, one has a compact data representation of an instance, and polynomial-time algorithms to verify whether a solution belongs to $\mathscr{S}$ and to compute the cost of any solution in $\mathscr{S}$ [Garey & Johnson, 1979]. The size of this representation, i.e., the number of bits needed to store it in a computer, is taken as the size of the problem instance.

The solution set is often represented by a set of *decision variables*, whose values can have certain ranges. A solution is then represented by a value assignment of the variables. These variables can be directly related to the model that is used to formulate the problem. An example is given by the integer variables in *integer linear programming* models [Schrijver, 1986; Nemhauser & Wolsey, 1988]. Furthermore, such solution representations can be of great use to model neighborhoods, as we will show below.

Definition 3 *Let* $(\mathscr{S}, f)$ *be an instance of a combinatorial optimization problem. A* neighborhood function *is a mapping* $\mathscr{N}: \mathscr{S} \to 2^{\mathscr{S}}$, *which defines for each solution* $i \in \mathscr{S}$ *a set* $\mathscr{N}(i) \subseteq \mathscr{S}$ *of solutions that are in some sense close to i. The set* $\mathscr{N}(i)$ *is the* neighborhood *of solution i, and each* $j \in \mathscr{N}(i)$ *is a* neighbor *of i. We shall assume that* $i \in \mathscr{N}(i)$, *for all* $i \in \mathscr{S}$.

Roughly speaking, a local search algorithm starts off with an initial solution and then continually tries to find better solutions by searching neighborhoods.

A basic version of local search is *iterative improvement*. Iterative improvement starts with some initial solution and searches its neighborhood for a solution of lower cost. If such a solution is found, it replaces the current solution, and the search continues. Otherwise, the algorithm returns the current solution, which is then *locally optimal* as defined below.

Definition 4 *Let* $(\mathscr{S}, f)$ *be an instance of a combinatorial optimization problem and let* $\mathscr{N}$ *be a neighborhood function. A solution* $\hat{i} \in \mathscr{S}$ *is* locally optimal (minimal)

with respect to $\mathcal{N}$ *if*

$$f(\hat{\imath}) \leqslant f(j) \quad \textit{for all } j \in \mathcal{N}(\hat{\imath}).$$

We denote the set of locally optimal solutions by $\hat{\mathcal{S}}$.

Note that local optimality depends on the neighborhood function that is used.

Definition 5 *Let* $(\mathcal{S}, f)$ *be an instance of a combinatorial optimization problem, and let* $\mathcal{N}$ *be a neighborhood function.* $\mathcal{N}$ *is* exact *if* $\hat{\mathcal{S}} \subseteq \mathcal{S}^*$.

Iterative improvement can apply either *first improvement*, in which the current solution is replaced by the first cost-improving solution found by the neighborhood search, or *best improvement*, in which the current solution is replaced by the best solution in its neighborhood. An iterative improvement algorithm terminates by definition in a local optimum, and if the applied neighborhood function is exact, it is an optimization algorithm.

A local search process can be viewed as a walk in a directed, vertex-labeled graph, whose vertices are given by the elements in $\mathcal{S}$ and whose arcs are given by all (solution, neighbor) pairs. The vertex labels are given by the corresponding cost values. An illustrative example of local search is the simplex algorithm for linear programming [Dantzig, 1951, 1963], where the simplex polytope determines the neighborhood graph and the successive applications of a pivoting rule determine the walk. Note that the corresponding neighborhood function is exact [Dantzig, 1963; Chvátal, 1983; Schrijver, 1986].

3 NEIGHBORHOODS

Neighborhoods depend on the problem under consideration, and finding efficient neighborhood functions that lead to high-quality local optima can be viewed as one of the challenges of local search. So far, no general rules are available and each situation has to be considered separately. The literature presents many examples of neighborhood functions, and even for the same problem there are often many different possibilities available. To structure the discussion of this section, we first present a class of neighborhood functions that are based on a simple exchange principle. Elaborating on the exchange principle we next introduce a class of more intricate neighborhood functions.

3.1 Simple exchange neighborhoods

In many combinatorial optimization problems, solutions can be represented as sequences or partitions. These solution representations enable the use of *k-exchange neighborhoods*, i.e., neighborhoods that are obtained by exchanging k elements in a given sequence or partition. We discuss some examples below. The definitions of the graph-theoretical concepts used in this section can be found in Harary [1972] and Bondy & Murty [1976].

Definition 6 (Sorting problem) *Given is a multiset* $A = \{a_1, a_2, \ldots, a_n\}$ *of* n *positive numbers. Find a nonincreasing order of the numbers in* A.

from a set of cardinality n. In the case of the traveling salesman problem there is an additional exponential factor originating from the number of different ways of reconnecting the k paths that are left over by removing k edges from the Hamiltonian cycle. The exchange neighborhood for sorting is exact, even in the case where the exchanges are restricted to adjacent numbers. The neighborhoods for the uniform graph partitioning problem and the traveling salesman problem are in general only exact for $k \geqslant n-1$. Note that the neighborhood graphs defined in the three examples are all strongly connected, implying that each solution can be reached from any other one. This is an important property in the analysis of the convergence of some algorithms, e.g., simulated annealing. The exchange neighborhoods for the uniform graph partitioning and traveling salesman problems given above are attributed to Kernighan & Lin [1970] and Lin [1965], respectively.

As in the simple iterative improvement algorithm, any local search algorithm has three basic steps: generation of a start solution, generation of a neighboring solution, and calculation of cost values.

For the examples discussed above, these steps involve simple computations, so efficient algorithms can be readily obtained. But this is not generally true. Efficient algorithms may sometimes require intricate computations, or they may not even exist. For instance, finding a feasible start solution for the traveling salesman problem with time windows is NP-hard. Also cost calculations may be quite difficult. For instance in VLSI design, placement is a well-known layout problem in which a set of modules is to be placed on a two-dimensional Euclidean map such that a weighted sum of the area of the surrounding box and the total length of the wires connecting the modules is minimal. Computing the wire length requires a routing model, and for some models, including the Steiner tree routing [Sahni & Bhatt, 1980], wire length computations are intractable.

To illustrate some of the complications that may arise in neighborhood computations, we discuss the job shop scheduling problem [Conway, Maxwell & Miller, 1967].

Definition 9 (Job shop scheduling problem) *Given are a set $\mathcal{J}$ of n jobs, a set $\mathcal{M}$ of m machines, and a set $\mathcal{O}$ of N operations. For each operation $v \in \mathcal{O}$ we are given a job $J_v \in \mathcal{J}$ to which it belongs, a machine $M_v \in \mathcal{M}$ on which it must be processed and a processing time $p_v \in \mathbb{Z}^+$. Furthermore, a binary precedence relation $\prec$ is given that decomposes $\mathcal{O}$ into chains, one for each job. Find a start time s_v for each $v \in \mathcal{O}$ such that the makespan defined as $\max_{v \in \mathcal{O}}(s_v + p_v)$ is minimal subject to the following constraints:*

(i)	$s_v \geqslant 0$	*for all $v \in \mathcal{O}$,*
(ii)	$s_w \geqslant s_v + p_v$	*for all $v, w \in \mathcal{O}$ with $v \prec w$, and*
(iii)	$s_w \geqslant s_v + p_v$ *or* $s_v \geqslant s_w + p_w$	*for all $v, w \in \mathcal{O}$ with $M_v = M_w$*

Constraint (i) implies no machine is available before time 0, constraint (ii) accounts for the binary precedence relation, i.e., if $v \prec w$ then w cannot start

before v is finished, and constraint (iii) implies that no machine can process two operations at the same time.

To apply local search we use the *disjunctive graph* introduced by Roy & Sussmann [1964]. This is a vertex-weighted mixed graph $G = (\mathcal{O}, A, E)$, where $A = \{(v, w) | v \prec w\}$ and $E = \{\{v, w\} | M_v = M_w\}$. The arc set corresponds to the binary precedence relation, and the edge set corresponds to a collection of cliques representing the operations that must be processed on the same machine. Each node $u \in \mathcal{O}$ has a weight given by p_u. A feasible solution uniquely corresponds to an acyclic directed graph $\bar{G} = (\mathcal{O}, A \cup \bar{E})$ that is obtained from G by orienting each edge of G and thus replacing its edge set by an arc set $\bar{E}$. The makespan of such a solution is given by the length of the longest path in $\bar{G}$, which can be computed in $O(|\mathcal{O}|)$ time in a straightforward way. Thus $\mathcal{S}$ can be chosen as the set of all acyclic directed graphs $\bar{G}$ that can be obtained from G, and the cost function computes the longest path length in $\bar{G}$.

A simple k-exchange neighborhood function can be defined as follows. Let i be a solution with corresponding weighted acyclic graph $\bar{G}_i = (\mathcal{O}, A \cup \bar{E}_i)$. For all $i \in \mathcal{S}$, $\mathcal{N}^{(k)}(i)$ is given by the set of weighted acyclic graphs $\bar{G}_j$ that are obtained from $\bar{G}_i$ by reversing the direction of k arcs in $\bar{E}_i$. Simply reversing an edge in $\bar{E}_i$ may lead to a cycle in the resulting graph, and thus to an infeasible solution. A disjunctive graph obtained by reversing an edge can be tested for acyclicity in $O(|A| + |E|)$ time by a simple depth-first search algorithm. It can be easily verified that the neighborhood graph obtained in this way is strongly connected. For more details the reader is referred to Van Laarhoven, Aarts & Lenstra [1992].

3.2 More intricate exchange neighborhoods

A major point of concern in local search is the observation that the algorithms may get stuck in poor local optima. It is generally known that, for small values of k, the simple k-exchange neighborhoods are easily explored but on average yield low-quality solutions. On the other hand, for large values of k, k-exchange neighborhoods can be expected to yield better solutions, but the time of $\Omega(n^k)$ needed to verify local optimality may not be worth the additional effort. Therefore, much research has concentrated on the construction of more intricate neighborhoods that can be explored without excessive computational effort. We now discuss a few steps in this direction.

One of the major achievements is the class of *variable-depth search* algorithms. These algorithms were first introduced by Kernighan & Lin [1970] for the uniform graph partitioning problem and by the same authors for the traveling salesman problem [Lin & Kernighan, 1973]. However, many other problems have subsequently been handled with the same type of algorithm. Here we discuss the uniform graph partitioning problem, which best illustrates the typical features of variable-depth search.

Recall the uniform graph partitioning problem of Definition 7. For a uniform partition (V_1, V_2) of V, the gain $g(a, b)$ in cost that is obtained by exchanging the

nodes $a \in V_1$ and $b \in V_2$ is given by

$$g(a,b) = \sum_{\substack{\{a,v\} \in E \\ v \in V_2 \setminus \{b\}}} w_{\{a,v\}} - \sum_{\substack{\{a,u\} \in E \\ u \in V_1}} w_{\{a,u\}} + \sum_{\substack{\{b,u\} \in E \\ u \in V_1 \setminus \{a\}}} w_{\{b,u\}} - \sum_{\substack{\{b,v\} \in E \\ v \in V_2}} w_{\{b,v\}}.$$

Note that the gain can be negative, zero, or positive.

The variable-depth search algorithm replaces a single 2-exchange by a well-chosen sequence of k 2-exchanges using the gain of an exchange to guide the search. Furthermore, the value of k may vary from iteration to iteration. For a given partition (V_1, V_2) of V with $|V_1| = |V_2| = n$, the variable-depth exchange neighborhood function defines a neighboring solution by the following steps:

- Step 1. Choose two nodes $a \in V_1$ and $b \in V_2$ such that $g(a,b)$ is maximal, even if that maximum is nonpositive.
- Step 2. Perform a *tentative* exchange of a and b.
- Step 3. Repeat Steps 1 and 2 n times, where a node cannot be chosen to be exchanged if it has been exchanged in one of the previous iterations of Steps 1 and 2. The sequence of 2-exchanges obtained in this way defines a sequence $g_1, g_2, \ldots, g_n$ of gains g_i, where g_i corresponds to the exchange of the nodes $a_i \in V_1$ and $b_i \in V_2$ in the ith iteration. The total gain after k exchanges equals

  $$G(k) = \sum_{i=1}^{k} g_i, \quad 0 \leqslant k \leqslant n.$$

  Note that, for $k = n$, V_1 and V_2 have been entirely interchanged, resulting in a total gain $G(n) = 0$.
- Step 4. Choose the value of k for which $G(k)$ is maximal. If $G(k) > 0$, the neighboring solution is given by the partition that is obtained from (V_1, V_2) by effectuating the *definite* exchange of $\{a_1, a_2, \ldots, a_k\}$ and $\{b_1, b_2, \ldots, b_k\}$. If $G(k) \leqslant 0$ then (V_1, V_2) has no neighboring solutions.

In the above neighborhood function each solution has at most one neighboring solution. The final solution, obtained by repeating the procedure until no favorable k-exchange is found anymore, is locally optimal with respect to the applied neighborhood function.

The above greedy procedure for finding an improving k-exchange for some $k \leqslant n$ avoids the exponential running time that a complete search would require. It is more open-minded than some such procedures might be, however, in that it does not require that all the interim k'-exchanges ($k' < k$) encountered be favorable themselves. The idea is that a few small steps in the wrong direction may ultimately be redeemed by large steps in the right direction. This sort of greedy, variable-depth approach can also be applied to other problems, and in particular has been applied to the traveling salesman problem in the algorithm of Lin & Kernighan [1973], described in Chapter 8 of this book. Two key problem-specific components of this approach are an appropriate *gain function* – for uniform graph partitioning, the change in the cut size – and a *locking rule* – for

uniform graph partitioning, the fact that no vertex is allowed to be exchanged twice in one sequence. The locking rule ensures that only a polynomial-length sequence of subexchanges can be made before no more subexchanges are possible. In most applications, the design of these two components will be less straightforward than in the case of uniform graph partitioning.

Another approach to the modification of simple exchange algorithms is to reduce the exchange neighborhoods in such a way that the efficiency of the algorithms is improved without affecting their effectiveness. Examples of such modifications for uniform graph partitioning are given by Fiduccia & Mattheyses [1982], who propose to split up a 2-exchange into two 1-exchanges by first selecting the best node to move from V_1 to V_2, and then the best node to move from V_2 to V_1, excluding the node that has already been exchanged. This idea was further refined by Dunlop & Kernighan [1985], who also made it applicable to the placement of standard cells in VLSI layout.

For the traveling salesman problem one can reduce the quadratic 2-exchange neighborhood to a linear-size neighborhood by requiring that one of the new edges links a city to one of its *nearest neighbors* [Lin & Kernighan, 1973; Bonomi & Lutton, 1984; Bentley, 1990a; Reinelt, 1992; Fredman et al., 1995]. For a detailed discussion, see Chapter 8 of this book.

We finally discuss some of the neighborhood reduction techniques for the job shop scheduling problem. Let $\bar{G}$ be a weighted directed graph representing a feasible schedule. It is obvious that reversing an arc not belonging to a longest path in $\bar{G}$ cannot reduce the longest path length, and it is not hard to see that reversing an arbitrary arc on a longest path in $\bar{G}$ always leads to an acyclic graph.

So, evidently, arc-reversal neighborhoods in job shop scheduling can be restricted to arc-reversal on a longest path. This not only leads to a substantial reduction of the size of the neighborhoods, but also renders the feasibility test superfluous. Furthermore, Van Laarhoven, Aarts & Lenstra [1992] prove the following interesting property.

Let $\bar{G}_{i_0}$ be a weighted directed graph representing a feasible schedule. Then there exists a finite sequence of graphs $\bar{G}_{i_1}, \ldots, \bar{G}_{i_m}$, each corresponding to a feasible schedule, such that $\bar{G}_{i_{k+1}}$ is obtained from $\bar{G}_{i_k}$ by exchanging one edge on a longest path, for $k = 0, 1, \ldots, m-1$, and $\bar{G}_{i_m}$ is optimal. Thus the restricted neighborhood graph is *weakly optimally connected*, i.e., there is a directed path from every node to an optimal node.

The neighborhood function can be further restricted by allowing only specific operations to be exchanged. For instance, Matsuo, Suh & Sullivan [1988] propose a neighborhood consisting of solutions in which not only the reversed arc (u, v) must be on a longest path but also at least one of the arcs $(p(u), u)$ or $(v, s(v))$ may not be on a longest path, where $p(u)$ and $s(v)$ denote the predecessor and successor operations of u and v, respectively. This condition further reduces the size of the neighborhoods, but also causes the corresponding neighborhood graph to be no longer weakly optimally connected.

Several authors have restricted this neighborhood still further or have given other neighborhood functions for job shop scheduling. For an overview the

reader is referred to Vaessens, Aarts & Lenstra [1996] and to Chapter 11 of this book.

4 ANALYSIS AND COMPLEXITY

Unless a local search algorithm employs an exact neighborhood function, it is generally not possible to give nontrivial bounds on the amount by which the cost of a local optimum deviates from the optimal cost. Furthermore, in general no nontrivial bounds on the time complexity of local search can be derived. However, in practice, many examples of local search algorithms are known that converge quickly and find high-quality solutions. In this section we discuss the performance of local search in more detail, concentrating on both practical and theoretical aspects.

In the performance analysis of combinatorial algorithms one usually distinguishes between the *average case* and the *worst case*. The analysis may be either theoretical or empirical, with *empirical worst-case analysis* taken to be the study of the robustness of an approach in practice. The performance of an approximation algorithm can be quantified by its *running time* and its *solution quality*. The running time is usually given by the number of CPU seconds the algorithm requires to find a final solution on a specified computer and operating system. The solution quality is typically measured by the ratio of its cost value to that of an optimal solution or to some easily computed bound on that optimal value.

4.1 Empirical results

Over the years many results have been published on the empirical performance of local search. A general conclusion is that local search algorithms can find good solutions for many problems of interest in feasible running times.

For uniform graph partitioning, Kernighan & Lin [1970] observed that their variable-depth search algorithm found solutions within a few percent of the optimum, and extrapolated the growth rate of the average running time to be $O(n^{2.4})$. The implementation of Fiduccia & Mattheyses reduced the average running time to $O(n \log n)$, without substantial loss of quality of the final solutions. More information about the performance of local search for uniform graph partitioning is given by Johnson et al. [1989].

For job shop scheduling, Van Laarhoven, Aarts & Lenstra [1992] report that simple exchange algorithms find solutions for instances with up to 15 jobs and 10 machines within 15% of the optimum with high frequency, which is substantially better than classical construction heuristics based on dispatching rules. The running times for the exchange algorithms are considerably larger than for the constructive algorithms. Applegate & Cook [1991] were able to improve on the results of Van Laarhoven, Aarts & Lenstra [1992], both in terms of solution quality and running time, using a combination of the *shifting bottleneck procedure* of Adams, Balas & Zawack [1988], limited backtracking, and a neighborhood based on revising the schedule on several machines.

Several authors improved on these results with other local search algorithms based on simulated annealing, genetic algorithms, and other approaches. A detailed overview is given by Vaessens, Aarts & Lenstra [1996]. Currently the best-performing approximation algorithms for job shop scheduling are the tabu search algorithm of Nowicki & Smutnicki [1996a], which uses simple exchange neighborhoods and backtracking to promising but rejected schedules, and the *guided local search* algorithm of Balas & Vazacopoulos [1998], which combines the shifting bottleneck procedure with variable-depth search based on larger exchange neighborhoods than were previously used. These algorithms can find solutions within 0.5% of optimal in a few minutes running times for instances with up to 100 jobs and 20 machines.

The traveling salesman problem is probably the best-studied problem for local search, and the scene of some of its greatest successes. Reiter & Sherman [1965] were the first to report on an extensive study involving simple exchange algorithms. They found optimal solutions for instances with up to 57 cities in less than 20 seconds of running time. Ever since, many new results have been published, reporting on the ongoing challenge to successfully handle increasingly large problem instances with local search. In summary, we have that algorithms based on the 2- and 3-exchange neighborhoods typically get within 3–6% of optimal, whereas the variable-depth search algorithm of Lin & Kernighan[1973] can get within 2%. Moreover, modern implementations of these algorithms have exploited data structures and neighborhood-pruning techniques (and powerful computers) to obtain surprisingly low running times, with even the slowest, Lin–Kernighan, taking less than an hour to handle 1 000 000 cities. Currently the best-performing approximation algorithm for the traveling salesman problem is the *iterated Lin–Kernighan* procedure of Johnson and coworkers [Johnson, 1990; Fredman et al., 1995], which can find solutions within 0.5% of optimal in a few minutes of running time for randomly generated instances with up to 1 000 000 cities. Similar results have been reported by Verhoeven, Swinkels, & Aarts [1995] for real-life instances with up to 100 000 cities.

4.2 Theoretical results

In the previous section we have argued that on average local search behaves quite well in practice. Good solutions are usually found in low-order polynomial running times. We now discuss a number of theoretical results showing the limitations of local search, at least from a worst-case point of view.

It has been known for quite some time that there are problem instances for which local search may require an exponential number of steps to converge to a final solution. Probably the best-known example is the *simplex algorithm* for linear programming [Klee & Minty, 1972]. However, the simplex algorithm uses an exact neighborhood and one might hope that finding a locally optimal solution is easier in those situations where local optima need not be globally optimal and hence many of them may be present. Furthermore, it might be possible to find locally optimal solutions in polynomial time by other more

sophisticated algorithms than local search. Unfortunately, the prospects are dim. The literature presents a number of 'bad examples'. Most of them hold for the traveling salesman problem, but there is no reason to expect that other NP-hard problems exhibit more favorable characteristics. Indeed, a general complexity theory for local search has been formulated that captures the characteristics of a broad class of problems. Before we outline this theory, we mention some of the bad examples.

Weiner, Savage & Bagchi [1976] proved that exact neighborhoods for the traveling salesman problem must have exponential size, provided they do not depend on the instances. This result did not rule out the existence of an exponential-size exact neighborhood for which an improving move could be found in polynomial time. However, Papadimitriou & Steiglitz [1977] showed that polynomially searchable exact neighborhoods cannot exist unless P = NP. They also constructed examples of instances with a single optimal solution and exponentially many second-best solutions that are locally optimal with respect to k-exchange neighborhoods, for $k < 3/8n$, while their costs are arbitrarily far from optimal [Papadimitriou & Steiglitz, 1978]. Also the more familar inexact neighborhoods have their worst-case drawbacks. Lueker [1975] has constructed instances and start solutions of the traveling salesman problem for which 2-exchange algorithms require an exponential number of steps. Examples of such bad worst-case behavior have also been reported for other problems, e.g., for the independent set problem [Rödl & Tovey, 1987].

Kern [1989] addressed the theoretical probabilistic behavior of local search, and showed that the 2-exchange algorithm for Euclidean instances of the traveling salesman problem with n points in the unit square with high probability converges within $O(n^{18})$ steps. This result has been improved by Chandra, Karloff & Tovey [1997]. For more details, see Chapter 8.

Other theoretical models for local search have been studied by Tovey [1985, 1986]. He considers classes of neighborhood graphs with a regular structure, e.g., the hypercube. For several types of random cost functions he proves low-order polynomial average-case running times. The worst-case results are invariably pessimistic. For a more detailed discussion of this topic, see Chapter 3.

Johnson, Papadimitriou & Yannakakis [1988] formalized the question of the worst-case behavior of local search algorithms by introducing the complexity class PLS of *polynomial-time local search* problems. This class and the underlying theoretical framework have proved a valuable tool in the analysis of local search. Informally speaking, PLS contains those local search problems for which local optimality can be verified in polynomial time. Furthermore, just as the NP-complete problems were defined as the hardest decision problems in NP, the *PLS-complete* problems are the hardest search problems in PLS, and many PLS-complete problems have been identified. None of these problems is known to be solvable in polynomial time, and if any one of them is solvable in polynomial time, then so are all problems in the class PLS.

To prove that a problem is PLS-complete, one must show that (i) it is in PLS, and (ii) there exists a known PLS-complete problem that can be PLS-reduced to

it. A PLS-reduction of a problem A to a problem B consists of two coordinated polynomial-time computable functions. The first function f maps instances X of A to instances $f(X)$ of B. The second function g maps local optima for $f(X)$ in B back to local optima for X in A. Roughly speaking, a PLS-reduction not only maps instances of A to those of B such that a solution to A corresponds one-to-one with a solution to B, as in NP-reductions, but it also maps the entire structure of the neighborhood graph of A to that of B such that the topology of both structures with respect to local optimality is the same. The generic PLS-complete problem is Flip, which is defined as follows. Given is a feedback-free Boolean circuit of AND, OR and NOR gates with m ordered inputs and n ordered outputs, where the cost of a solution, i.e., an input, is the binary number encoded by its corresponding output. The problem is to find an input whose cost cannot be decreased by changing a single input value.

Since its introduction, the class PLS has received considerable attention. A number of local search problems have been proved PLS-complete, including uniform graph partitioning with the Kernighan–Lin variable-depth neighborhood function [Johnson, Papadimitriou & Yannakakis, 1988], Satisfiability and 2-Sat with the analog of the above Flip neighborhood [Schäffer & Yannakakis, 1991], and a variant of the Lin–Kernighan neighborhood function for the TSP [Papadimitriou, 1992]. Other important contributions to the area were made by Krentel [1989, 1990]. For more examples and a more thorough discussion of the topic, see Chapter 2 of this book.

5 ADVANCED SEARCH STRATEGIES

Most of the local search algorithms discussed above have the advantage of being *generally applicable* and *flexible*, i.e., they require only a specification of solutions, a cost function, a neighborhood function, and an efficient method for exploring a neighborhood, all of which can be easily obtained for most problems. Nevertheless, poor local optima are found in several cases. To remedy this drawback – while maintaining the paradigm of neighborhood search – many people have been investigating the possibilities of broadening the scope of local search to obtain neighborhood search algorithms that can find high-quality solutions in possibly low running times. A straightforward extension of local search would be to run a simple local search algorithm a number of times using different start solutions and to keep the best solution found as the final solution. Several authors have investigated this *multistart* approach, but no major successes have been reported; see for instance Aarts & Van Laarhoven [1985a] and Johnson [1990] for the traveling salesman problem, and Aarts et al. [1994] for the job shop scheduling problem.

Instead of restarting with an arbitrary initial solution, one could consider an approach that applies multiple runs of a local search algorithm by combining several neighborhoods, i.e., by restarting the search in one neighborhood with a solution selected from another neighborhood. Such approaches are often called *multilevel*. An example is *iterated local search*, in which the start solutions of

subsequent local searches are obtained by modifying the local optima of a previous run. Examples for the traveling salesman problem are the *large-step Markov chains* approach of Martin, Otto & Felten [1992] and the *iterated Lin–Kernighan* algorithm of Johnson [1990] inspired by the Martin, Otto & Felten approach. Johnson's method uses multiple runs of the variable-depth search algorithm of Lin & Kernighan with a random 4-exchange to obtain new start solutions. For the traveling salesman problem this is currently the best-performing local search algorithm, even when running times are taken into account. See also Chapter 8 of this book.

In addition, there are several approaches that take advantage of search strategies in which cost-deteriorating neighbors are accepted. We mention *simulated annealing*, *tabu search*, *genetic algorithms*, and *neural networks*. We now give a brief introduction to each of these search strategies.

5.1 Simulated annealing

Simulated annealing [Kirkpatrick, Gelatt & Vecchi, 1983; Černý, 1985] is based on an analogy with the physical process of annealing, in which a pure lattice structure of a solid is made by heating up the solid in a heat bath until it melts, then cooling it down slowly until it solidifies into a low-energy state. From the point of view of combinatorial optimization, simulated annealing is a randomized neighborhood search algorithm. In addition to better-cost neighbors, which are always accepted if they are selected, worse-cost neighbors are also accepted, although with a probability that is gradually decreased in the course of the algorithm's execution. Lowering the *acceptance probability* is controlled by a set of parameters whose values are determined by a *cooling schedule*. A detailed discussion of the theory of simulated annealing is given in Chapter 4.

Simulated annealing has been widely applied with considerable success. The randomized nature enables asymptotic convergence to optimal solutions under certain mild conditions. Unfortunately, the convergence typically requires exponential time, making simulated annealing impractical as a means of obtaining optimal solutions. Instead, like most local search algorithms, it is normally used as an approximation algorithm, for which much faster convergence rates are acceptable [Aarts & Korst, 1989a].

5.2 Tabu search

Tabu search [Glover, 1989, 1990] combines the deterministic iterative improvement algorithm with a possibility to accept cost-increasing solutions. In this way the search is directed away from local minima, such that other parts of the search space can be explored. The next solution visited is always chosen to be a *legal neighbor* of the current solution with the best cost, even if that cost is worse than that of the current solution. Note the similarity to the choice mechanism used in the variable-depth search algorithm discussed in Section 3.2. The set of legal neighbors is restricted by a *tabu list* designed to prevent us going back to recently

visited solutions. The tabu list is dynamically updated during the execution of the algorithm. The tabu list defines solutions that are not acceptable in the next few iterations. However, a solution on the tabu list may be accepted if its quality is in some sense good enough, in which case it is said to attain a certain *aspiration level.*

Tabu search has also been applied to a large variety of problems with considerable successes; see for instance Hertz & de Werra [1991]. Tabu search is a general search scheme that must be tailored to the details of the problem at hand. Unfortunately, there is little theoretical knowledge that guides this tailoring process, and users have to resort to the available practical experience. A detailed discussion is given in Chapter 5.

5.3 Genetic algorithms

Another search strategy is based on a theme borrowed from the *genetic algorithms* approach introduced by Holland [1975]; see also Goldberg [1989]. Holland uses concepts from population genetics and evolution theory to construct algorithms that try to optimize the *fitness* of a *population* of elements through *recombination* and *mutation* of their genes. There are many variations known in the literature of algorithms that follow these concepts. As an example, we discuss *genetic local search*, which was introduced by Mühlenbein, Gorges-Schleuter & Krämer [1988]. The general idea of genetic local search is best explained by the following scheme:

- Step 1, *initialize.* Construct an initial population of n solutions.
- Step 2, *improve.* Use local search to replace the n solutions in the population by n local optima.
- Step 3, *recombine.* Augment the population by adding m offspring solutions; the population size now equals $n + m$.
- Step 4, *improve.* Use local search to replace the m offspring solutions by m local optima.
- Step 5, *select.* Reduce the population to its original size by selecting n solutions from the current population.
- Step 6, *evolute.* Repeat Steps 3 through 5 until a stop criterion is satisfied.

Evidently, *recombination* is an important step, since here one must try to take advantage of the fact that more than one local optimum is available, i.e., one must exploit the structure present in the available local optima.

The above scheme has been applied to several problems, and good results have been reported. The general class of genetic algorithms contains many other approaches, which may differ substantially from the genetic local search approach presented above. A detailed discussion of genetic algorithms is given in Chapter 6.

5.4 Neural networks

Recent advances in the design and manufacturing of integrated circuits have brought, within our reach, the construction of parallel computers consisting of

thousands of individual processing units. A direct consequence of these technological advances is the growing interest in computational models that support the exploitation of massive parallelism. *Connectionist models* [Feldman & Ballard, 1982] are computational models that are inspired by an analogy with the neural network of the human brain, in which massive parallelism is generally considered to be essential. The corresponding parallel computers are called *neural networks*, and the field of research *neural computing*.

A neural network consists of a network of elementary nodes (neurons) that are linked through weighted connections. The nodes represent computational units, which are capable of performing a simple computation, consisting of a summation of the weighted inputs, followed by the addition of a constant called the threshold or bias, and the application of a nonlinear response function. The result of the computation of a unit constitutes its output. This output is used as an input for the nodes to which it is linked through an outgoing connection. The overall task of the network is to achieve a certain network configuration, for instance a required input–output relation, by means of the collective computation of the nodes. This process is often called *self-organization*.

Over the years a large variety of neural network models have been proposed, which differ in the network architecture and the computational model used for self-organization. For detailed overviews, refer to the textbooks by Aarts & Korst [1989a], Hecht-Nielsen [1990], Hertz, Krogh & Palmer [1991], and Haykin [1994].

During the past decade, many researchers have investigated the use of neural networks for solving combinatorial optimization problems. Most of the work is motivated by the potential speedup obtained by massively parallel computation. Seminal papers in this area are by Hopfield & Tank [1985] and Baum [1986b]. For a more detailed overview of neural network approaches to combinatorial optimization, see Looi [1992]. In many neural network models, the self-organization of the network tries to find a stable network configuration. For models such as the *Hopfield network* [Hopfield, 1982] or the *Boltzmann machine* [Hinton & Sejnowski, 1986], the self-organization can be modeled in terms of local search. Here the issue of escaping from local optima is again of prime importance. In a number of cases it is resolved by applying probabilistic methods, as in simulated annealing. A detailed treatment of the subject is given in Chapter 7.

6 CONCLUSION

Over the past three decades, local search has grown from a simple heuristic idea into a mature field of research in combinatorial optimization. It provides a basis for a number of newly developed algorithmic approaches, including many of the currently popular *heuristics from nature*, such as simulated annealing, genetic algorithms, and neural networks. This book covers local search and its variants from theoretical and practical points of view. It is organized into two parts.

The first part deals with the theory of local search and describes the principal search strategies in the area. More specifically, Chapter 2 reviews results on the

computational complexity of local search, and Chapter 3 discusses worst-case and average-case analyses of local search as a function of assumptions about the structure of the neighborhoods. Next we have four chapters on the principal local search strategies: simulated annealing in Chapter 4, tabu search in Chapter 5, genetic algorithms in Chapter 6, and neural networks in Chapter 7.

The second part of the book presents a wealth of results on applications of local search to problems in management science and engineering: the traveling salesman problem in Chapter 8, vehicle routing in Chapters 9 and 10, machine scheduling in Chapter 11, VLSI design in Chapter 12, and code design in Chapter 13.

Local search is an active research area. Many recent developments and applications are not covered in this book. We mention the extensive bibliography compiled by Osman and Laporte [1996] as a source of further information.

Local Search in Combinatorial Optimization
Edited by E. Aarts and J. K. Lenstra
© 1997 John Wiley & Sons Ltd.

2

Computational complexity

Mihalis Yannakakis
Bell Laboratories, Lucent Technologies, Murray Hill

1 INTRODUCTION 19
2 PRELIMINARIES 22
3 THE CLASS PLS 26
3.1 Definition 26
3.2 Basic properties 27
3.3 PLS-reductions and PLS-completeness 29
3.4 A first PLS-complete problem 30
4 PLS-COMPLETE PROBLEMS 32
4.1 Definitions 32
4.2 Reductions 37
5 COMPLEXITY OF THE STANDARD LOCAL SEARCH ALGORITHM 42
5.1 The neighborhood and transition graphs 42
5.2 Tight PLS-reductions 43
5.3 Running time of local search heuristics 45
5.4 The standard local optimum problem 47
6 PARALLEL COMPLEXITY 49
6.1 Parallel complexity of the local optimality problem 49
6.2 Parallel complexity of local search problems 50
7 LOCAL VERSUS GLOBAL OPTIMALITY 52
8 CONCLUSIONS 54

1 INTRODUCTION

Local search is a widely used, general approach to solving hard optimization problems. An optimization problem has a set of *solutions* and a *cost function* that assigns a numerical value to every solution. The goal is to find an *optimal* solution, one that has the minimum (or maximum) cost. To obtain a local search heuristic for an optimization problem, one superimposes a *neighborhood structure* on the solutions, that is, one specifies for each solution a set of 'neighboring' solutions. The heuristic starts from some initial solution that is constructed by some other algorithm, or just generated randomly, and from then on it keeps moving to a better neighbor, as long as there is one, until finally it terminates at a *locally optimal* solution, one that does not have a better neighbor.

This scheme has been used successfully for several problems. The two important issues concerning a local search heuristic are (1) the quality of the obtained

solutions, i.e., how good are the local optima for the chosen neighborhood structure and how do they relate to the global optima; and (2) the complexity of the local search heuristic, i.e. how fast we can find local optima. There is a clear trade-off in the choice of the neighborhood structure: the larger (more general) the neighborhoods, the better will be the local optima, but it may be harder to compute them. Designing a good local search heuristic involves choosing a neighborhood that strikes the right balance. Although a few principles and techniques have been identified, the design and analysis of good local search heuristic has remained up to now mainly an experimental art.

Over the past few years there has been significant progress both on the experimental front – faster algorithms have been designed that find reasonable solutions in reasonable amounts of time for very large instances of classical optimization problems such as the traveling salesman problem – and on the theoretical front, with the development of a complexity theory of local search. In this chapter we will cover some of the theoretical work on local search.

Empirically, local search heuristics appear to converge usually rather quickly, within low-order polynomial time. There is not much analytical work supporting these experimental observations, and even analyzing the worst-case complexity of the heuristics is often a difficult task. Furthermore, even if the local search heuristic has exponential (worst-case) complexity, this does not preclude the possibility of finding local optima faster by other, possibly noniterative, methods. Linear programming is an interesting example: this problem can be viewed as a local search problem, where the 'solutions' are the vertices of a polytope and the neighborhood structure is given by the edges of the polytope. In this case local optimality coincides with global optimality, and the standard local search algorithm is Simplex, which has been shown to require an exponential number of iterations in the worst case (for most pivoting rules). However, linear programming can be solved in polynomial time by other direct methods such as the ellipsoid or interior point algorithms that do not even work with the vertices of the polytope [Khachiyan, 1979; Karmarkar, 1984; Grötschel, Lovász & Schrijver, 1988]. Thus, it is important to make a distinction between the complexity of the local search *problem* itself on the one hand, i.e., finding local optima by any means, and the complexity of the local search *heuristic* on the other hand, i.e., finding local optima by the standard iterative algorithm.

The complexity of finding locally optimal solutions for many interesting problems remains open, that is, we do not know whether it can be done in polynomial time or not. A complexity class was introduced for this reason by Johnson, Papadimitriou & Yannakakis [1988] to capture these problems. This class, called PLS (polynomial-time local search), contains the problems whose neighborhood can be searched in polynomial time; this is a minimal requirement that is satisfied by the neighborhoods used by the common local search heuristics. The class PLS lies somewhere between P and NP. Recent results show that many important local search problems are complete for PLS under an appropriately defined reduction, and that therefore PLS characterizes the complexity of local search problems in the same sense that NP characterizes the complexity of (hard)

combinatorial optimization problems. Furthermore, the PLS-completeness theory has enabled us also to analyze the complexity of many popular local search heuristics.

The rest of the chapter is organized as follows. Section 2 sets up a formal framework for local search and defines the associated complexity questions. We illustrate these concepts and issues with a few (hard and easy) classical combinatorial optimization problems, and the neighborhood structures that are associated with them in local search heuristics. There are several other search problems, which do not arise directly from optimization, but which can be expressed naturally as instances of local search. One example is the problem of finding stable configurations in neural networks in the Hopfield model. We describe some search problems of this type, where the goal is to find a local optimum for an appropriately defined cost function.

Section 3 defines the class PLS and relates it to the familiar complexity classes P and NP. We define PLS-reductions, a type of reduction that is suitable for local search problems because it relates the difficulty of finding local optima between two problems, and we prove that a certain generic problem is complete for PLS. Section 4 contains many natural PLS-complete problems associated with common local search heuristics for graph partitioning, the traveling salesman problem, and other optimization problems, as well as the stable configuration problem for neural nets. These results can be viewed as universality theorems on the computational power of different local search techniques. For example, neural nets form a universal model of computation for (polynomial) local search, meaning roughly that any local search algorithm for any problem that spends polynomial time per iteration can be implemented (with polynomial overhead) on a neural net by setting its weights appropriately. We will give some of the proofs that are of reasonable length, and refer the reader to the references for the more complex proofs.

Section 5 concerns the complexity of local search heuristics. We define a framework and a tighter notion of PLS-reductions, which allows us to show that many common heuristics have exponential complexity. We show also that the problem which is actually solved by the heuristics – given an initial solution find the specific final solution that will be computed by the heuristic – is in fact in many cases PSPACE-complete, and thus presumably harder than the problem of computing *some* local optimum (which is in PLS $\subseteq$ NP).

Section 6 discusses the parallel complexity of local search problems. This is most interesting for problems that can be solved in polynomial time, in which case we would like to know how much parallelism can help, and in particular whether the problem is in NC. Note that, if the costs are polynomially bounded, the standard local search heuristic converges in polynomial time because it decreases the cost in each iteration; this is the case if the numbers are small or given in unary, or if the problem is unweighted. It turns out that many simple unweighted local search problems are P-complete.

Finally we touch on the quality of the local optima. For a given optimization problem, the number and quality of the locally optimal solutions and their

relationship to the global optima depends on the complexity of the optimization problem. We formalize this dependence and summarize several results along these lines in Section 8. We conclude in Section 9.

2 PRELIMINARIES

A general computational problem Π has a set D_Π of instances (its domain), and for each instance $x \in D_\Pi$ there is a set $O_\Pi(x)$ of corresponding possible acceptable answers. An algorithm *solves* problem Π if on input $x \in D_\Pi$ it outputs a member y of $O_\Pi(x)$ (any member) or, in case $O_\Pi(x)$ is empty it reports that no such y exists. Inputs (instances) and outputs are represented as strings over a finite alphabet, which without loss of generality can be taken to be $\{0, 1\}$.

There are different types of computational problems depending on the size of the output sets $O_\Pi(x)$. The most general kind is a *search problem*, where every $O_\Pi(x)$ can have zero, one or more elements. If $O_\Pi(x)$ is nonempty for all $x \in D_\Pi$, the problem is called *total*. If $|O_\Pi(x)| \leqslant 1$ for all x, it is *functional*. An important special case of total functional problems is the set of *decision problems* where, for every instance, the (unique) answer is 'Yes' (1) or 'No' (0). For example, does a given graph have a Hamiltonian circuit?

A *combinatorial optimization problem* Π is a special kind of search problem. Every instance $x \in D_\Pi$ has a finite set $\mathscr{S}_\Pi(x)$ of solutions, and every solution $i \in \mathscr{S}_\Pi(x)$ has a cost $f_\Pi(i, x)$. The problem is, given an instance x, find a (globally) optimal solution $i^* \in \mathscr{S}_\Pi(x)$, i.e., one with maximum cost (for a maximization problem) or minimum cost (for a minimization problem). Of course, there may be more than one optimal solution with the same optimal cost, so this is in general a search problem. Computing the optimal cost is a functional problem. Note that the solutions and their costs are not listed explicitly but are instead determined implicitly from the input x. For example, consider the *graph partitioning problem*: given a graph $G = (V, E)$ with $2n$ nodes and for each edge a weight $w_e \in \mathbb{Z}^+$, find a partition of the node set V into two sets A, B with $|A| = |B| = n$ that minimizes the weight of the cut $w(A, B)$, i.e. the sum of the weights of the edges that have one endpoint in A and one endpoint in B. Here an instance x consists of (an encoding of) the graph G (for example represented by its adjacency matrix or a list of its edges) and the list of the edge weights w_e written in binary notation. The cost of a solution i (equipartition of the nodes into two sets) is determined from the input x and the solution i. For this reason x appears as an argument both for the set of *solutions* $\mathscr{S}$ and the cost function f. We will sometimes drop the subscript Π, when it is clear from the context which problem Π we are referring to.

A *local search problem* is obtained from an optimization problem by defining a *neighborhood function* on the solution set. Given an instance x, every solution $i \in \mathscr{S}_\Pi(x)$ is assigned a set $\mathscr{N}_\Pi(i, x) \subseteq \mathscr{S}_\Pi(x)$ of neighboring solutions. Note again that the neighbors are not listed explicitly for each solution i, but are determined indirectly from the instance x and the solution i. The problem is as follows: given an instance x, compute a *locally optimal solution* $\hat{\imath}$, i.e., a solution that does not have a strictly better neighbor (a neighbor with smaller cost for a minimization

problem, or greater cost for a maximization problem). Because an instance may have many locally optimal solutions, this too is a search problem.

A given optimization problem OP produces different local search problems for different neighborhood functions $\mathcal{N}$; we will use the notation $OP/\mathcal{N}$ to denote the corresponding local search problem. Although the term '*neighborhood*' suggests closeness in some sense, i.e., that neighboring solutions are obtained from each other by a small perturbation, this need not necessarily be the case. The neighborhood function could be quite complicated and does not need even to be symmetric, i.e., it is possible that i has a neighbor j, but j does not have i as its neighbor.

Consider the graph partitioning problem. A very simple neighborhood is the *Swap* neighborhood: a partition (A, B) of the node set V into two equal parts has as its neighbors all the partitions obtained by swapping a node of A with a node of B. A much more involved neighborhood is explored by the Kernighan–Lin heuristic [Kernighan & Lin, 1970]. This neighborhood is not symmetric and depends on the edge weights. The Kernighan–Lin heuristic moves from one partition (A, B) to a neighboring partition by a sequence of greedy swaps. In each step of the sequence, we choose to swap the best possible pair of nodes among those that have not been moved in the previous steps of the sequence, where '*best*' means that the swap produces the minimum cost differential, i.e., the weight of the cut decreases the most or increases the least; in case of a tie for best swap at a step, a tie-breaking rule is used to choose uniquely a pair of nodes to be swapped next. Thus, from the partition (A, B), a sequence of partitions (A_i, B_i), $i = 1, \ldots, n$, is generated where $|A_i - A| = |B_i - B| = i$. All these partitions are neighbors of the original partition (A, B). Obviously this is a more powerful neighborhood than the simple Swap neighborhood. Observe that, if a partition is locally optimal under Kernighan–Lin, it is locally optimal under Swap because otherwise its first neighbor would be better. Therefore, finding local optima under Kernighan–Lin is at least as hard as under Swap. It is rather surprising that, as we shall see, the two problems are actually polynomially equivalent. In Section 4 we will define a number of other common local search problems that are also equivalent.

In general, the more powerful the neighborhood, the harder it is to explore it and to find local optima, but presumably the optima have better quality. The most powerful neighborhood is an *exact* neighborhood, one in which every local optimum is also a global optimum. In this case the local search problem coincides with the optimization problem itself. In any optimization problem one could make local and global optima coincide by choosing large enough neighborhoods. The problem however is that it may then be hard to search the neighborhood, i.e., determine whether a solution is locally optimal and find a better neighbor if it is not. An essential consideration is that we must be able to search the neighborhood efficiently.

We typically think of local search in connection with NP-hard optimization problems. However, many classical polynomially solvable optimization problems can themselves be viewed as local search problems by defining an appropriate exact neighborhood structure that can be searched efficiently. This amounts

to proving a theorem of the following form: a solution is not (globally) optimal if and only if it can be improved by a certain kind of perturbation. We give below some examples of this kind; see Papadimitriou & Steiglitz [1982] for definitions and more information on these problems. As we shall see, these polynomial examples point to some important issues in the study of the complexity of local search.

- *Linear programming.* Geometrically, a vertex of the polytope of feasible solutions is not an optimal solution if and only if it is adjacent to another vertex of the polytope with better cost; that is, the adjacency of vertices on the polytope defines an exact neighborhood. Algebraically, assuming that the linear program is nondegenerate, a basic feasible solution is not optimal if it can be improved by exchanging a variable of the basis for another variable outside the basis. The local search algorithm corresponding to this neighborhood is Simplex.
- *Maximum matching.* A matching is not maximum if and only if it has an augmenting path. Thus, the neighborhood, where two matchings are neighbors if their symmetric difference is a path, is an exact neighborhood. Similar theorems hold for maximum weighted matching, minimum cost perfect matching, maximum flow, and minimum cost flow, and these characterizations form the bases of polynomial-time algorithms for solving these problems.
- *Minimum spanning tree.* A spanning tree is not optimal if and only if it can be improved by adding an edge to the tree and removing another edge in the (unique) cycle that is thus formed. Thus, the neighborhood, where two spanning trees are neighbors if one can be obtained from the other by exchanging one edge for another, is an exact neighborhood. Although the minimum spanning tree problem could be solved in principle by a local search algorithm based on this characterization, it is not a very efficient method; the problem is usually solved by direct algorithms such as Prim's or Kruskal's algorithm; see Aho, Hopcroft & Ullman [1974]; Papadimitriou & Steiglitz [1982].

Besides the cases of optimization problems with an exact neighborhood, there are several other examples of search problems that do not arise themselves from optimization, but which can be posed as suitable local search problems. In these cases, local optimization is not a compromise but is the goal itself; i.e., we wish to find a locally optimal solution, not necessarily the global optimum. The cost function is not inherent to these problems but is added as a means for proving the existence of the sought objects (i.e., that the search problem is total) and characterizing them.

For example, consider the following *Submatrix* problem proposed by Knuth: given an $m \times n$ matrix A with $m < n$ of rank m, find a (nonsingular) $m \times m$ submatrix B of A such that all entries of $B^{-1}A$ have absolute value at most one. Now consider the following local search problem. The solutions are the $m \times m$ nonsingular submatrices of A, the cost of a submatrix is the absolute value of its determinant, and two matrices are neighbors if one can be obtained from the other by exchanging a column. Using Cramer's rule, it is not hard to see that

a solution submatrix B is locally optimal if and only if all entries of $B^{-1}A$ have absolute value at most one. In particular this reformulation shows that such a submatrix always exists. The complexity of this submatrix problem is open: we do not know if it can be solved in polynomial time or if it is PLS-complete; we conjecture the former.

Another important example, where local optimization is used only as a proof tool, is finding *stable configurations* in neural networks in the Hopfield model [Hopfield, 1982]. We are given an undirected graph $G=(V, E)$ with a positive or negative weight w_e on each edge e and a threshold t_v for each node v (we can consider missing edges as having weight 0). A *configuration* assigns to each node v a state s_v which is either 1 ('on') or -1 ('off'). (Some authors use the values 1 and 0 instead of 1 and -1; the two versions are equivalent.) A node v is *happy* if $s_v = 1$ and $\sum_u w_{(u,v)}s_u + t_v \geqslant 0$, or $s_v = -1$ and $\sum_u w_{(u,v)}s_u + t_v \leqslant 0$. A configuration is *stable* if all the nodes are happy. The problem is to find a stable configuration for a given network. It is not obvious a priori that such a configuration exists; in fact, in the case of directed networks it may not exist [Godbeer, 1987; Lipscomb, 1987]. Hopfield showed that in the undirected case there is always a stable configuration [Hopfield, 1982]; see also Goles Chacc, Fogelman-Soulie & Pellegrin [1985]. To prove this he introduced a cost function $\sum_{(u,v)\in E} w_{(u,v)}s_u s_v + \sum_{v\in V} t_v s_v$ and argued that if a node is unhappy then changing its state will increase the cost; that is, the stable configurations coincide with the locally optimal solutions for this function with respect to the neighborhood that flips the state of a single node. As we shall see, the stable configuration problem is PLS-complete.

We summarize now the problems that we will be examining in the rest of this chapter. The framework consists of a set of instances D_Π, and for each instance $x \in D_\Pi$ there is a set of solutions $\mathscr{S}_\Pi(x)$, a cost function $f_\Pi(i, x)$, $i \in \mathscr{S}_\Pi(x)$, and a neighborhood function $\mathscr{N}_\Pi(i, x)$, $i \in \mathscr{S}_\Pi(x)$. The basic problem is the *local search problem* itself: given an instance x, compute a locally optimal solution. Note that we allow the use of any algorithm whatsoever, not necessarily a local search algorithm. This is very important, as shown by the linear programming example, and especially the spanning tree problem.

Besides the basic local search problem itself, we will discuss three problems related to the complexity of the corresponding local search heuristics. The first problem concerns the complexity of performing one iteration. This takes polynomial time for all common local search heuristics; of special interest is the parallel complexity.

Local optimality problem. Given an instance x and a solution $i \in \mathscr{S}_\Pi(x)$, determine if i is locally optimal, and find a better neighbor if it is not.

The other two problems concern the 'standard' local search algorithm associated with a local search problem. Given an instance x and an initial solution i_0 (constructed by some other algorithm or just generated randomly), the standard local search algorithm starts from i_0 and then moves iteratively from a solution to a better neighboring solution, as long as there is one, until it finally terminates at a locally optimal solution $\hat{i}$. The running time of the local search algorithm, in

particular whether it is polynomially bounded, is determined by the number of iterations. A given neighborhood function does not determine uniquely the local search algorithm. Some solutions may have more than one better neighbor, and a local search algorithm has freedom to choose any one of these neighbors to move to. A rule which assigns a better neighbor to each solution that is not locally optimal, we call it a *pivoting rule*. The choice of a pivoting rule may affect drastically the complexity of a local search algorithm. For example, consider the maximum flow problem with the neighborhood function corresponding to augmentation along a path: it is well known that if we choose the augmenting paths arbitrarily, then an exponential number of augmentations may be required, but if we always augment along shortest paths, the optimum will be found after a polynomial number of iterations [Papadimitriou & Steiglitz, 1982]. Thus, in examining local search algorithms, we want to analyze the complexity for different pivoting rules, and also try to characterize the best possible rule.

Running time problem: What is the worst-case time complexity (as a function of instance size over all instances and starting solutions) of the standard local search algorithm with a given pivoting rule?

The third problem concerns the computational complexity of the actual problem that is solved by a local search algorithm: a mapping from every starting solution to the specific locally optimal solution that is reached by the local search algorithm. How fast can we find this final solution if we are free to use any algorithm?

Standard local optimum problem: Given an instance x and an initial solution i_0, find the local optimum that would be produced by the standard local search algorithm starting from i_0.

3 THE CLASS PLS

3.1 Definition

The class PLS of *polynomial-time local search* problems is designed to capture the local search problems associated with usual local search heuristics. Such heuristics have some common properties, namely, it is possible to efficiently generate initial solutions to start the heuristics, to evaluate the cost of solutions, and to efficiently search the neighborhoods. Formally, PLS is defined as follows. Consider a local search problem Π. We assume that instances are encoded as binary strings, and that for each instance x, its solutions $s \in \mathscr{S}_\Pi(x)$ are also binary strings and have length bounded by a polynomial in the length of x; without loss of generality we can assume all solutions are encoded as strings of the same length $p(|x|)$. We also assume for simplicity that costs are nonnegative integers (the theory extends in a straightforward way to rational costs).

Definition 1 *A local search problem* Π *is in* PLS *if there are three polynomial-time algorithms* A_Π, B_Π, C_Π *with the following properties*:

1. *Given a string* $x \in \{0,1\}^*$, *algorithm* A_Π *determines whether* x *is an instance* ($x \in D_\Pi$), *and in this case it produces some solution* $s_0 \in \mathcal{S}_\Pi(x)$.
2. *Given an instance* x *and a string* s, *algorithm* B_Π *determines whether* $s \in \mathcal{S}_\Pi(x)$ *and if so,* B_Π *computes the cost* $f_\Pi(s,x)$ *of the solution* s.
3. *Given an instance* x *and a solution* s, *algorithm* C_Π *determines whether* s *is a local optimum, and if it is not,* C_Π *outputs a neighbor* $s' \in \mathcal{N}_\Pi(s,x)$ *with (strictly) better cost, i.e.,* $f_\Pi(s',x) < f_\Pi(s,x)$ *for a minimization problem, and* $f_\Pi(s',x) > f_\Pi(s,x)$ *for a maximization problem.*

All the common local search problems are in PLS, in particular the problems defined in the previous section. The above definition gives rise directly to a local search algorithm, which starts from the initial solution $s_0 = A_\Pi(x)$ generated by algorithm A_Π, and then applies repeatedly algorithm C_Π until it reaches a local optimum. If the number of different solution costs is polynomial – as would be the case with the unweighted versions of many optimization problems, such as graph partitioning without weights on the edges – clearly there will be at most a polynomial number of iterations and the algorithm will terminate in polynomial time. In the general case where there is an exponential range of solution costs (e.g., problems with binary weights), no a priori bound on the number of iterations exists better than exponential. In this case, we would like to know whether or not the local search problem can be solved in polynomial time.

3.2 Basic properties

Most of complexity theory is based on decision problems. In particular, the fundamental complexity classes P and NP are classes of decisions problems. These are usually sufficient to address the complexity of optimization problems. On the one hand, we can show that an optimization problem *OP* is 'easy' by exhibiting a polynomial-time algorithm that solves it. On the other hand, if the problem *OP* is 'hard', we can usually show this by transforming it to a related decision problem, *OP*-decision, and proving that the latter problem is NP-complete.

OP-decision: Given an instance x and a cost c, does there exist a solution $s \in \mathcal{S}_{OP}(x)$ with cost at least as good as c ($f_{OP}(s,x) \leqslant c$ for a minimization problem, $f_{OP}(s,x) \geqslant c$ for a maximization problem)?

Clearly, the *OP*-decision problem is no harder than *OP* itself, since an algorithm for *OP* can be used to answer *OP*-decision. If the *OP*-decision problem is NP-complete, the optimization problem *OP* is NP-hard, i.e., there is an algorithm for an NP-complete problem that uses an algorithm for *OP* as a subroutine and takes polynomial time, apart from the time spent executing the subroutine calls. For local search problems, no such transformation presents itself to a suitable decision problem that is no harder than the original search problem. It is also possible to ask yes–no questions such as, Does there exist a local optimum with better cost than a given value c? Or 'Is there a local optimum s that is

lexicographically greater than a given string y?' These questions might be more difficult than the local search problem itself, since it is not clear how to use an efficient algorithm for the local search problem to solve these decision problems. Thus, even if these decision problems are NP-complete, this does not mean that the local search problem is also hard. For this reason, the classes P and NP of decision problems are not sufficient to characterize the complexity of local search problems, and we have to study them properly as search problems.

We can define the search analogues P_S and NP_S of P and NP. The class NP_S is the class of search problems (relations) $R \subseteq \{0,1\}^* \times \{0,1\}^*$ that are polynomially bounded and polynomially recognizable, i.e., (1) if $(x,y) \in R$ then $|y|$ is polynomially bounded in $|x|$, and (2) there is a polynomial-time algorithm that, given a pair (x,y), determines whether (x,y) is in R. Such a search problem R is in P_S if there is a polynomial-time algorithm that solves it, i.e., given an instance x it either outputs a y such that $(x,y) \in R$ or reports (correctly) that no such y exists. Note that if R is in NP_S then there is a polynomial-time *nondeterministic algorithm* that solves it (i.e., recognizes the set of instances $D_R = \{x$: there is a y such that $(x,y) \in R\}$ and is such that every accepting computation outputs a y satisfying $(x,y) \in R$) by simply guessing for a given instance x a string y of the appropriate length and then testing whether $(x,y) \in R$. It is easy to see from the definitions that the following equivalence holds.

Proposition 1 $P_S = NP_S$ *if and only if* $P = NP$.

The class PLS lies somewhere between P_S and NP_S. On the one hand, it is not hard to see that any search problem R in P_S can be formulated as a PLS-problem. That is, we can define for every instance x of R a set of solutions $\mathcal{S}(x)$, a cost function $f(s,x)$, and a neighborhood function $\mathcal{N}(s,x)$ along with corresponding algorithms A, B, C satisfying the conditions in the definition of PLS, and such that $(x,y) \in R$ if and only if y is a local optimum for x. Simply let $\mathcal{S}(x) = \{y: (x,y) \in R\}$, and for all $y \in S(x)$ let $f(y,x) = 0$ and $\mathcal{N}(y,x) = \{y\}$. Algorithm A in Definition 1 is the polynomial algorithm that solves the problem R, algorithm B uses the algorithm that recognizes membership in R, and algorithm C is trivial.

On the other hand, any problem Π in PLS is also in NP_S: the relation $\{(x,y)$: y is a local optimum for $x\}$ is polynomially bounded by our assumption on the length of solutions, and polynomially recognizable because of algorithm C_Π in Definition 1. Thus, we have the following theorem.

Theorem 2 $P_S \subseteq PLS \subseteq NP_S$.

Does PLS coincide with one or the other end of this range, or is it properly in between? On the lower side, it is unclear whether PLS could be equal to P_S. Although we cannot rule it out a priori on the basis of current complexity theory, such a result would be remarkable and would require a general approach to finding local optima, at least as clever as the ellipsoid algorithm, since linear programming is in PLS, and in fact it is one of its 'simpler' and better-behaved members. For example, in linear programming local and global optimality

coincide, we can easily enforce uniqueness of local optima (by a standard simple perturbation of the costs), and the global optimization problem is in P and is not NP-hard, unlike other problems in PLS. On the upper side we have strong complexity theoretic evidence of proper containment. Note that trivially NP_S contains NP-hard problems. For example, the relation consisting of the pairs (x = a graph, y = a Hamiltonian circuit of x) is in NP_S. Solving this search problem is NP-hard since it contains the Hamiltonian circuit problem. The following fact shows that it is very unlikely that PLS contains NP-hard problems. The complexity class co-NP is the class of decision problems whose complement is in NP. The problem 'Is a given graph non-Hamiltonian?' belongs in co-NP. It is generally believed that NP $\neq$ co-NP.

Theorem 3 *If a* PLS-*problem* Π *is* NP-*hard, then* NP = *co*-NP.

Proof If Π is NP-hard, there is, by definition, an algorithm M for any NP-complete problem X that calls an algorithm for Π as a subroutine and takes polynomial time (apart from time spent in the subroutine calls). But then we can verify that a given string x is a 'no'-instance of X in nondeterministic polynomial time as follows: simply guess a computation of M on input x, including the inputs and outputs of the calls to the subroutine for Π. The validity of the computation of M outside of the subroutines can be checked in deterministic polynomial time (by our assumption of M); the validity of the subroutine outputs can be verified using the polynomial time algorithm C_Π, whose existence is implied by the fact that Π is in PLS, to check whether the output is really a locally optimal solution for the input. Thus the set of 'no'-instances of X is in NP, i.e., $X \in$ co-NP. Since X is NP-complete, this implies that NP = co-NP. □

3.3 PLS-reductions and PLS-completeness

Since we cannot use NP-hardness to relate local search problems to NP and argue that they are intractable, we will instead relate them to each other with suitable reductions and identify the hardest problems in PLS.

Definition 2 *Let* Π_1 *and* Π_2 *be two local search problems. A* PLS-*reduction from* Π_1 *to* Π_2 *consists of two polynomial-time computable functions h and g such that*

(a) *h maps instances x of* Π_1 *to instances* $h(x)$ *of* Π_2,
(b) *g maps (solution of* $h(x)$, x*) pairs to solutions of x, and*
(c) *for all instances x of* Π_1, *if s is a local optimum for instance* $h(x)$ of Π_2, *then* $g(s, x)$ *is a local optimum for x.*

If there is such a reduction, we say that Π_1 PLS-*reduces to* Π_2.

It is easy to see that PLS-reductions have the standard desirable properties: they compose, and they allow us to relate the difficulty of one problem to that of another.

Proposition 4 *If* Π_1, Π_2, *and* Π_3 *are problems in* PLS *such that* Π_1 PLS-*reduces to* Π_2 *and* Π_2 PLS-*reduces to* Π_3, *then* Π_1 PLS-*reduces to* Π_3.

Proposition 5 *If Π_1 and Π_2 are problems in* PLS *such that Π_1* PLS-*reduces to Π_2, and if there is a polynomial-time algorithm for finding local optima for Π_2, then there is also a polynomial-time algorithm for finding local optima for Π_1.*

We say that a problem Π in PLS is PLS-*complete* if every problem in PLS can be PLS-reduced to it. We prove below PLS-completeness for a generic problem that is the basis of further reductions. In the next section we will see many other natural PLS-complete problems.

3.4 A first PLS-complete problem

We describe the ingredients of the problem, which we call *Circuit/Flip*. An instance is (the encoding of) a combinational (feedback-free) Boolean circuit composed of AND, OR, and NOT gates. Let x be such a circuit with m inputs and n outputs. The set of solutions $\mathscr{S}(x)$ consists of all binary strings of length m, i.e., all possible input vectors. The neighborhood $\mathscr{N}(s, x)$ of the solution s consists of all strings of length m that have Hamming distance 1 from s, i.e., a strings that can be obtained from s by flipping exactly one bit. The cost of a solution s is defined as the output vector of the circuit for input s, interpreted as a number written in binary notation; i.e., $f(s, x) = \sum_{j=1}^{n} 2^{j-1} y_j$, where y_j is the jth output of the circuit with input s, indexed from right to left. We can define the problem either as a maximization or as a minimization problem; as we shall see in a moment, the two versions are equivalent. Intuitively, the local search problem asks for an input such that the output cannot be improved lexicographically by flipping a single input bit.

It is easy to see that Circuit/Flip is in PLS. In Definition 1 we let algorithm A return the all-one vector of the appropriate length m, we let algorithm B check that the given string s has length m and then we let it evaluate the circuit x with input s, and we let algorithm C evaluate the circuit x for all input vectors within Hamming distance 1 of s (there are only m of them). Algorithm C returns one of the vectors if that vector has better cost.

The maximization and minimization versions are equivalent, both as optimization and as local search problems. To convert an instance of one to an instance of the other that has the same global and local optima, simply add an extra level of logic to the circuit that flips all the output variables (changes 1's to 0's and vice versa). In particular, this transformation is clearly a PLS-reduction from one version to the other. We will sometimes use the prefix Max- or Min- to clarify which version of Circuit/Flip we are referring to.

Theorem 6 *Both the maximization version and the minimization version of Circuit/Flip are* PLS-*complete.*

Proof Since the two versions are PLS-reducible to each other, it suffices to prove the result for Min-Circuit/Flip. We shall show that every local minimization problem Π in PLS can be PLS-reduced to Min-Circuit/Flip; a similar argument can be used to show that every local maximization problem can be PLS-reduced to Max-Circuit/Flip, which as we noted can be further reduced to Min-Circuit/Flip.

Let L be a local minimization problem in PLS. We can assume without loss of generality that, for each instance x, all solutions are strings of length $p(|x|)$, no two of which are within Hamming distance 1; we can easily ensure this, for example, by duplicating every bit. We will first PLS-reduce L to an intermediate PLS-problem Q that has the same instances as L, but which has the same solutions and neighborhood function as Circuit/Flip; i.e., its solutions are all strings of a certain length and two solutions are neighbors if and only if their Hamming distance is 1. All the complexity of Q is in its cost function. Let x be an instance of L, and $p = p(|x|)$ the length of its solutions. We have the same instance x in Q. The solutions for x in Q are all strings of length $2p+2$. Only some of these strings have low enough cost to be candidates for local optima. For each solution u of L, we have the following possible candidates:

(a) $vu11$, where v is any string of length p; the cost is $f_Q(vu11, x) = (2p+4) f_L(u, x) + h + 2$, where h is the Hamming distance between v and u ($h = 0$, i.e., $v = u$ is allowed).
(b) $uu10$, with cost $f_Q(uu10, x) = (2p+4) f_L(u, x) + 1$.
(c) $uu00$, with cost $f_Q(uu10, x) = (2p+4) f_L(u, x)$.
(d) $uv00$, where u is not a local optimum and v is a string on the shortest Hamming path from u to its better neighbor $w = C_L(u, x)$ returned by the algorithm C_L. The cost is $f_Q(uv00, x) = (2p+4) f_L(w, x) + (p+2) + h + 2$, where h is the Hamming distance between v and w ($h = 0$, i.e., $v = w$ is allowed).
(e) $uw10$, where $w = C_L(u, x)$ as above, with cost $f_Q(uw10, x) = (2p+4) f_L(w, x) + (p+2) + 1$.

The rest of the strings of length $2p+2$ are 'noncandidates' and their cost is determined as follows. Recall from Definition 1 that there is a polynomial-time algorithm B_L that computes the cost $f_L(s, x)$ for a given instance x and solution s. Since $|s| = p(|x|)$, the running time of B_L, and hence also the number of bits of $f_L(s, x)$, is bounded by a polynomial in $|x|$, say $q(|x|)$. Thus, $f_L(s, x) \leqslant 2^{q(|x|)}$. Let $Z = (4p+4)(2^{q(|x|)})$, and let a be the standard solution returned by algorithm A_L. If s is a noncandidate solution and h is its Hamming distance from $aa11$, then $f_Q(s, x) = Z + h$. Thus, there is a downhill path from any noncandidate solution to $aa11$, and therefore only candidate solutions can be locally optimal. The polynomial algorithms A, B, and C for Q follow in a straightforward way from the corresponding algorithms for L.

To prove that this is a PLS-reduction from L to Q we need to show that we can recover a local optimum of L from any local optimum of Q. In fact we will show something stronger, which will also be useful later when we analyze the complexity of local search heuristics: the locally optimal solutions of Q are precisely the strings of the form $uu00$ such that u is a local optimum for L; furthermore, starting from any candidate solution, a local optimization algorithm for Q will follow a unique forced path that simulates the standard local search algorithm for L that uses algorithm C_L in each iteration to move from a solution to a better neighbor. Suppose that we are at a candidate solution of the form $vu11$. If $v \neq u$ we will first change the first component v one bit at a time until it becomes equal to u; this

improves the Hamming distance h and hence the cost. Note that no other change leads to improvement; in particular, changing a bit of u will turn u into a nonsolution u' of L (because we assume that any two solutions of L are at Hamming distance 2 or more), hence the new string $vu'11$ is not a candidate solution and has higher cost. Next we will move from $uu11$ to $uu10$ and then to $uu00$. If u is a locally optimal solution for L, then $uu00$ is locally optimal for Q. If u is not locally optimal for L, we will change the second occurrence of u one bit at a time until it becomes equal to $w = C_L(u, x)$; note that $f_L(w, x) \leqslant f_L(u, x) - 1$ because w is a better solution than u in L and the costs are integer. Observe also that flipping any other bits will lead to a worse solution. From $uw00$ we will move to $uw10$ and then to $uw11$ to start a new cycle. Thus, the only locally optimal solutions of Q are the strings of the form $uu00$ such that u is a local optimum for L.

To complete the proof, we PLS-reduce Q to Circuit/Flip. This is relatively straightforward, given the known relationship between algorithms and circuits: if we have an algorithm M, we can construct efficiently for each input size m a Boolean circuit that evaluates M on inputs of size m and the size of the circuit is polynomial in the running time of M. Note that Q has the right solution set and neighborhood function, and all its complexity resides in its cost function. Given an instance x of Q, construct a polynomial-size circuit $h(x)$ with $2p(|x|) + 2$ inputs that computes $f_Q(s, x)$ for all solutions s of x, with the outputs of the circuit encoding the answer in binary notation. This can be done because the algorithm B_Q that computes costs in Q runs in polynomial time. It is obvious by construction that the instances x of Q and $h(x)$ of Circuit/Flip have the same locally optimal solutions. □

We remark that in the definition of Circuit/Flip we could have used any other complete Boolean basis, instead of {AND, OR, NOT}, for example only NOR gates or only NAND gates. The same PLS-completeness result holds (the last step of the proof can be carried out for any complete basis). In further reductions from Circuit/Flip it is sometimes more convenient to use circuits over different bases.

4 PLS-COMPLETE PROBLEMS

In this section we will present a number of local search problems that are PLS-complete. We will define the problems, then describe some of the proofs. PLS-completeness proofs are similar in spirit to NP-completeness proofs, except that they are usually a great deal more delicate and difficult; some proofs span ten or more pages. Here we shall describe some of the simpler reductions.

4.1 Definitions

Graph partitioning

We defined two neighborhoods for this problem in Section 2, Swap and Kernighan–Lin. We will define now two other neighborhoods that are based on

the local search algorithm proposed by Fiduccia & Mattheyses [1982]. This algorithm is a variant of the Kernighan–Lin heuristic that has smaller running time in each iteration. Dunlop & Kernighan [1985] implemented both heuristics in a practical setting; they reported that the Fiduccia–Mattheyses heuristic generally reaches a local optimum faster than Kernighan–Lin, but sometimes finds inferior local optima. This heuristic uses also a sequence of steps to move from one partition to a neighboring one, but the steps are somewhat different, in that vertices are moved from one side to the other one at a time. Let $G = (V, E)$ be an edge-weighted graph with $2n$ nodes and (A, B) a partition of the nodes into two equal sets. The *Fiduccia–Mattheyses* (*FM*) neighborhood of this partition (solution) contains all solutions reachable by a sequence of (at most n) steps, where each step consists of the following two substeps: in the first substep, we examine all the nodes that have not moved since the beginning of the sequence, and choose to move the best such node from one side to the other; in the second substep, we move from the opposite side the best as-yet-unmoved node, so that the partition becomes balanced again. In both substeps, 'best' means that the move produces the smallest decrease (or largest increase) in the weight of the cut. In case of ties for best move in a substep, there is a tie-breaking rule that determines which node to move; the PLS-completeness result holds regardless of the tie-breaking rule.

We can define also another much simpler neighborhood for graph partitioning, which we will call *FM-Swap*: each partition has just one neighbor, the partition obtained after the first step of the Fiduccia–Mattheyses heuristic. Thus, FM-Swap bears in some sense the same relationship to the Fiduccia–Mattheyses neighborhood as Swap to Kernighan–Lin.

We have defined graph partitioning as a minimization problem, i.e., we wish to minimize the weight of the cut. There is also a maximization version where we want to maximize the weight of the cut. Corresponding heuristics and neighborhoods can be defined in a straightforward way for the maximization version. The two versions are equivalent, both as optimization problems and as local search problems under all of the neighborhoods we defined. The reduction is very simple: given an instance for one version consisting of a graph $G = (V, E)$ and edge weights w_e, $e \in E$, let the instance for the other version consist of the complete graph K with weights $w'_e = W - w_e$, where $W = \max_{e \in E} w_e$ and we take $w_e = 0$ if $e \notin E$. Equivalence follows from the following observation. For every partition of the nodes into two equal sets of size n, the weight of the cut in one version is equal to $n^2 W$ minus the weight of the cut in the other version.

Traveling salesman problem (TSP)

We are given the complete graph on n nodes (the 'cities') with positive integer weights $w(e)$ on its edges (the 'distances'), and we want to find the least-weight tour that passes exactly once through each city. If the weights satisfy the triangle inequality, the problem is called the *metric TSP*. A special case is when the cities are points on the plane, and the weights of the edges are the Euclidean distances between the points. The simplest neighborhood is 2-*Opt*: replace two edges (a, b),

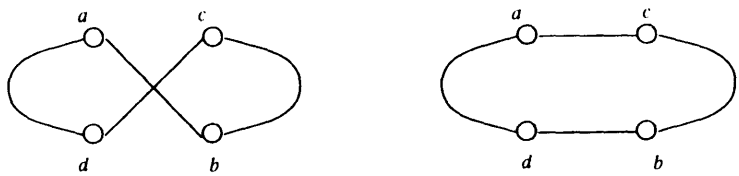

Figure 2.1 The 2-opt neighborhood

(c, d) by two other edges (a, c) and (b, d) to form another tour (Figure 2.1). For example, in the planar Euclidean case, if the two edges (a, b) and (c, d) cross each other, this replacement always yields a better tour because of the triangle inequality. Note, however, that on the plane, even a tour that does not self-intersect may have a cost-improving 2-opt move. This neighborhood can be extended to 3-opt, and more generally, k-opt, where we replace at most k edges of the current tour by k new edges [Lin, 1965].

The Lin–Kernighan heuristic allows the replacement of an arbitrary number of edges in moving from a tour to a neighboring tour, where again a complex greedy criterion is used in order to permit the search to go to an unbounded depth without an exponential blowup [Lin & Kernighan, 1973]. We can sketch the basic idea as follows. Given a tour, we can remove an edge (a, b) to obtain a Hamilton path with endpoints a and b. Regard one of the endpoints as fixed, say a, and the other, b, as variable. If we add an edge (b, c) from the variable endpoint, a cycle is formed; there is a unique edge (c, d) incident to c whose removal breaks the cycle, producing a new Hamilton path with a new variable endpoint d (Figure 2.2). This operation is called a *rotation*. We can always close a tour by adding the edge connecting the fixed endpoint a with the current variable endpoint d. A move of the Lin–Kernighan heuristic from one tour to a neighbor consists of first removing an edge to form a Hamilton path, then performing a sequence of 'greedy' rotations, and finally reconnecting the two endpoints to form a tour. There are different variants of this basic scheme depending on how exactly the rotation in each step is chosen, and on the

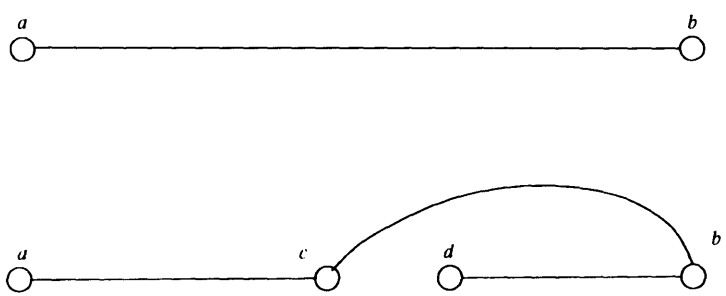

Figure 2.2 A rotation

restrictions on edges to enter and leave the tour. We define more precisely a variant, which was shown PLS-complete by Papadimitriou [1992]; we will call it LK′.

Let T be a tour. All 2-opt and 3-opt neighbors of T are included in the LK′ neighborhood. In addition, for every pair of adjacent edges $x_1 = (a, b) \in T$, $y_1 = (b, c) \notin T$ of the complete graph with $w(y_1) < w(x_1)$, we have a sequence of neighboring tours $T_1, T_2, \ldots$ defined as follows. Let $H = T - x_1$ be the Hamilton path with fixed endpoint a and free endpoint b. Perform the first rotation that adds the edge y_1 and deletes a (uniquely determined) edge x_2 to form a new Hamilton path H_1, then close the path to obtain the first neighboring tour T_1. Edges x_1 and x_2 are marked 'ineligible' and cannot reenter the tour during the sequence.

For the ith step, $i = 2, \ldots$, consider all eligible edges $y_i \notin H_{i-1}$ incident to the variable endpoint such that $\sum_{j=1}^{i} [w(y_j) - w(x_j)] < 0$; if none exists, the sequence is terminated. If there are some such edges, choose among them that edge y_i whose addition to H_{i-1} and rotation leads to the least expensive new Hamilton path H_i (with ties broken lexicographically); i.e., $w(y_i) - w(x_{i+1})$ is minimized, where x_{i+1} is the (unique) edge that has to be removed after adding y_i. Edge x_{i+1} becomes ineligible. The ith neighbor T_i is the tour obtained by closing the path H_i. Observe that the sequence terminates after less than n^2 steps because once an edge leaves the tour it becomes ineligible and cannot reenter. This concludes the formal description of the neighborhood LK′.

The main difference with the Lin–Kernighan algorithm as described by Lin & Kernighan [1973] is that in LK′ an edge is allowed to enter the tour and later depart again, whereas in the original algorithm it cannot: the sets of departing edges x_i and entering edges y_i are required there to be disjoint. It is open whether the PLS-completeness holds for the original Lin–Kernighan neighborhood with this requirement. Note also that the variant discussed in Chapter 8 is somewhat different: in this variant once an edge enters the tour it cannot leave, although an edge that leaves the tour is allowed to reenter. Under this restriction, a sequence terminates after n steps (instead of n^2) because at most n new edges can enter the tour.

Max-Cut

An instance consists of a simple undirected graph $G = (V, E)$ with positive weights on the edges. A feasible solution is a partition of V into two sets A, B (not necessarily of equal size). The cost is defined as for graph partitioning to be the weight of the cut (A, B), which we wish to maximize. The maximization and minimization versions here are not equivalent. In fact the minimization version, Min-Cut, can be solved in polynomial time, whereas Max-Cut is NP-hard. The simplest neighborhood for this problem is the *Flip* neighborhood, in which two solutions (partitions) are neighbors if one can be obtained from the other by moving a single vertex from one side of the partition to the other side.

Maximum satisfiability

In maximum satisfiability (Max-Sat) we have a Boolean formula in conjunctive normal form (CNF) with a positive integer weight for each clause. A solution is an assignment of 0 or 1 to all the variables. Its cost, to be maximized, is the sum of the weights of the clauses that are satisfied by the assignment. The restriction of Max-Sat to instances with at most k literals in each clause is called *Max-kSat*. The simplest neighborhood is the *Flip* neighborhood, where two solutions (assignments) are neighbors if one can be obtained from the other by flipping the value of one variable.

Not-all-equal maximum satisfiability

An instance of not-all-equal maximum satisfiability (NAE Max-Sat) consists of clauses of the form $NAE\ (a_1, \ldots, a_k)$, where each a_i is a literal (i.e., a variable or its negation) or a constant, 0 or 1. Such a clause is satisfied if its constituents do not all have the same value. Each clause is assigned a positive integer weight. A solution is again an assignment of 0 or 1 to all the variables, and its cost (to be maximized) is the sum of the weights of the satisfied clauses. The Flip neighborhood is defined as for Max-Sat. The restriction of NAE Max-Sat to those instances that contain at most k literals in each clause is called *NAE Max-kSat*. The restriction to instances without negation, i.e., where the clauses do not contain any negative literals, is called *Pos NAE Max-kSat*.

In analogy with graph partitioning and TSP, we can define also a variable-depth neighborhood for Max-Cut, Max-Sat, and NAE Max-Sat, which we will call also Kernighan–Lin. The neighbors of a solution (partition in Max-Cut, assignment in the satisfiability problems) are all solutions that are obtained by a sequence of flips, where in each step we choose the most profitable flip of a node or variable that has not been flipped before. Again, ties are broken according to some tie-breaking rule. Of course, this neighborhood is more powerful than Flip: if a solution is locally optimal under this neighborhood, it is also locally optimal under Flip. So, finding local optima for Kernighan–Lin is harder than for Flip.

Finally, recall the *stable configuration* problem for neural networks, which we defined in Section 2. We summarize in the next theorem the fact that all these problems are PLS-complete.

Theorem 7 *The following problems are* PLS-*complete*:

1. *Graph partitioning under the Kernighan–Lin neighborhood (for any tie-breaking rule)* [*Johnson, Papadimitriou & Yannakakis*, 1988].
2. *Graph partitioning under the following neighborhoods: (a) Swap, (b) Fiduccia Mattheyses, (c) FM-Swap* [*Schäffer & Yannakakis*, 1991].
3. *Traveling salesman problem under the k-Opt neighborhood for some constant k* [*Krentel*, 1989].
4. *Traveling salesman problem under the LK′ (Lin–Kernighan) neighborhood* [*Papadimitriou*, 1992].

5. *Max-Cut, under the flip neighborhood* [*Schäffer & Yannakakis*, 1991].
6. *Max-2Sat and Pos NAE Max-3Sat under the Flip neighborhood* [*Krentel*, 1990; *Schäffer & Yannakakis*, 1991].
7. *Stable configuration for neural networks* [*Schäffer & Yannakakis*, 1991].

In the TSP with the k-Opt neighborhood (for some k), Krentel [1989] does not give an explicit value for k. A straightforward calculation from the reduction as described in the paper would lead to a huge value of k. However, doing the reduction very carefully seems to bring down the value to $k=8$, and it may conceivably be possible to reduce this further to $k=6$ [Krentel, 1994]. Further reductions to say $k = 2$ or $k = 3$ would require substantially new ideas.

The PLS-completeness of Max-Sat with an unbounded number of literals under the Flip neighborhood was first proved by Krentel [1990]. The result for Max-2Sat (2 literals per clause) was shown by Schäffer & Yannakakis [1991], and this problem along with Pos NAE Max-3Sat/Flip (or Max-Cut/Flip) are simple problems that are useful starting points for further reductions.

4.2 Reductions

We will describe now some of the reductions. We will give a complete proof for the PLS-completeness of graph partitioning under the Kernighan–Lin neighborhood. We will not prove completely any of the other results – the complete proofs are quite complicated and long. However, we will present simple reductions from Pos NAE Max-3Sat/Flip to the other problems in parts 2, 5, 6, and 7 of Theorem 7. Figure 2.3 shows the sequence of reductions for the problems of Theorem 7; the solid arrows indicate the reductions that we will present here. We will start by proving completeness for Pos NAE Max-3Sat under the Kernighan–Lin neighborhood, and then reduce this to graph partitioning.

Lemma 8 *Pos NAE Max-3Sat with the Kernighan–Lin neighborhood* (*with any tie-breaking rule*) *is* PLS-*complete.*

Proof We reduce from the Max-Circuit/Flip problem; it is convenient in particular to use the version where all the gates are NOR gates. Let C be such a circuit with set of inputs $x_1, \ldots, x_m$ and outputs $y_1, \ldots, y_n$; we use x and y to denote the vector of inputs and outputs. We wish to find an x such that flipping any single bit does not produce a lexicographically greater y.

We first modify C to a circuit C' that computes in addition to the output vector y, a vector $z = (z_1, \ldots, z_m)$ that is a better neighbor of x in case x is not a local optimum, and $z = x$ if x is a local optimum. It is easy to do this modification: add n copies of C, where the ith copy C_i computes the value of the output vector y for the ith neighbor of the input vector x (i.e., the one that differs in the ith bit), and then on top of them we add logic that compares the outputs of C and the C_i's and computes accordingly the output vector z. If there is more than one better neighbor, we can design the circuit to arbitrarily choose one of them, for example the one that flips the earliest bit, or the one that gives the greatest improvement. Let

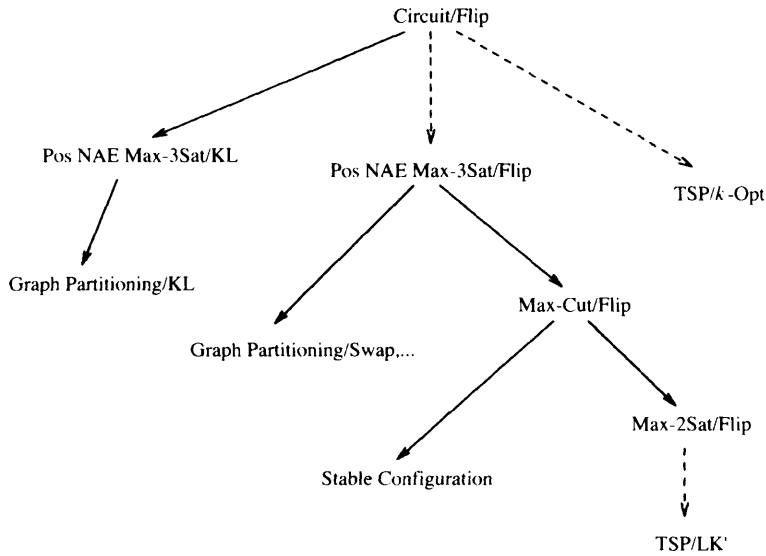

Figure 2.3 The sequence of reductions for the problems of Theorem 7

N be the number of gates, assume without loss of generality that $N > 6$, and let $M = N^{2N}$.

We construct an instance ϕ of Pos NAE Max-3Sat as follows. Partition the gates of C' into levels, where the level of every gate is one more than that of its highest level input; the input variables x_i are considered to be at level 0. For every input x_i, we have three variables x_i, x'_{i1}, x'_{i2}, and for every gate g_i, $i = 1, \ldots, N$ (including the output gates y and z), we have two variables g_i, g'_i; we identify variable g_i with the output of the gate. The primed variables are supposed to be the complements of the unprimed. We have clauses NAE (x_i, x'_{i1}), NAE (x_i, x'_{i2}) and NAE (g_i, g'_i) of weight M. Fore each gate $a = \text{NOR}\ (b, c)$, we have three NAE clauses: (a, b, c), $(a, b, 1)$ and $(a, c, 1)$ with weight $M \cdot N^{-2l}$, where l is the level of the gate. For each output variable y_j we have a clause NAE $(y_j, 0)$ of weight 2^j. Finally, for each $i = 1, \ldots, m$ we have a clause NAE (x'_{i1}, z_i) of weight 1. This concludes the construction.

We claim that an assignment to the variables of ϕ is locally optimal if and only if (i) the primed variables are the complements of the unprimed, (ii) the values of the gate variables are consistent with the circuit, and (iii) the input part x is a locally optimal solution to the circuit problem. To show PLS-completeness we only need the 'only if' part, so we leave the 'if' part to the reader. Suppose that we have an assignment that violates (i). Then we can flip the primed variable to gain weight M, while losing at most a clause of weight 1.

Suppose that (i) holds, but (ii) does not, and let us determine the first greedy flip of Kernighan–Lin. Flipping any variable will lose at least weight M ($2M$ for an input variable x_i) from the clause connecting a primed and unprimed variable.

Let $g_i = \text{NOR}(b, c)$ be a gate at level l that is inconsistent, i.e., $g_i = \text{OR}(b, c)$ in the assignment. The three clauses corresponding to a gate were designed so that they are all satisfied if and only if the values are consistent with the gate. If we flip the gate variable g_i, we will gain at least weight $M \cdot N^{-2l}$ from one of these clauses, but we might lose weight from gates at higher levels that now become inconsistent. The latter weight we might lose is certainly no more than $N \cdot M \cdot N^{-2(l+1)} \leqslant 1/2M \cdot N^{-2l}$. Thus, the loss from flipping g_i is between $M(1 - N^{-2l})$ and $M(1 - 1/2N^{-2l})$. Flipping an inconsistent gate at higher level will cause a higher loss, whereas flipping any consistent gate, primed variable or input x_i will lose weight at least M. It follows that in the first step we will flip an inconsistent gate variable g_i at the lowest level. It is easy to check that in the second step we will flip the corresponding variable g'_i, for a net total gain.

Suppose that (i) and (ii) hold, but the input x is not a locally optimal solution to the circuit problem. Then $z_i = \neg x_i = x'_{i1}$ for some i, and $z_j = x_j$ for $j \neq i$. Flipping variable x'_{i1} loses the clause NAE (x_i, x'_{i1}) of weight M and gains NAE(x'_{i1}, z_i) of weight 1. Flipping any other variable costs at least weight M, thus Kernighan–Lin will first flip x'_{i1}. In the second step it will flip x_i and then x'_{i2}. From then on it will propagate the new values upwards fixing the values of the gates, bottom up, level by level. At the end it will arrive at a new consistent assignment, which has greater weight because it corresponds to a better solution of Circuit/Flip and the weight is reflected in the clauses NAE$(y_j, 0)$. □

Lemma 9 *Pos NAE Max-3Sat with the Kernighan–Lin neighborhood* PLS-*reduces to graph partitioning with the Kernighan–Lin neighborhood.*

Proof We will reduce to the maximization version of graph partitioning, which as we noted is equivalent to the minimization version. Let ϕ be a Pos NAE Max-3Sat formula with variables $x_1, \ldots, x_m$, and let L be the total weight of all the clauses. Construct a graph G that contains two nodes x_i, x'_i for each variable x_i, and two more nodes y, z for the constant 0 and two nodes y', z' for the constant 1. There is an edge (x_i, x'_i) for each $i = 1, \ldots, m$, and also edges connecting both 0 nodes to both 1 nodes, all of weight $3L$. For every 2-clause NAE(a, b) of weight w we include an edge (a, b) of weight w, where for the constants 0 or 1 we use the node y or y'. For a 3-clause NAE (a, b, c) of weight w, we include edges (a, b), (b, c), (a, c) each of weight $w/2$. If a pair of nodes occurs in several clauses, we add the weights that arise from all these clauses.

We claim that a partition (A, B) of the nodes into two equal parts is locally optimal if and only if (i) both 0 nodes are on the same side, the 1 nodes are on the other side, and for each i the nodes x_i and x'_i are on different sides, and (ii) the assignment where a variable x_i is 1 if and only if node x_i is on the same side as the 1 nodes, is locally optimal for ϕ. We will only prove the 'only if' part. Assume that (i) does not hold. Suppose that A contains a pair u, u' of corresponding nodes. Since the other side B has the same size, it also contains a pair v, v'. Swapping u and v gains weight at least L. Suppose that corresponding nodes u, u' are on opposite sides, but the two 0 nodes are split, say $y \in A$ and $z \in B$, and thus $y' \in B$ and $z' \in A$; swapping z and z' improves the weight.

So, assume that (i) holds. Let us call such a partition reasonable. There is a 1–1 correspondence between reasonable partitions and truth assignments. Note that if a clause NAE (a, b, c) is satisfied then exactly two of the three corresponding edges are in the cut, but if it is not satisfied then none of its edges are in the cut. It follows easily then that the weight of a partition is equal to the weight of the corresponding assignment plus $3L(m+4)$. When considering greedy swaps from a reasonable partition, we only need to restrict attention to swaps involving a corresponding pair of variable nodes x_i, x_i' because the other swaps incur a huge cost. Swapping x_i and x_i' corresponds to flipping variable x_i. (Had we included only one node for each constant, swapping them would also be reasonable, which would correspond to flipping all the variables simultaneously.) Thus, the Kernighan–Lin neighborhood of a reasonable partition consists of a sequence of reasonable partitions corresponding to a sequence of flips. Claim (ii) then follows from the 1–1 correspondence of reasonable partitions with truth assignments and the fact that the correspondence is weight-preserving up to an additive factor. □

Pos NAE Max-3Sat is in fact PLS-complete under the much smaller Flip neighborhood. The proof of this result is quite complicated and we will not give it here; it can be found in Schäffer & Yannakakis [1991]. Once we have shown this, completeness follows easily for several more problems.

Lemma 10 *There are* PLS-*reductions from Pos NAE Max-3Sat/Flip to the following problems*: (a) *graph partitioning with the Swap, FM-Swap or Fiduccia–Mattheyses neighborhood*; (b) *Max-Cut/Flip*; (c) *Max-2Sat/Flip*; (d) *stable configuration.*

Proof (a) The construction of Lemma 9 works for (the maximization version of) all three neighborhoods. The arguments for Swap are essentially contained in the proof of Lemma 9. The arguments for FM-Swap are very similar, and the Fiduccia–Mattheyses neighborhood dominates FM-Swap.

(b) Let ϕ be a weighted Pos NAE Max-3Sat formula. The reduction is very similar to that of Lemma 9. We construct a graph G that has one node for each variable of ϕ and in addition two more nodes labeled with the constants 0 and 1. We have an edge (0, 1) with 'huge' weight $3L$, where L is the total weight of all the clauses of ϕ. In addition, for every clause of length two, NAE (x, y) with weight w in ϕ, where x and y are variables or constants, we include an edge (x, y) in G with the same weight w. For every length-3 clause NAE (x, y, z) with weight w, we include three edges (x, y), (y, z), (x, z), with weight $w/2$ each; again if the same edge arises from many clauses, we add the weights. Call a partition of G reasonable if nodes 0 and 1 are on opposite sides. A reasonable partition corresponds to a truth assignment, where a variable x is 0 or 1 depending on whether it is on the same side with node 0 or 1. Clearly, if nodes 0 and 1 are on the same side, we can increase the weight of the cut by moving one of them to the other side. Thus, a locally optimal partition must be reasonable. Furthermore, moving a variable node from one side to the other in a reasonable partition corresponds to flipping

the variable in the truth assignment. It follows that a locally optimal partition of G induces a locally optimal assignment for ϕ.

Note incidentally that Max-Cut is essentially identical to the special case of Pos NAE Max-3Sat where all the clauses have length 2 and there are no constants; hence a reduction in the opposite direction holds trivially. Given a weighted graph G for Max-Cut, construct a formula ϕ that has a variable for each node of G and has a clause NAE (x, y) for each edge (x, y) of G with the same weight. A truth assignment for the variables of ϕ induces in a natural way a partition of G with the same weight, namely the partition which contains on one side the nodes that correspond to true variables and on the other side the nodes that correspond to false variables.

(c) Since Max-Cut is a special case of Pos NAE Max-3Sat, it is easier to start reductions from Max-Cut. Let G be a weighted graph for Max-Cut/Flip. We construct a weighted 2Sat formula ϕ that has one variable for every node of G. If G has an edge (x, y) with weight w, we include in ϕ clauses $(x \vee y)$ and $(\bar{x} \vee \bar{y})$ each of weight w. Note that if in a truth assignment we have $x = y$ then only one of these two clauses is satisfied, whereas if $x \neq y$ then both clauses are satisified. A truth assignment of ϕ induces a partition of G, which contains on one side the nodes corresponding to true variables and on the other side the nodes corresponding to false variables. The weight of the satisfied clauses is equal to the weight of the cut plus the total weight of all the edges. Thus, flipping a variable in the 2Sat instance ϕ changes the weight of the assignment by the same amount as the weight of the cut is changed by moving the corresponding node to the other side. Therefore, a locally optimal truth assignment induces a locally optimal partition.

(d) Let G be a weighted graph for Max-Cut/Flip. The instance of stable configuration consists of the same graph G, where all the vertices have threshold 0, and the weight of every edge is the negative of its weight in G. Let W be the total weight of the edges of G. A configuration for the neural network induces a partition of G, where one side contains the vertices in state $+1$ and the other side the vertices in state -1. Flipping the state of a vertex in a configuration corresponds to moving the vertex to the other side in the induced partition of G. An easy calculation shows that the weight of a configuration is equal to $2T - W$, where T is the weight of the cut of the induced partition of G. It follows that a stable configuration induces a locally optimal partition of G. □

The stable configuration problem for neural networks where all the edge weights are negative and all thresholds are 0 is equivalent to Max-Cut/Flip; if the thresholds are arbitrary, it is equivalent to $s-t$ Max-Cut/Flip, where we have also two distinguished nodes s, t that have to be on different sides of the partition. If all the weights are positive, the stable configuration problem is equivalent to Min-Cut/Flip or $s-t$ Min-Cut/Flip, depending on whether or not the thresholds are 0; see also Bruck & Goodman [1988], Godbeer [1987], Lipscomb [1987]. In the case of positive weights the problem can be solved in polynomial time because, as is well known, one can find even the globally optimal solutions for

Min-Cut and $s-t$ Min-Cut in polynomial time [Papadimitriou & Steiglitz, 1982].

Another special case when one can find locally optimal solutions in polynomial time is Max-Cut/Flip for cubic graphs [Loebl, 1991; Poljak, 1995]. Note that finding the globally optimal solution in this case is NP-hard.

The reductions for the traveling salesman problem are quite complicated. Papadimitriou [1992] proves the PLS-completeness of TSP/LK′ via an intricate reduction from Max-2Sat/Flip. Krentel [1989] first proves that Max-Sat/Flip is PLS-complete even if there is both a bounded number of literals in each clause and every variable occurs in a bounded number of clauses. Then he reduces this restricted version of Max-Sat/Flip to the TSP under the k-Opt neighborhood for some fixed k. We refer the reader to the references for the detailed proofs.

5 COMPLEXITY OF THE STANDARD LOCAL SEARCH ALGORITHM

In this section we will be concerned with the running time of local search *algorithms*. We will see how the theory of PLS-completeness can be adapted to study this issue. We will also address the complexity of the standard local optimum problem. We need first some definitions.

5.1 The neighborhood and transition graphs

Definition 3 *Let* Π *be a local search problem and let* x *be an instance of* Π. *The* neighborhood graph $NG_\Pi(x)$ *of the instance* x *is a directed graph with one node for each feasible solution to* x, *and with an* arc $s \to t$ *whenever* $t \in N_\Pi(s, x)$. *The* transition graph $TG_\Pi(x)$ *is the subgraph that includes only those* arcs $s \to t$ *for which the cost* $f_\Pi(t, x)$ *is strictly better than* $f_\Pi(s, x)$ *(i.e., greater if* Π *is a maximization problem, and smaller if* Π *is a minimization problem). The* height *of a node* v *is the length of the shortest path in* $TG_\Pi(x)$ *from* v *to a sink (a vertex with no outgoing arcs). The* height *of* $TG_\Pi(x)$ *is the largest height of a node.*

We will be concerned mainly with the transition graph. First, note that this is an acyclic graph. The cost induces a topological ordering of the nodes: the arcs are directed from worse to better nodes. The local optima are the sinks of the graph. The transition graph $TG_\Pi(x)$ represents the possible legal moves for a local search algorithm on instance x; starting from some initial node (solution) v, the standard local search algorithm moves from node to node, following some path to a sink. The length of the path is the number of iterations of the algorithm, which determines its running time. The precise path that is followed (and hence the complexity) is determined by the chosen *pivoting rule*; at each node that is not a sink, the pivoting rule selects one of the outgoing arcs to follow. The height of a node v is a lower bound on the number of iterations needed by the standard algorithm even if it uses at each step the best possible pivoting rule; this rule may not be computable in polynomial time in the size of the instance (in general the transition graph has an exponential number of nodes).

Analyzing the complexity of particular local search algorithms is generally not a simple task. This is well illustrated by the example of linear programming, with the Simplex algorithm as its corresponding local search algorithm. The complexity depends on the pivoting rule. Several papers have examined the complexity of Simplex and have constructed bad examples for many natural pivoting rules (such as steepest descent and others) showing that they lead to an exponential number of iterations [Klee & Minty, 1972; Jeroslow, 1973]. However, some specific pivoting rules are still not well understood. For example, we do not know the running time of the random pivoting rule which, at each suboptimal solution, simply selects a better neighbor at random, and whether in fact it might guarantee a polynomial number of iterations; the only known lower bound is a quadratic bound that was shown recently [Gärtner & Ziegler, 1994]. In general, it is still an open problem whether there is a rule that turns simplex into a polynomial-time algorithm. The best upper bound is roughly $2^{O(\sqrt{d \log m})}$, where d is the number of variables and m the number of constraints [Kalai, 1992; Matousek, Sharir & Welzl, 1992].

The related Hirsch conjecture asserts that every (bounded) polytope always has small diameter, at most $m-d$ for a (bounded) polytope in dimension d with m facets; see Klee & Kleinschmidt [1987]. This claim is known to be false for unbounded polyhedra; in the bounded case it has been shown for some polytopes but in general it remains open. Note that the polytope for a linear program corresponds to the neighborhood graph. It is known that the *monotone* version of the Hirsch conjecture (i.e., if the cost has to improve in each step) is not true: Todd [1980] has presented an example where the height of the transition graph is greater than $m-d$; that is, regardless of the pivoting rule, the number of iterations must exceed $m-d$, but the lower bound in this example is still only linear, approximately $4/3(m-d)$. The best upper bound (for both the monotone and the unrestricted case) is roughly $m^{\log d+1}$ [Kalai, 1992].

If a local search problem has instances where the transition graph has exponential height, the standard local search algorithm will take exponential time in the worst case, regardless of how it chooses better neighbors. This turns out to be the case with all the problems that have been shown PLS-complete. The notion of PLS-reductions that we defined is not sufficient to prove this, but it can be strengthened in an appropriate way.

5.2 Tight PLS-reductions

Definition 4 *Let P and Q be two local search problems, and let (h, g) be a* PLS-*reduction from P to Q. We say that the reduction is* tight *if for any instance x of P we can choose a subset R of feasible solutions for the image instance $y = h(x)$ of Q so that the following properties are satisfied:*

1. *R contains all local optima of y.*
2. *For every solution p of x we can construct in polynomial time a solution $q \in R$ of y such that $g(q, x) = p$.*

3. *Suppose that the transition graph of y, $TG_Q(y)$ contains a directed path from $q \in R$ to $q' \in R$, such that all internal path nodes are outside R, and let $p = g(q, x)$ and $p' = g(q', x)$ be the corresponding solutions of x. Then either $p = p'$ or $TG_P(x)$ contains an arc from p to p'.*

It is easy to see that tight reductions compose. We will usually refer to the solutions in R as the *reasonable* solutions of the instance y. Tight reductions allow us to transfer lower bounds on the running time of the local search algorithm from one problem to another.

Lemma 11 *Suppose that P and Q are problems in* PLS, *and that h, g define a tight* PLS-*reduction from problem P to problem Q. If x is an instance of P and $y = h(x)$ the image instance of Q, then the height of $TG_Q(y)$ is at least as large as the height of $TG_P(x)$. Thus, if the standard algorithm of P takes in the worst-case exponential time, then so does the standard algorithm for Q.*

Proof Let x be an instance of P, let $TG_P(x)$ be its transition graph, and let p be a solution (node) whose height is equal to the height of $TG_P(x)$. Let $y = h(x)$, and let $q \in R$ be a solution of y such that $g(q, x) = p$. We claim that the height of q in $TG_Q(y)$ is at least as large as the height of p in $TG_P(x)$. To see this, consider a shortest path from q to a sink of $TG_Q(y)$, and let the nodes of R that appear on this path be $q, q_1, \ldots, q_k$. Let $p_1, \ldots, p_k$ be the images under g of these solutions, i.e., $p_i = g(q_i, x)$. From the definition of a tight reduction, we know that q_k is a local optimum of y, and thus p_k is a local optimum of x. Also, for each i, either $p_i = p_{i+1}$ or there is an arc in $TG_P(x)$ from p_i to p_{i+1}. Therefore, there is a path of length at most k from vertex p to a sink of $TG_P(x)$. □

All the reductions that we have presented are tight. Consider for example the reduction of Theorem 6 for Circuit/Flip. We first reduced an arbitrary PLS-problem L to an intermediate problem Q, then we reduced Q to Circuit/Flip. In the reduction from L to Q we can take the set R of reasonable solutions to be the set of all solutions of the form $uu00$, where u is a solution for L. As we proved in Theorem 6, this set R includes all the local optima of Q and $uu00$ is locally optimal for Q if and only if u is locally optimal for L. Furthermore, we argued that if a solution $uu00$ is not locally optimal, then there is a unique cost-improving path from it that leads to a solution $ww00$ where w is a better neighbor of u in L, and all the intermediate nodes on the path are not in R. It follows that the conditions in Definition 4 are satisfied, and the reduction from L to Q is tight. The reduction from Q to Circuit/Flip preserves solutions, costs and neighborhoods (it only changes the representation of the instance), hence it is trivially tight, by taking R to be the set of all solutions.

Consider the reduction of Lemma 8 from Circuit/Flip to Pos NAE Max-3Sat with the Kernighan–Lin neighborhood. We let R be the set of assignments to the variables of the constructed formula ϕ in which (i) the primed variables are the complements of the unprimed, and (ii) the values of the gate variables are consistent with the circuit. We showed in Lemma 8 that the locally optimal assignments are exactly those that satisfy these conditions, and the input part x is

locally optimal for the instance C of Circuit/Flip. Furthermore, if x is not locally optimal for C, then Kernighan–Lin will move to a new consistent assignment whose input part is a better neighbor z of x, so this choice of R shows that the reduction is tight.

In the reduction of Lemma 9 from Pos NAE Max-3Sat to graph partitioning, we let R be the set of partitions of the nodes into two equal parts that we called there reasonable, i.e., the partitions in which both 0 nodes are on the same side and the 1 nodes are on the other side, and for each i the nodes x_i and x_i' are on different sides. It follows easily again from the proof of Lemma 9 that this choice of R satisfies the conditions of Definition 4.

It can be shown similarly that the reductions of Lemma 10, as well as all the other known PLS-reductions are tight.

5.3 Running time of local search heuristics

To prove that the worst-case running time of the standard local search algorithm for the tightly PLS-complete problems is exponential, it suffices to exhibit one problem in PLS that has this property.

Lemma 12 *There is a local search problem in PLS whose standard algorithm takes exponential time.*

Proof Consider the following artificial minimization problem. For any instance x of size n, the solution set consists of all n-bit integers $0, \ldots, 2^n - 1$. For each solution i, its cost is i, and if $i > 0$ it has one neighbor, $i - 1$. Thus, there is a unique local and global minimum, namely 0, and the transition graph is a path from $2^n - 1$ down to 0. The local search algorithm starting at $2^n - 1$ will follow this path. □

Theorem 13 *The standard local search algorithm takes exponential time (in the worst case) for the following problems, regardless of the tie-breaking and pivoting rules used:*

- *Graph partitioning under the neighborhoods (a) Kernighan–Lin, (b) Swap, (c) Fiduccia–Mattheyses, (d) FM-Swap.*
- *Traveling salesman problem under the k-Opt neighborhood for some constant k, and the LK′ (Lin–Kernighan) neighborhood.*
- *Max-Cut, Max-2Sat and Pos NAE Max-3Sat under the Flip neighborhood.*
- *Stable configuration for neural networks.*

Note that the exponential bounds hold for any pivoting rules, including randomized rules and rules that may not be implementable in polynomial time.

Outside this general approach to proving bounds on the complexity of local search heuristics, there had been very few results, usually based on ad hoc methods and concerning individual pivoting rules. For the problem of finding stable configurations in neural networks, Goles Chacc & Olivos [1981] studied the algorithm where in each iteration *all* nodes that are not happy flip their state

simultaneously in parallel. They constructed examples on which this algorithm takes exponential time to converge. Haken & Luby [1988] studied the local search algorithm with the steepest-descent rule – in each iteration we flip the state of an unhappy node that yields the most improvement in the cost function – and showed that it has exponential complexity. Haken [1989] then found a direct construction, independently of Schäffer & Yannakakis [1991], for which every pivoting rule takes exponential time to converge.

On the positive side, Poljak [1995] proved that in the Max-Cut problem restricted to cubic graphs with positive and negative edge weights, the Flip local search algorithm with any pivoting rule takes $O(n^2)$ iterations. His basic approach was to prove that, for any given weighting of the edges of a cubic graph, there exists another edge weighting such that all the weights are small, at most $O(n)$, and the local search algorithm essentially cannot tell the difference between the two weightings. Since the graph has $O(n)$ edges, the number of iterations in the case of the small edge weighting is $O(n^2)$. The quadratic bound is tight.

In the case of the traveling salesman problem (with the triangle inequality), Lueker [1976] constructed instances where the 2-Opt heuristic can take an exponential number of iterations under a particular pivoting rule. Chandra, Karloff & Tovey [1997] extended the construction to k-Opt for all fixed $k > 2$. By Theorem 13, for sufficiently large k, the exponential complexity is unavoidable for any pivoting rule [Krentel, 1989]. It is not known whether this is true also in the case of 2-Opt or 3-Opt, or whether there is a way of performing the improvements so that these algorithms will converge in (worst-case) polynomial time.

A positive result for the case of the TSP on the plane with a weaker neighborhood was shown by Van Leeuwen & Schoone [1980]. Note that if a tour on the plane has two edges that cross, we can improve its cost by performing the 2-change that uncrosses them. Thus, a 2-Opt tour cannot have any crossing edges. Note that performing a 2-change to eliminate a crossing may introduce new crossings. Van Leeuwen and Schoone proved that $O(n^3)$ uncrossings suffice to transform any tour on the plane to a tour that does not have any crossings. Eliminating all edge crossings is not sufficient for a tour on the plane to be 2-Opt. We do not know if there is a 2-Opt algorithm on the plane that converges in polynomial time in the worst case.

For random point sets on the plane we have such bounds. Kern [1989] proved that for points uniformly distributed in the unit square, the 2-Opt heuristic converges with high probability in polynomial time (his bound was quite high: $O(n^{16})$). More recently, Chandra, Karloff & Tovey [1997] improved this, showing that the expected running time of 2-Opt is $O(n^{10})$ for the Euclidean metric, and $O(n^6)$ for the L_1 metric.

In practice, local search heuristics have been observed to converge very quickly in many cases. This situation is similar to the observed behavior of the Simplex algorithm. Several papers have considered probabilistic models of linear programming and analyzed the performance of Simplex, proving that the average complexity is bounded by a low-order polynomial. The probabilistic analysis of

sophisticated local search heuristics (e.g., Lin–Kernighan) appears to be a difficult problem. Tovey [1985, 1986] analyzed an abstract model for local search problems, where the neighborhood structure is an n-dimensional hypercube, whose vertices correspond to the solutions. The cost function induces an orientation of the hypercube from worse to better solutions. Tovey considered various natural probability distributions on the possible orientations; for example, one such distribution assigns the same probability to all possible orderings of the vertices according to cost. He proved then that the expected complexity of the local search heuristic under these distributions is bounded by a low-order polynomial. See Chapter 3 for further details.

Fischer [1995] examines a different question regarding the running time of a local search heuristic: Given a solution s and a number k in unary, does there exist a local optimum within k steps of s? (Is the height of s at most k?) She shows that this question is NP-complete for various local search problems: Max-Cut, Max-2Sat and Pos NAE Max-3Sat under the Flip neighborhood, TSP under the 2-Opt neighborhood. This does not apply automatically to all the PLS-complete problems: for any problem, if one fixes the pivoting rule of the heuristic, the local search problem does not change (the set of local optima remains the same), but one can determine easily whether there is a local optimum within k steps by simply running the heuristic, since k is given in unary.

When the bound on the number of steps (the height) k is given in binary, we can show that the above question of the existence of a local optimum within k steps is PSPACE-complete for the problems of Theorem 13 using their tight PLS-reductions and the methods of Section 5.4.

5.4 The standard local optimum problem

This is the problem that is actually solved by a local search heuristic: given an instance x and an initial solution s_0, find the specific locally optimal solution that is constructed by the heuristic at the end.

The standard local optimum problem is generally a harder problem than the local search problem itself because it asks for a specific local optimum, as opposed to just any one. Indeed it turns out that, for all the problems that we discussed, finding the standard local optimum is a PSPACE-complete problem, as compared to 'mere' PLS-completeness of the local search problem itself (recall that PLS is somewhere between P and NP) [Papadimitriou, Schäffer & Yannakakis, 1990].

First, observe that for any PLS-problem, the standard local search algorithm uses only polynomial space: at each iteration it only needs to remember the current solution (which is polynomially bounded by the definition of PLS) and find a better neighbor, if there is one, which takes polynomial time and hence space. Thus, for any PLS-problem, the corresponding standard local optimum problem is in PSPACE.

The PSPACE-hardness of the standard local optimum problem for all the PLS-complete problems that we have discussed holds regardless of the pivoting

rule used by the local search heuristic. This follows from the fact that the following 'easier' task is PSPACE-hard: given an instance x and an initial solution s_0, find a sink of the transition graph $TG(x)$ that is reachable from s_0. In the following we will refer to this task as the standard local optimum problem, so that we do not need to worry about individual pivoting rules. The PSPACE-hardness can be proved again unformly using tight PLS-reductions. For this we need to show that (1) tight reductions transfer the computational difficulty of the standard local optimum problem, and that (2) there is a PLS-problem for which finding the standard local optimum is a PSPACE-complete problem.

Lemma 14 *If there is a tight* PLS-*reduction from a local search problem P to a problem Q, then there is a polynomial-time reduction from the standard local optimum problem for P to the standard local optimum problem for Q.*

Proof Suppose that the functions h, g define a tight PLS-reduction from problem P to problem Q. Let x be an instance of P and p an initial solution of x. We can find a locally optimal solution (sink of $TG_P(x)$) that is reachable from p as follows. Construct $y = h(x)$, the image instance of Q, and compute a solution $q \in R$ of y such that $g(q, x) = p$. Then solve the standard local optimum problem for Q to find a locally optimal solution $\hat{q}$ of y that is reachable from q, and return the solution $\hat{p} = g(\hat{q}, x)$. By similar arguments to the proof of Lemma 11, $\hat{p}$ is a locally optimal solution of x that is reachable from p. □

Lemma 15 *There is a problem in* PLS *whose corresponding standard local optimum problem is* PSPACE-*complete.*

Proof Let M be a deterministic linear bounded automaton whose acceptance problem is PSPACE-complete; see Garey & Johnson [1979, Problem AL3]. A *configuration* of M for inputs of length n consists of the state of M, the position of the head, and a string of length n over the tape alphabet of M; the number of configurations is at most $T = d^n$ for some constant d.

Consider the following local search problem Π. The instances of Π are the same as the inputs of M. An instance x of length n has as solutions all pairs (c, t) where c is a configuration of length n and t is an integer between 1 and T. The weight (that is to be maximized) of solution (c, t) is t. If $t = T$ then the solution (c, t) does not have any neighbors, otherwise it has a unique neighbor $(c', t+1)$ where c' is the configuration resulting from c after one step of M (if c is a halting configuration, we let $c' = c$). Clearly, problem Π is in PLS.

Let c_0 be the initial configuration corresponding to the input x. Consider the initial solution $s_0 = (c_0, 1)$. The transition graph $TG_\Pi(x)$ contains a unique path starting at s_0 and ending at a locally optimal solution (sink) of the form (c_f, T), where c_f is the final configuration of the machine M. The input x is accepted by M if and only if the state of the final configuration c_f is accepting. Thus, the standard local optimum problem for Π is PSPACE-complete. □

From Lemmas 14 and 15 and the tightness of the PLS-reductions, we conclude Theorem 16.

Theorem 16 *Finding the standard local optimum for the following problems is* PSPACE-*complete: graph partitioning under the neighborhoods* (*a*) *Kernighan–Lin*, (*b*) *Swap*, (*c*) *Fiduccia–Mattheyses*, (*d*) *FM-Swap; traveling salesman problem under the k-Opt neighborhood for some constant k, and the LK′* (*Lin–Kernighan*) *neighborhood; Max-Cut, Max-2Sat and Pos NAE Max-3Sat under the Flip neighborhood; stable configuration for neural networks.*

6 PARALLEL COMPLEXITY

How much can parallelism help to speed up local search? By their nature, local search heuristics are sequential; they visit solutions one after the other. Thus, if we use a standard local search heuristic, parallelism can help only by improving the complexity of performing each iteration, i.e. determining whether a solution is locally optimal, and computing a better neighbor if it is not. This is what we called the 'local optimality' problem. On the other hand, if we do not restrict ourselves to using a local search heuristic, it is conceivable that one can find a local optimum faster by a direct parallel algorithm that avoids altogether the sequential transitioning from solution to solution. Then the question is, What is the parallel complexity of the local search problem itself? What is the parallel complexity of finding any local optimum by any algorithm?

The parallel complexity of a problem depends of course on the number of processors. Qualitatively, a minimal reasonable assumption is that there is (at most) a polynomial number of processors. If a problem takes sequentially exponential time, a polynomial number of processors cannot help in reducing the time to polynomial. Thus, the issue of parallel complexity is most interesting for problems that can be solved sequentially in polynomial time. Parallel complexity theory tries to classify problems broadly into those that are in the class NC, i.e., can be solved in polylogarithmic time ($O(\log^c n)$ for some constant c) using a polynomial number of processors, and those that are not in NC. Although we do not have a proof yet that P $\neq$ NC, this is widely believed to be the case. As in the case of P and NP, we do not know how to prove that any problem is not in NC, but instead we provide indirect evidence for this by showing that the problem is P-*complete*, i.e. complete for P under log-space reductions. If a P-complete problem is in NC, then P = NC. A prototypical P-complete problem is the circuit value problem: given a Boolean circuit C and an input vector x, compute the output $C(x)$. For more information on parallel algorithms and complexity see Karp & Ramachandran [1990].

In this section we will discuss this issue for the local search problems themselves and for their corresponding local optimality problem.

6.1 Parallel complexity of the local optimality problem

By definition, the local optimality problem for all PLS-problems can be solved in polynomial time. The first local search problems that were shown PLS-complete, Circuit/Flip and graph partitioning/Kernighan–Lin, make full use of polynomial

time in each iteration, in the sense that the local optimality problem is P-complete. In the case of Circuit/Flip, computing the cost of a solution involves evaluating a circuit for a given input, i.e. solving an instance of the circuit value problem. It follows easily that the local optimality problem for Circuit/Flip is P-complete. Using the reductions described in Section 4, the same result can be shown for graph partitioning under the Kernighan–Lin neighborhood [Johnson, Papadimitriou & Yannakakis, 1988]. In fact, Savage & Wloka [1991] showed that P-completeness of the local optimality problem holds even for unweighted graphs (i.e., with all edge weights 1).

It was thought initially that in order for a local search problem to be PLS-complete, it would be necessary to use the full power of polynomial time in each iteration. This was disproved by Krentel [1990], who showed the PLS-completeness of Max-Sat under the Flip neighborhood.

Max-Sat/Flip and many of the problems that we discussed have a very simple neighborhood function, where every solution s has a polynomial number of neighbors that can be generated very easily by changing 'few' bits in the solution, and the cost of a solution can be also easily computed in NC since it involves adding an appropriate set of weights. This is the case with the graph partitioning problem under the Swap neighborhood, the traveling salesman problem under the k-Opt neighborhood (for fixed k), Max-Cut and Max-Sat under the Flip neighborhood. Thus, the local optimality problem for these local search problems can be solved in NC in a straightforward way. In general, the parallel complexity of the local optimality problem is independent of the difficulty of the local search problem itself.

6.2 Parallel complexity of local search problems

The parallel complexity issue is of interest especially for polynomially solvable problems. For the PLS-complete problems, we do not know in general how to find local optima in polynomial time. Local search heuristics require exponential time only if the numbers (weights, distances, etc.) are large and are encoded in binary. When the numbers are small, i.e. their value is polynomially bounded in the number of nodes, cities, etc., or if the problems are not weighted, the heuristics terminate in polynomial time because every iteration decreases the cost function, and there are only polynomially many possible different costs. So we will concentrate on unweighted instances. In this case, we would like to find algorithms that work faster in parallel.

Some important paradigms of efficient parallel computation can be viewed as local search problems. One such paradigm is the *maximal independent set problem*: given a graph G, find an independent set I that is maximal, i.e., such that every node outside I is adjacent to some node in I. This problem can be viewed as a local search problem, where the solutions are the independent sets, and the neighbors of a solution J are all the independent sets (if any) that consist of J and one more node. Sequentially, it is easy to find a maximal independent set by a straightforward (local search) algorithm: starting from the empty set, look for

a node that is independent from the current set, add it to the set, and iterate. It is not easy to parallelize this algorithm. In fact, finding the set that is computed by this algorithm if we process the nodes in a given order is a P-complete problem (this is known as the greedy or the lexicographically first maximal independent set problem); see Karp & Ramachandran [1990]. However, if we do not care to compute a specific set, it is possible to compute *some* maximal independent set in NC using much more sophisticated parallel methods [Karp & Wigderson, 1985; Luby, 1986].

Luby [1986] showed that the maximal independent set problem can be formulated as a special case of the stable configuration problem by choosing appropriate (small) weights, and he raised the question of the parallel complexity of the latter problem. There is another special case of the stable configuration problem that corresponds to another paradigmatic problem that can be solved efficiently in parallel. If all the edge weights are positive, the stable configuration problem is equivalent to finding a local optimum for a Min-Cut or $s-t$ Min-Cut problem (depending on whether there are node thresholds) in an undirected graph, under the Flip neighborhood. In the case of general weights, the Min-Cut problem itself (i.e., finding a global optimum) can be solved in (deterministic) NC [Karger & Motwani, 1994], whereas the status of $s-t$ Min-Cut is open (for undirected graphs; for directed graphs both optimization problems are P-complete [Goldschlager, Shaw & Staples, 1982; Karger, 1993]). In the case of polynomially bounded weights, the $s-t$ Min-Cut problem is in Random NC [Karp, Upfal & Wigderson, 1986].

The special case of the stable configuration problem with all edge weights -1 and no thresholds is equivalent to finding a local optimum for Max-Cut in an unweighted graph under the Flip neighborhood. This problem is also known as the *different-than-majority labeling problem*: we wish to assign one of two labels to each node of a graph so that the majority of its neighbors have different label. Although this sounds like a rather simple problem, it turns out in fact to be P-complete [Schäffer & Yannakakis, 1991]. An interpretation of the result is that unweighted Hopfield neural nets form a universal model of computation for polynomial time: For any polynomial time computable function f and instance x, we can choose the edges of the net (with all the edge weights being -1), so that no matter which configuration the net starts from, it will converge to a configuration that encodes $f(x)$. The proof is quite complicated and involves a delicate reduction from the circuit value problem.

The P-completeness of finding a local optimum can be transferred from unweighted Max-Cut/Flip to other unweighted problems using log-space reductions that are the same or similar to the ones we described in Section 4. We summarize these results in the next theorem [Schäffer & Yannakakis, 1991].

Theorem 17 *Finding a local optimum for the unweighted versions of the following problems is* P*-complete: graph partitioning under the neighborhoods* (*a*) *Kernighan–Lin*, (*b*) *Swap*, (*c*) *Fiduccia–Mattheyses*, (*d*) *FM-Swap*; *Max-Cut*,

Max-2Sat and Pos NAE Max-3Sat under the Flip neighborhood; stable configuration for neural networks.

It follows that the standard local optimum problem – given an initial solution find the specific local optimum reached by the standard heuristic – for the unweighted versions of these problems is also P-complete. For graph partitioning this was shown independently by Savage & Wloka [1991]. We do not know if similar P-completeness results hold for the TSP with small weights under the k-Opt or the Lin–Kernighan neighborhood.

7 LOCAL VERSUS GLOBAL OPTIMALITY

Local search is usually applied to tackle hard optimization problems. The neighborhood that is imposed must trade off complexity with the quality of the local optima. The neighborhood must be tractable enough to be possible to search it efficiently, but at the same time we would like to find good solutions. Ideally we would like to have an exact neighborhood, one in which local optimality and global optimality coincide. It is intuitively clear that the complexity of the optimization problem has a bearing on the quality of the local optimal solutions that we are guaranteed to find. This can be formalized as follows.

Recall that a problem is called *strongly* NP-hard if it is NP-hard even if the weights (costs) are polynomially bounded. In terms of approximation, we measure the quality of solutions by the ratio of their cost to the optimal cost. We say that a neighborhood structure guarantees ratio r, if for any instance x and locally optimal solution $\hat{s}$, the cost of $\hat{s}$ is within a factor r of the optimal cost. We have the following theorem.

Theorem 18 *Let Π be an optimization problem and $\mathcal{N}$ a neighborhood function such that the local search problem* $(\Pi, \mathcal{N})$ *is in* PLS.

1. *If Π is strongly* NP-*hard (respectively,* NP-*hard), then $\mathcal{N}$ cannot be exact unless* P = NP (*resp.* NP = *co*-NP).
2. *If the approximation of Π within a factor r is strongly* NP-*hard (resp.,* NP-*hard), then $\mathcal{N}$ does not guarantee ratio r unless* P = NP (*resp.* NP = *co*-NP).

Proof Suppose that Π is strongly NP-hard, and consider an instance with polynomially bounded weights. Then the standard local search algorithm will converge in polynomial time, and if the neighborhood is exact, the computed locally optimal solution will be also globally optimal.

In general suppose that Π is an NP-hard (possibly, not strongly) optimization problem, say a minimization problem for concreteness. Formally, the following decision problem is NP-hard: given an instance x and a value v, does there exists a solution with cost at most v? If $\mathcal{N}$ is an exact neighborhood, we can solve the complementary decision problem in nondeterministic polynomial time as follows: given an x and v, guess a solution $\hat{s}$, verify that $\hat{s}$ is locally (and hence globally) optimal, and that its cost satisfies $f(\hat{s}) > v$.

The analogous statements about the approximation ratio follow by the same arguments. □

Papadimitriou & Steiglitz [1977] showed first results along these lines for the traveling salesman problem. They proved that assuming P $\neq$ NP, if we can search the neighborhood in polynomial time, it cannot be exact; furthermore, in the absence of triangle inequality, there must be local optima that are arbitrarily worse than the global optimum.

The conjecture NP $\neq$ co-NP is stronger than P $\neq$ NP, but it is widely believed also to be true. Moreover, we remark that even for problems that are only weakly NP-hard, it is often the case that the following decision problem is also NP-hard: given a solution, is it (globally) optimal? In that case it follows easily that a polynomially searchable neighborhood N cannot be exact.

The optimization problems we have discussed in this chapter, TSP, graph partitioning, Max-Cut, maximum satisfiability, are strongly NP-hard. In terms of approximability, TSP without triangle inequality cannot be approximated within any constant factor [Sahni & Gonzalez, 1976]. In the presence of triangle inequality, there is a 3/2 approximation [Christofides, 1976]. However, even in the case when all the distances are 1 and 2 (which satisfy the triangle inequality) the TSP is MAX SNP-hard, and hence there is an $r > 1$ such that approximation with ratio r is (strongly) NP-hard [Papadimitriou & Yannakakis, 1993; Arora et al., 1992].

The approximability status of graph partitioning with a minimization objective is open (for the purposes of approximability it makes a difference whether we are considering the minimization or maximization version). Max-Cut and maximum satisfiability can be approximated within a constant factor. In fact, for both problems, every locally optimal solution under the Flip neighborhood guarantees a ratio 2. There are other approximation algorithms that run in polynomial time even for weighted instances and achieve better ratios [Goemans & Williamson, 1994; Johnson, 1974; Yannakakis, 1994], but again both problems are MAX SNP-hard (even in the unweighted case) and cannot be approximated within some constant factor r unless P = NP [Papadimitriou & Yannakakis 1991; Arora et al., 1992].

Motivated by examples such as Max-Cut and maximum satisfiability, some researchers have started looking at what types of optimization problems can be approximated within a guaranteed constant factor using (polynomial) local search. Of course, since PLS contains P, if no condition is imposed on the neighborhood (other than that it be searchable in polynomial time), this is possible for all the problems that can be approximated within a constant factor by any polynomial-time approximation algorithms. Ausiello & Protasi [1995] examine this question under the restriction that the neighborhood is definable by bounded Hamming distance (i.e., as k-Opt) for a suitable encoding of solutions. Khanna et al. [1994] use standard encodings but allow the local search to use a different cost function than the optimization problem (they call it 'non-oblivious' local search). Halldorsson [1995] derives improved

approximation ratios for various packing and covering problems using local search.

For specific problems and restricted types of neighborhoods it is possible to show sharper results than Theorem 18 without complexity theoretic assumptions. Weiner, Savage & Bagchi [1976] and Vizing [1978] showed that, in the traveling salesman problem, neighborhoods that are exact must be exponentially large, assuming that they are data-independent, i.e., do not depend on the distances; this fact is true even if one is concerned only with instances where all distances are 1 or 2 (and thus satisfy the triangle inequality). Papadimitriou & Steiglitz [1978] have constructed examples, where there is only one optimal tour, but there are exponentially many second-best tours that (i) cannot be improved by changing less than 3/8 of the edges, and (ii) have very large cost, arbitrarily worse than the optimal (in the absence of triangle inequality). Thus, for k-Opt with fixed k (actually, for $k < 3n/8$), all these exponentially many tours are local optima of bad quality. A result with a similar flavor for the maximum independent set problem was shown by Rödl & Tovey [1987]. They construct a graph G which has the property that for any neighborhood structure of polynomial size, there is a renaming of the graph G that has exponentially many independent sets that are locally optimal solutions, but not globally optimal.

There has been some further analysis of the k-Opt heuristics for the metric TSP more recently by Chandra, Karloff & Tovey [1997], who showed a number of negative and positive results; earlier Plesnik [1986] had shown a negative result for 2-Opt. The approximation ratio of 2-Opt for the metric TSP is $\Theta(\sqrt{n})$, and even general k-Opt cannot guarantee a ratio better than $\Omega(n^{1/2k})$. If the cities are points in the Euclidean space $\mathbb{R}^d$, the ratio of k-Opt improves to $O(\log n)$. For points on the plane $\mathbb{R}^2$, 2-Opt has ratio at least $\Omega(\log n/\log\log n)$.

Empirically, local search heuristics seem to produce very good approximate solutions. For example, TSP heuristics come within a few percentage points of the optimum for 'typical' instances on the plane, and outperform other algorithms with better worst-case approximation properties (e.g., Christofides' algorithm); see Chapter 8. There are few analytical results supporting this behavior. One such result shown by Chandra, Karloff & Tovey [1997] is that for points sampled i.i.d. uniformly from the unit hypercube in $\mathbb{R}^d$, the worst-case ratio of 2-Opt is bounded by a constant with high probability, and in expectation.

8 CONCLUSIONS

We reviewed in this chapter some of the theoretical work that has been done on the complexity of local search problems. We defned the class PLS, and saw that many local search problems are PLS-complete. Thus, although we do not know whether any one of them can be solved in (worst-case) polynomial time, we know that either all these problems can be solved in polynomial time or none of them can. And this theory helped us to establish the compexity of the standard local search heuristics for these problems (namely, exponential time). PLS-complete-

ness seems to be a widespread phenomenon that applies to many local search problems, although the reductions tend to be considerably more delicate and involved than NP-completeness reductions.

Many theoretical problems remain open in the field of local search. The most important and difficult question is the relationship of PLS to P and NP. Besides this, there are many interesting local search problems that we do not know to be in P or to be PLS-complete. For example, TSP under the 2-Opt neighborhood, and the subdeterminant problem mentioned in Section 2. Another interesting problem concerns the computation of optimal policies for two-person simple stochastic games; see Condon [1989] for the definition. Finally, there has been very little work on analyzing the average performance of local search heuristics, both in terms of complexity and quality of approximation.

Local Search in Combinatorial Optimization
Edited by E. Aarts and J. K. Lenstra
© 1997 John Wiley & Sons Ltd.

3

Local improvement on discrete structures

Craig A. Tovey
Georgia Institute of Technology, Atlanta

1 INTRODUCTION 57
1.1 Preliminaries 58
2 WORST-CASE ANALYSIS 59
2.1 Beating the standard algorithm 59
2.2 Separator theorems and upper bounds on the cost of finding a local minimum 61
2.3 Lower bounds on finding a local optimum 63
2.4 Worst-case cost of local search on the hypercube and typical neighborhood graphs 66
2.5 A best possible randomized algorithm on the hypercube 67
2.6 Summary 70
3 THE STANDARD ALGORITHM: ANALYSIS AND VARIATIONS 70
3.1 Basic properties of the standard algorithm on the hypercube 70
3.2 Structured functions on the hypercube 72
3.3 Variations on the standard algorithm 75
3.4 Summary 79
4 AVERAGE-CASE ANALYSIS 79
4.1 The uniform boundary distribution on unimin functions 80
4.2 The uniform edge-boundary distribution 83
4.3 Other distributions and computational results 84
4.4 Summary 86
5 MULTIPLICITY AND QUALITY OF LOCAL OPTIMA 86
ACKNOWLEDGMENTS 89

1 INTRODUCTION

Anyone who uses local search must make several crucial design decisions, no matter what specific problem is to be solved. What neighborhood should be selected? What search algorithm should be used? If the standard algorithm is used, what selection rule should be employed?

Because these are general questions, we try to investigate them in a general way. Let $(\mathcal{S}, f)$ be an instance of a combinatorial optimization problem, where the cost function f is a mapping from the set of solutions $\mathcal{S}$ to $\mathbb{Z}$. To use local search we must select a neighborhood function $\mathcal{N}: \mathcal{S} \to 2^{\mathcal{S}}$. A local optimum is an

$\hat{i} \in \mathscr{S}$ whose cost is no worse than its neighbors: $f(\hat{i}) \leqslant f(j) \forall j \in \mathscr{N}(\hat{i})$. We will measure the amount of work required to find a local optimum by the number of points at which f must be evaluated.

This is an abstract way of thinking about local search. It is a medium for developing and evaluating *general* approaches to local search, divorced from the details of specific problems. Another reason to study local search in a general setting is that the standard algorithm behaves much the same across a wide range of problems. For decades it has been observed that the standard algorithm (see Chapter 2, Section 2) performs well in practice and badly in the worst case, on many specific optimization problems. This consistent pattern of performance seems to call for a general unifying explanation.

The main result in this chapter is that for almost all choices of neighborhood functions $\mathscr{N}$, the standard algorithm has bad worst-case but good average-case speed. This analysis confirms empirical observations and deepens our understanding of local search.

As with any abstraction, this general approach emphasizes certain characteristics and downplays others. What are its primary features? In these studies, the form of the cost function f is usually arbitrary. Even if it is constrained to possess a single local minimum, or in some other way, f is still not as structured as in an actual combinatorial optimization problem. In addition, this kind of analysis tends to ignore the actual values of f, and depends mainly on *ordinal* information instead. The graphical point of view also emphasizes the neighborhood structure.

This would suggest that our approach should be useful to analyze the standard algorithm. And indeed that is where the analysis generates the most insight. The analysis also leads to some promising variants of the standard algorithm, and indirectly suggests we should design neighborhoods that induce maximum correlation between objective function values of adjacent points.

1.1 Preliminaries

A graph $G = (V, E)$ represents the combinatorial optimization problem $(\mathscr{S}, f)$ together with the neighborhood function $\mathscr{N}: \mathscr{S} \to 2^{\mathscr{S}}$. G has vertex set $V = \mathscr{S}$. If $i \in \mathscr{N}(j)$ then $\{j, i\} \in E$. We will usually assume the neighborhood relation is symmetric, $i \in \mathscr{N}(j) \Leftrightarrow j \in \mathscr{N}(i)$, so G is an undirected graph. A local minimum $\hat{i} \in V$ corresponds to a local minimum of f on $\mathscr{S}$: $f(\hat{i}) \leqslant f(j) \forall \{\hat{i}, j\} \in E$.

The typical graphs we are interested in have low degree compared with $|V|$. This is because the size of a neighborhood is typically much smaller than the size of the search space. Also the diameter of the graphs is small, for otherwise the standard algorithm would have to fare poorly. (The diameter of a graph is the greatest distance between any pair of vertices, where distance is the minimum length connecting path; a disconnected graph has infinite diameter.)

The prototypical graph is Q^d, the d-dimensional cube, which has logarithmic (in $|V|$) degree and diameter. The maximum size of a neighborhood $|\mathscr{N}(i)|$ is the maximum degree $d_{\max}$ of the associated graph. We generally assume G is d-regular. Though d appears to be a constant, it is usually a nondecreasing

function polylogarithmically bounded in $|V|$, i.e., $d = O((\log|V|)^k)$ for some constant k.

2 WORST-CASE ANALYSIS

The generally accepted measure of work in finding a local minimum of a function f on graph G is the number of function evaluations required. We are charged one unit for each call to an oracle which, given $v \in V$, evaluates $f(v)$. Under this oracle model of computation, what is the worst-case cost of finding a local optimum on G? In this section we present a general theory that provides fairly tight bounds for any graph. We apply the general theory to our prototypical graph, the hypercube, and to symmetric d-regular neighborhood relations. Finally, we present a worst-case analysis of randomized local search algorithms on the hypercube.

2.1 Beating the standard algorithm

Our aim is to determine $v(G)$, the minimum over all algorithms A of the worst-case number of queries for A to find a local minimum of f in G. This is well defined because the number of possible algorithms A is finite for analytical purposes, as the ordinal values of f are all that matter.

Although the standard algorithm will always find a local minimum in a graph, its worst-case performance can be far from optimal. The simplest example is where G is a simple path of length n. The standard algorithm requires $1 + \lceil \frac{n}{2} \rceil$, but it is not hard to see that Fibonacci search has the optimal worst-case cost of $\log_\phi n + O(1)$ queries, where ϕ is the golden mean reciprocal $(\sqrt{5} + 1)/2$.

For a more revealing example, let G be an $m \times m$ grid graph, and let f describe a descending spiral, as shown for $m = 8$ in Figure 3.1 (f values of vertices of G are given in the corresponding matrix cell). M is any large value such as m^2. The standard algorithm must spiral outwards to the unique local minimum in the corner, requiring $\Omega(m^2)$ function evaluations.

On the other hand, the following divide-and-conquer strategy will require only $3m + O(\log m)$ evaluations: query the middle (e.g., fifth) column of the matrix. Let v be the vertex in that column with smallest f value ($f(v) = 5$) in this case). If v is

1	2	3	4	5	6	7	8
M	M	M	M	M	M	M	9
27	28	29	30	31	32	M	10
26	M	M	M	M	33	M	11
25	M	39	M	M	34	M	12
24	M	38	37	36	35	M	13
23	M	M	M	M	M	M	14
22	21	20	19	18	17	16	15

Figure 3.1 Fooling the standard algorithm on a grid

a local optimum, stop. If not, imagine running the standard algorithm from v. Since the standard algorithm never lets f increase, and it starts at the least function value in the fifth column, the path it follows will never cross the column. Therefore, there is a local optimum in the left half of the matrix. Discard everything to the right of the fifth column, and proceed by querying the middle (fifth) row. Let w be the vertex in that row with smallest f value ($f(w) = 25$ in the example). If $f(w) \geqslant f(v)$ then the standard algorithm, starting from v, will never cross the fifth row, and we can discard everything below the row (this is what occurs in the example). If $f(w) < f(v)$ then local search from w would never cross either the fifth row or fifth column and again we may discard half the remaining graph. Iterating this procedure on the remaining $m/2 \times m/2$ portion, we find a local optimum in $3m + O(\log m)$ queries.

In fact, a local optimum of the $m \times m$ grid can be found in $cm + O(\log m)$ queries where $c < 3$. The idea is to make the first set of queries along a diagonal. The best known value for c is about 2.554 [Llewellyn & Tovey, 1993].

The strategy above easily generalizes to an algorithm to find a local optimum in any graph.

Definition 1 *A graph $G = (V, E)$ is* separated *by a subset of vertices $S \subseteq V$ if and only if $S = V$ or there exist vertices u and $v \in V$ such that for all paths $P = (u, \ldots, v)$ connecting u and v in G, $\{P \setminus \{u, v\}\} \cap S \neq \varnothing$.*

Algorithm 1 Divide and conquer algorithm to find a local minimum *The algorithm is given a graph $G = (V, E)$ and may call an oracle to evaluate function $f: V \to \mathbb{R}$. It returns a local minimum of f on G.*

1. *Set $i = 0$, set $G_0 = G$, and define $f(u_0) = \infty$.*
2. *Select vertices one at a time for f evaluation until the set S_i of selected vertices separates G_i.*
3. *Find $v' \in S_i$ that minimizes $f(v)$ for all $v \in S$. Break ties arbitrarily.*
4. *Evaluate $f(w)$: $w \in N(v')$, the neighbors of v'.*
5. *If $f(v') \leqslant f(w)$ for all $w \in N(v')$, return $v^* = v'$ and stop.*
6. *Let $w \in N(v') \setminus S$ be such that $f(w) < f(v')$. If $f(w) \leqslant f(u_i)$, set $u_{i+1} = w$, otherwise set $u_{i+1} = u_i$. Let G_{i+1} be the connected component of $G_i \setminus S_i$ containing u_{i+1} and let $i = i + 1$.*
7. *Go to Step 2.*

Theorem 2 *Algorithm 1 finds a local minimum of f in G.*

Proof By induction in the ith iteration, let $S = \bigcup_{j=0}^{i} S_j$. If the standard algorithm were employed starting with u_{i+1} (chosen in Step 5), it could never select a vertex in S because $f(u_{i+1}) \leqslant f(w) < f(v') \leqslant f(v) \forall v \in S_i$ and $f(u_{i+1}) \leqslant f(u_i) \leqslant f(v) \forall v \in S \setminus S_i$ (by induction). Hence the search could never leave G_i and so would find a local minimum there. □

Observe how we make hypothetical use of the standard algorithm to prove that Algorithm 1 works! The key to the divide-and-conquer strategy is to query a set of vertices that separates the graph into connected components. Cleverly choos-

ing the separating sets can speed up the algorithm. We can imagine the following game, whose solution is the best choice of separating sets to optimize worst-case performance. Player I, the minimizer, removes vertices from graph G until G is disconnected. Play then passes to player II, the maximizer, who chooses one of the newly created components to become the new G. The other components are discarded and play returns to player I. The game ends when G has a single vertex, with value equal to the total number of vertices removed by player I.

Separation game

Input. Graph $G = (V, E)$.

1. Set $i = 0$; $G^0 = G$; $V^0 = V$; $score = 0$.
2. If $|V^i| \leqslant 1$ stop.
3. Player I chooses $S^i \subseteq V^i$ to delete so that $G^i \backslash S^i$ is empty or not connected. Set $score = score + |V^i|$.
4. Player II selects G^{i+1}, a connected component of $G^i \backslash S^i$. Set $i = i + 1$.
5. Go to Step 2.

The value of the separation game is the final value of score when each player plays optimally, player I to minimize and player II to maximize.

For any graph G, let $s(G)$ denote the value of the separation game on G, and let $i(G)$ denote the number of iterations (turns taken by player I) when $s(G)$ is attained. From our divide-and-conquer algorithm, we have an upper bound on the cost of local search.

Lemma 3 *The divide-and-conquer algorithm finds a local optimum in G with at most $s(G) + i(G)d_{\max}$ queries, where $d_{\max}$ is the maximum degree of any vertex in G.*

The values of both $i(G)$ and $d_{\max}$ are usually logarithmically (in $|V|$) small. Therefore $s(G)$ is usually within a small additive amount, an $O(\log^2 |V|)$ term, of being an upper bound on $v(G)$, the optimal worst-case cost.

2.2 Separator theorems and upper bounds on the cost of finding a local minimum

How do we go about using Algorithm 1 on actual graphs? If our divide-and-conquer method is to be efficient, we must separate the graph by a small number of vertices. In terms of the separation game, player I wants to find small separating sets S^i. In this subsection we discuss how good separating sets may be found for certain graph classes.

A *separator theorem* guarantees that one can find a 'small enough' number of vertices whose deletion separates the graph into components that are not 'too large'. If a separator theorem applies to all subgraphs of a given graph, then player I of the separation game can use it repeatedly to shrink the size of the current graph, without increasing the score too much. Hence separator theorems give upper bounds on $s(G)$.

Definition 2 *A class of graphs $\mathcal{G}$ is $z(n)$-separable with constants $\frac{1}{2} < \alpha < 1$ and $\beta > 0$ if the n vertices of any $G \in \mathcal{G}$ can be separated by a set of less than $\beta z(n)$ vertices into two sets A and B with $|A| \leqslant \alpha n$ and $|B| \leqslant \alpha n$.*

Theorem 4 [Llewellyn, Tovey & Trick, 1989] *For any planar graph on n vertices, the value of the separation game is at most* $13.35\sqrt{n}$.

Corollary 5 *A local optimum on a planar graph with n vertices and maximum degree d can be found in*

$$13.35\sqrt{n} + d\left(\frac{\log n}{\log\ 3 - 1}\right)$$

function evaluations.

Proof Player I follows the strategy outlined above, using the following result of Djidjev [1982]: the class of planar graphs is $\sqrt{n}$-separable with constants $\alpha = 2/3$ and $\beta = \sqrt{6}$. The corollary follows from Lemma 3. □

Similarly, for graphs of fixed genus a separator theorem of Gilbert, Hutchinson & Tarjan [1984] leads to upper bounds on the time to find a local optimum.

Theorem 6 [Llewellyn, Tovey & Trick, 1989] *For any graph of genus g on n vertices, the value of the separation game is at most* $(6\sqrt{g} + 2\sqrt{2})(3 + \sqrt{6})\sqrt{n} + O(\log n)$.

Corollary 7 *A local optimum on graph of genus g with n vertices and maximum degree d can be found in*

$$(6\sqrt{g} + 2\sqrt{2})(3 + \sqrt{6})\sqrt{n} + O(\log n) + d\left(\frac{\log n}{\log 3 - 1}\right)$$

function evaluations.

The genus-based bounds are not always very tight. For example, our prototypical neighborhood graph Q^d has genus $(d-4)2^{d-2} + 1$ [Beineke & Harary, 1965; Ringel, 1965], from which Corollary 7 guarantees that a local optimum can be found with approximately $10\sqrt{d}\,2^d$ function evaluations, considerably more than the number of vertices in the graph! By direct application of Algorithm 1, we can reduce the number of queries needed to $\sim \log d/\sqrt{d}$ of the number of vertices.

Proposition 8 [Llewellyn, Tovey & Trick, 1989] *Algorithm 1 can find a local minimum on Q^d in at most* $(\sqrt{2/\pi} + o(d))(\log d)2^d/\sqrt{d}$ *function evaluations.*

Proof Define the mth *layer* of Q^d to consist of those vertices with exactly m 1's, so the mth layer contains $\binom{d}{m}$ vertices. Each layer $(0 < m < d)$ separates Q^d. Player I selects the middle layer of whatever portion of Q^d remains at each turn, in effect performing binary search to shave the graph down to a single layer. Then at most $\lfloor \log d \rfloor + 1$ layers, each of size at most $\binom{d}{\lfloor d/2 \rfloor}$, are queried. The results follows from the extended Stirling approximation. □

2.3 Lower bounds on finding a local optimum

Lemma 3 of the previous section shows that adding a small quantity to $s(G)$, the value of the separation game, yields an upper bound on $v(G)$, the optimal cost of finding a local optimum on G. Next we show that *the same quantity*, $s(G)$, *is itself a lower bound on* $v(G)$. Therefore $s(G)$ provides a very close estimate of the optimal worst-case cost $v(G)$.

Theorem 9 *Any algorithm to find a local minimum in G requires at least* $s(G)$ *queries; the divide-and-conquer algorithm requires at most* $s(G) + i(G)d_{\max}$ *queries.*

Proof The second statement is Lemma 3. For the lower bound, we play the adversary against an arbitrary algorithm. The key idea is that as long as the set of queries Q does not separate G, the adversarial oracle can respond $f(v) = k \forall v \in Q$, and the local optimum could still be anywhere else in the graph. This is because there exist paths of unqueried vertices from every $v \in Q$ to every vertex in $V \backslash Q$. These paths could be decreasing in f value, so the local minimum could be hiding anywhere in $V \backslash Q$.

The adversarial strategy is illustrated in Figure 3.2. The first vertices queried are k,l,b,x,w,i, and o. The adversary returns function value 100 (arbitrarily chosen) for each of them. The next query, v, disconnects the graph. Suppose the adversary decides to hide the local optimum in the left half of the graph. In this case, it fixes values in the right half of the graph (see Figure 3.2) giving decreasing values towards vertex v, so there are no local optima in that portion of G. For any future queries in that part of G, the fixed values will be returned. For future queries in the left half of the graph, the adversary will return value 94, until that portion is itself disconnected.

More precisely, define a *bare tree* of G to be an acyclic connected subgraph T of G such that all vertices of G are in or adjacent to T. Removing the leaves from a spanning tree results in a bare tree, hence the name.

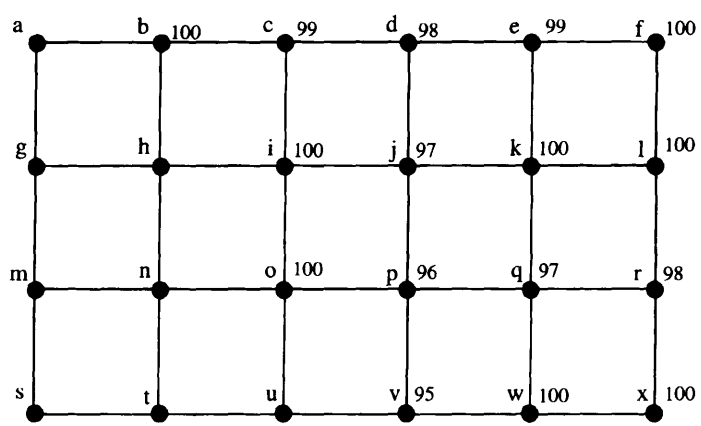

Figure 3.2 The adversarial strategy

The oracle's adversarial strategy

1. Set $i = 0$, $G_0 = G$, and set w_0 to any initial integer value.
2. For every vertex query, mark the vertex and respond with the value w_i until G_i contains no bare tree of unmarked vertices. For the vertex v^i that eliminates the last bare tree of unmarked vertices, do not respond with value w_i, and instead proceed to Step 3.
3. Choose a bare tree T of G_i, using only unmarked vertices. By Step 2, T must contain v^i, which is still unmarked. For all unmarked vertices u in G_i not in T, fix $f(u) = w_i$. Let D equal the length of a longest path in T emanating from v_i. Fix $f(v^i) = w_i - D - 1$ and mark v^i.
4. Removing v^i disconnects T into $T_1, \ldots, T_k$ where T_r is a bare tree of the connected component C_{ir} induced by the unmarked vertices of G_i. Select some q, $1 \leqslant q \leqslant k$ (corresponding to the component player II selects in Step 4 of the separation game). For all $r \neq q$, for all $w \in T_r$, fix $f(u) = f(v^i) + d(v^i, u)$, where $d(v^i, u)$ is the length of the path from v^i to u in the tree T.
5. Set $i = i + 1$, $G_i = C_{i-1,r}$, $w = f(v^i) - 1$.
6. If G_i contains more than one vertex, proceed to Step 2.

As long as the graph is connected, the oracle can devise a network of deepening paths towards any unqueried vertex. Therefore, as long as G has more than one (unqueried) vertex, it cannot be determined where G has a local minimum. We remark for future use that the oracle actually creates a unimin function, i.e., f possesses a unique local minimum on G. □

Theorem 9 tells us that the value of the separation game, $s(G)$, is an extremely close estimate of the worst-case number of queries to find a local optimum in the graph G. Thus the problem of worst-case analysis reduces to finding $s(G)$. Unfortunately, solving the separation game turns out to be NP-hard.

Theorem 10 [Llewellyn, Tovey & Trick, 1989] *For graph G and integer k, determining whether the separation game value $s(G) \leqslant k$, is* NP-*complete.*

Proof sketch Apply the reduction of Garey & Johnson [1979], taking 3-Sat to Vertex Cover, on the restricted form of Sat from Tovey [1984], in which each variable appears in at most three clauses, appears complemented exactly once, and each clause has two or three literals. The resulting graphs can be explicitly 3-colored. Hence Vertex Cover is NP-complete when restricted to explicitly 3-colored graphs. In the reduction to the separation game, we create a graph made up of three heavily connected components, so that player I's optimal strategy must be first to separate these three components. In the graph, each component corresponds to a color, and the links between components are the links between the color classes of the Vertex Cover instance. Player I therefore must solve the Vertex Cover instance to optimally separate the three components. □

In view of Theorem 10, we seek estimates of $s(G)$. Section 2.2 has already shown how to get upper bounds from separator theorems. The next results show how to get lower bounds on $s(G)$ from isoperimetric boundary theorems.

Definition 3 *In graph $G = (V, E)$, for any subset of vertices $S \subseteq V$ let the* boundary *of S, denoted $\partial(S)$, be*

$$\{v \in V \setminus S : \exists s \in S \text{ s.t. } \{s, v\} \in E\}.$$

Let the edge-boundary *of S, denoted $\partial^e(S)$, be*

$$\{\{s, v\} \in E : s \in S,\ v \notin S\}.$$

Theorem 11 *For any graph $G = (V, E)$ the value of the separation game is bounded by*

$$s(G) \geqslant \min_{(|V|/3) \leqslant |S| \leqslant (2|V|/3)} |\partial(S)|.$$

Proof Each time player I separates the graph, player II selects the largest remaining connected component until there is none with cardinality greater than $2|V|/3$. Either one of the connected components contains between $|V|/3$ and $2|V|/3$ vertices, or all of them contain fewer than $|V|/3$. In either case, there must be a subset (a singleton in the first case) of components whose vertices, taken as a set S, are disconnected from the rest of G. Moreover S has cardinality $|V|/3 \leqslant |S| \leqslant 2|V|/3$. Since S is separated from the rest of G, its entire boundary $\partial(S)$ must be contained in the set of vertices deleted in Step 2 of the separation game. □

Theorem 12, a similar bound given by Althöfer & Koschnick [1993], is appealing in its simplicity.

Theorem 12 *For any connected graph $G = (V, E)$ the value of the separation game is bounded by the cardinality of the smallest subset $C \subset V$ such that the largest connected component of $G - C$ has at most $\frac{1}{2}(|V| - 1)$ vertices.*

Proof This follows from the same adversarial strategy used in the proof of Theorem 9. □

Theorem 13, given by Llewellyn, Tovey & Trick [1989], is a stronger bound but also more cumbersome.

Theorem 13 *For any graph and integer t,*

$$s(G) \geqslant \min\left\{t, \max_k \min_{S: k-t \leqslant |S| \leqslant k} |\partial(S)|\right\}.$$

In particular, where t is set to

$$\beta(G) = \max_i \min_{|S| = i} |\partial(S)|,$$

we have, for any graph G,

$$s(G) \geqslant \max_k \min_{k - \beta(G) \leqslant |S| \leqslant k} |\partial(S)|.$$

The proof is similar to that of Theorem 11.

2.4 Worst-case cost of local search on the hypercube and typical neighborhood graphs

We now apply the general theory developed in the previous section to the hypercube, our prototypical neighborhood graph, and to the family of random typical neighborhood graphs. For the hypercube graph Q^d we have the following result.

Theorem 14 [Llewellyn, Tovey & Trick, 1989] *Any algorithm that finds a local minimum on the d-dimensional hypercube Q^d requires at least $(\sqrt{2/\pi} - o(d))2^d/\sqrt{d}$ function evaluations.*

Proof By Theorem 9 we seek a lower bound on $s(Q^d)$. We will use an inequality resulting from Harper's [1966] isoperimetric theorem for Q^d [Tovey, 1986], which we state as Lemma 15.

Lemma 15 *Let S be any subset of the vertices of Q^d. If*

$$\sum_{j=0}^{i-1} \binom{d}{j} \leqslant |S| < \sum_{j=0}^{i} \binom{d}{j}$$

then

$$|\partial(S)| \geqslant \begin{cases} \binom{d}{i} & \text{if } i \leqslant \frac{1}{2}(d-1), \\ \binom{d}{i+1} & \text{if } i \geqslant \frac{1}{2}(d-1). \end{cases}$$

In addition, $\beta(Q^d) = \binom{\lfloor d}{d/2 \rfloor}$ where $\beta(G)$ is as defined in Theorem 13.

The bound on $s(Q^d)$ now follows by estimating terms with the extended Stirling approximation and applying Theorem 13; see Llewellyn, Tovey & Trick [1989, pp. 173–175] for details. □

Theorem 14 says that any algorithm to find a local minimum in Q^d must examine at least $\sim 1/\sqrt{d}$ of its vertices in the worst case. Is the hypercube an exceptional graph in this regard, or do other d-regular graphs require a similar number of function queries? In other words, how typical is our prototypical graph? Our next result shows that most d-regular neighborhoods have a similar, and in fact slightly worse lower bound.

We must clarify what we mean by 'most neighborhoods'.

Definition 4 *Let $\mathcal{G}_{n,d}$ denote the class of d-regular graphs on n vertices. In $\mathcal{G}$ is a sequence of classes from $\mathcal{G}_{n,d}$, one class for each integer n, then we say* almost every *graph in $\mathcal{G}$ has property Q, when the probability that a random graph from $\mathcal{G}$ on n vertices has property Q converges to 1 as $n \to \infty$.*

For models of random d-regular graphs see Bollobás [1985, II.4] and Friedman, Kahn & Szemerédi [1989]. Bollobás [1985, X.3] proved with de la Vega that for all $d \geqslant 3$ almost all d-regular graphs on n vertices have diameter $O(\log n/\log d)$. By the equivalence between symmetric d-regular neighborhood functions and their graphs, the same definition may be used to make statements about 'most symmetric d-regular neighborhood functions'.

Theorem 16 *For almost all symmetric d-regular neighborhood functions on $\mathscr{S}$, any algorithm to find a local minimum of f must evaluate at least 1/9 of the points of $\mathscr{S}$, in the worst case.*

Proof Let $\mathscr{G}$ be a sequence of classes of the neighborhood graphs on n vertices, regular of degree $d(n) \geqslant 3$. Then the statement of the theorem is equivalent to saying that almost every graph in $\mathscr{G}$ has

$$s(G) \geqslant \frac{(d - O(\sqrt{d})n}{8d} >_{n \to \infty} n/9.$$

Hence almost every graph in $\mathscr{G}$ must have at least 1/9 of its vertices queried, in the worst case, to find a local optimum.

Just as in Theorem 14, the key is to show that the boundaries of large subsets of vertices must be large. This was provided by Lemma 15 for Q^d. In the random graph case we utilize the theory of eigenvalues and graph expanders.

Definition 5 *A graph $G = (V, E)$ is an (n, d, c)-expander if G is a d-regular graph on n vertices such that $\forall W \subset V: |W| \leqslant n/2$ the boundary of W is cardinality at least $|\partial(W)| \geqslant c|W|$.*

To illustrate, Lemma 15 implies that Q^d is a $(2^d, d, 1/d)$-expander.

Lemma 17 *If G is an (n, d, c)-expander than $s(G) \geqslant cn/4$.*

Proof Choose $k = n/2$ and $t = n/4$ in Theorem 13. □

Let Λ_2 denote the second largest eigenvalue of a d-regular graph G on n vertices. Alon, Spencer & Erdös [1992, Corollary 9.2.2] prove that G is an (n, d, c)-expander with $c = (d - \Lambda_2)/2d$.

To connect this to $\mathscr{G}$, note it is sufficient to consider the sparsest graph classes where d is fixed. Friedman, Kahn & Szemerédi [1989, Theorem A] prove that, for fixed d and almost all d-regular graphs, the second largest eigenvalue, $\Lambda_2 = O(\sqrt{d})$. Combining this with the preceding result of Alon and Milman, almost all the graphs in $\mathscr{G}$ are (n, d, c)-expanders with $c = (d - O(\sqrt{d}))/2d$. Applying Lemma 17 completes the proof of Theorem 16. □

Theorem 16 tells us that any local search algorithm must examine at least a constant fraction of the vertices of almost every typical neighborhood graph, in the worst case. The reason for this is that almost every such graph is an expander graph. This suggests that local search is a generically difficult task, unless the structure of the function being minimized is restricted in some way. What are the implications for local search algorithms? We should try to take into account the structure of the objective function f when designing the local search algorithm.

2.5 A best possible randomized algorithm on the hypercube

In traditional worst-case analysis we imagine our algorithm A pitted against an omniscient adversary who can select f to foil our choice of A. What if the adversary does not known A in advance? If A is permitted to randomize its

actions, this question is equivalent to a worst-case analysis of randomized algorithms. Aldous [1983] shows that we still do exponentially badly, though the number of steps decreases from order 2^d to roughly order $2^{d/2}$. In particular, Aldous defines the following *local search game* on Q^d:

- Player I chooses a unimin function $f: Q^d \to \mathbb{Z}$, trying to make the number of steps large.
- Player II simultaneously chooses an algorithm, A, trying to minimize the number of steps.
- The outcome of the play is the number of function evaluations, $NF(f, A)$, that A requires to find the minimum of f.

Since only relative values of f matter, each player has a finite number of pure strategies. Thus, by Von Neumann–Morgenstern theory [Von Neumann and Morgenstern, 1944] the game has a value

$$v = \sup_{\mu} \inf_{\zeta} \mathbb{E}_{\mu\zeta}(NF(f, A)) = \inf_{\zeta} \sup_{\mu} \mathbb{E}_{\mu\zeta}(NF(f, A)),$$

where μ and ζ are probability distributions on the set of possible functions f and algorithms A, respectively.

The value of v is essentially $2^{d/2}$.

Theorem 18 [Aldous, 1983] *The value v of the local search game on Q^d satisfies*

$$\frac{\ln v}{d} \to \frac{\ln 2}{2} \text{ as } d \to \infty.$$

Proof sketch For the upper bound, player II could use the following *randomized multistart standard algorithm* (corresponding to distribution ζ over possible A):

- Step 1 Pick $m = 2^{d/2}$ vertices $v_1, \ldots, v_m$ randomly from Q^d. Let $j \in I$ have the least f value among $v_1, \ldots, v_m$.
- Step 2 Perform the standard algorithm starting from j.

The idea is that we can expect i to have f value about $\sqrt{n}$ (where $n = 2^d$) from the minimum, since it is the best of $\sqrt{n}$ random selections from the list ordering of length n. So the standard algorithm can expect to take $O(\sqrt{n})$ iterations, corresponding to $\sim d\sqrt{n}$ additional function evaluations in Step 2. Regardless of f, this choice of ζ gives ζ gives $\mathbb{E}_{\zeta}(NF(f, A)) = O(d2^{d/2})$, where the constants implicit in the O are independent of the choice of f.

For the lower bound, player I could use the following random unimin function, corresponding to a distribution μ over the possible unimin functions f.

Execute a continuous-time random walk $X_t : t \geqslant 0$ on the vertices of Q. Times between transitions are exponentially distributed with mean 1, and transitions from a vertex are equally likely to occur to its d neighbors. The walk begins at a random vertex, so $\mathbb{P}(X_0 = i) = 1/2^d \forall i \in Q$. Let H_i denote the first hitting time of the walk X_t at vertex i. That is, $H_i = \inf_{t \geqslant 0}\{X_t = i\}$, the time at which the random walk first reaches vertex i.

These hitting times H_i define a random unimin function $f(i) = H_i$ because, as the walk traverses neighbor to neighbor, every vertex except the initial one is reached after at least one of its neighbors. We can only outline the proof that this μ achieves the claimed lower bound.

Assume that the first vertex evaluated by A has hitting time value roughly $2^{d/2}$. Now suppose, inductively, that α is the set of vertices that A has already evaluated. Let $\Delta = d^2$. If we go backwards in time by Δ from the smallest hitting time β of all vertices in α, the location of X_t at $t < \beta - \Delta$ is sufficiently randomized, i.e., independent of α, for *any* vertex $j \notin \alpha$, $\mathbb{P}(H_j < \beta - \Delta) = O(2^{-d/2})$. This is a kind of rapid mixing property. If the location of X_t were truly random, immediately we would know that the chance of any j being hit in the time interval $0 \leqslant t \leqslant 2^{d/2}$ was $O(2^{-d/2})$. It turns out that, regardless of α, this is true within Δ steps. Intuitively, the algorithm A can use information about the location of its best vertices to decrease f by at most Δ, or it can try to take an $O(2^{-d/2})$ chance on being lucky. The actual proof is greatly complicated by the fact that the random walk does not reverse in time so simply, because it is not permitted to reenter any vertex of α during the time $t < \beta$. $\square$

We may recast Theorem 18 as a worst-case analysis of randomized algorithms. Player II's mixed strategy, which is a randomized algorithm, is guaranteed to have expected cost $2^{d/2}\ d$ or better against any pure strategy of player I's, i.e., against any specific function f. Thus the randomized multistart standard algorithm is guaranteed to perform at this expected level. On the other hand, every randomized algorithm faces expected cost at least $\sim 2^{d/2}$ against the mix of functions generated by the mixed strategy of player I. Hence, every randomized algorithm faces expected cost at least $\sim 2^{d/2}$ against some specific function f. Therefore no randomized algorithm has a significantly better worst-case expected cost than the randomized multistart standard algorithm.

We also observe a simple consequence of Theorem 18. Define a modified game on Q^d where the function f chosen by player I need not be unimin, and where player II needs only to find a local minimum of f. Let v' be the value of this game. Then $(\ln v')/d \to (\ln 2)/2$ as $d \to \infty$.

On the one hand, $v' \geqslant v$ because any of player I's pure strategies for the original game is also a pure strategy for the modified game, and f unimin means player II must find the global minimum. On the other hand, the proof of the *upper* bound of Theorem 18 doesn't actually rely on f being unimin. If in Step 2 the standard algorithm starts from j and stops at a value higher than the minimum, so much the better.

Corollary 19 *The best possible randomized algorithm to find a local minimum of a function on Q^d, or to find the minimum of a unimin function on Q^d, requires, in the worst case, $2^{d/2}\ O(d)$ function evaluations in expectation.*

If the game permitted f arbitrary but required player II to globally minimize f, then the value would be 2^d. It is worth emphasizing that Theorem 18 is constructive, both in the randomized algorithm and in the probability measure on the functions.

Finally, observe that the randomized multistart algorithm has the same order average performance as long as the neighborhood function $\mathcal{N}$ has small degree.

Proposition 20 *Let $\mathcal{N}$ be any neighorhood function on $\mathcal{S}$ bounded by $d \geqslant |N(i)|$ $\forall i$. Then the randomized multistart algorithm finds a local minimum of f on $\mathcal{S}$ in worst-case expected effort $O(d\sqrt{|\mathcal{S}|})$.*

2.6 Summary

For arbitrary symmetric neighborhood functions $\mathcal{N}$, the worst-case cost of finding a local optimum is accurately estimated by $s(G)$, the value of the separation game on the associated graph G. The standard algorithm can cost much more than $s(G)$, whereas the divide-and-conquer method is provably nearly as efficient as possible. For almost all neighborhood functions, *any* algorithm must examine at least a constant fraction of the points of $\mathcal{S}$, in the worst case, to find a local optimum. A randomized algorithm can find a local optimum of $\mathcal{S}$ by examining only about $\sqrt{|\mathcal{S}|}$ points.

What implications do these results have for local search algorithms? On the one hand they suggest that we look for methods different from the standard method, if we wish to have a better worst-case performance. Arguing for the standard method, however, we must point out that it is fast on average (see Section 4), and also that it solves a different problem than 'find a local optimum'. The standard algorithm is given an initial solution x, and finds a local optimum at least as good as x. This is a crucial feature, since local improvement is so often used as the second step in a hybrid algorithm, which builds a good starting solution via some other heuristic methods. So even if a divide-and-conquer algorithm were found to have better worst-case performance than the standard method, the standard method might well prevail as the method of choice because of its other desirable features.

3 THE STANDARD ALGORITHM: ANALYSIS AND VARIATIONS

This section contains a somewhat scattered set of results, in which more structure is put on the neighborhood graph G, the function f, or the local search algorithm. We will begin by examining the worst-case behavior of the standard local improvement algorithm on the hypercube Q. This behavior will turn out to be quite poor, leading us to consider restrictions of f, partly in hope of getting better performance on a smaller problem class. Finally, we will discuss some interesting variations on the standard algorithm.

3.1 Basic properties of the standard algorithm on the hypercube

There are several reasons to study the standard algorithm on Q. The principal reason is that the standard algorithm is the almost universally used local search method, and that Q is the simplest natural neighborhood structure. If we think

about the standard method in its most basic form, we may arrive at general insights about local search. We also find fundamental combinatorial variations on the standard algorithm, which can be applied to almost all specific problems. A second historical reason for this study is that it contains a combinatorial abstraction of the Klee–Minty example of a linear programming instance for which the simplex method takes exponentially many steps [Klee & Minty, 1972]. This is because that example is a perturbed hypercube. Related to this is the fact that a random polytope looks very much like a hypercube, in some probabilistic models. So we are close to studying average-case LP, in a sense. Third, this is a study of pseudo-Boolean optimization, which has a number of applications, particularly when some structure is put on the function f [Hammer et al., 1988].

We already known some important things about the standard algorithm's worst-case performance on Q^d. First, from Chapter 2 we know there are instances of Flip and starting points for which the standard algorithm must take exponentially many steps to reach a local optimum, no matter how it selects among improving neighbors. Second, we have an explicit lower bound on the worst-case number of iterations as a corollary to Theorem 14, which tells us that any algorithm to find a local optimum of an arbitrary function on Q^d requires at least $(\sqrt{2/\pi} - o(d))2^d/\sqrt{d}$ queries of f. Algorithm 1 can come within a $\log d = \log\log n$ factor of this by Proposition 8. What does this tell us about the standard algorithm? Since each iteration except the first requires at most $d-1$ queries of f, and the first takes two extra queries, we get Corollary 21.

Corollary 21 *Any version of the standard algorithm requires, in the worst case, at least*

$$\left\lceil \frac{(\sqrt{2/\pi} - o(d))2^d/\sqrt{d} - 2}{d-1} \right\rceil \sim 2^d/d^{1.5}$$

iterations to find a local optimum of a unimin function of Q^d.

Neither of these results subsumes the other. Both are strong in applying to the standard algorithm regardless of selection method. The Flip result further restricts f to be polynomially computable, a very strong restriction. The other result gives a larger bound and permits us to require f to have a single local minimum.

What if the standard algorithm is not clever in its selection among improving neighbors? The Gray code is a well-known Hamiltonian traversal of Q. If f has values decreasing along a Hamiltonian path, the standard algorithm might visit every vertex.

Proposition 22 *The worst-case performance of the standard algorithm for unimin $f: Q^d \to \mathbb{Z}$ is $2^d - 1$ iterations.*

Obviously things can't be any worse if f is arbitrary rather than unimin. In general, and perhaps surprisingly, our results are not much more optimistic for unimin than for arbitrary functions.

This worst-case behavior predicted by Proposition 22 is actually realized in the linear complementarity problem. Fathi's [1979] worst-case analysis of principal

pivoting is a construction requiring $2^d - 1$ iterations, because the algorithm goes through all 2^d possible choices of complementary bases. Indeed, Cottle [1980] shows the sequence of bases is a transformed Gray code (obviously it would have to be some Hamiltonian path).

Now we focus on the greedy standard algorithm (GSA), which at each step selects that neighbor giving the greatest improvement. Hammer et al. [1988] show that the worst-case performance is $\Omega(2^{d/2})$. Proposition 22 does not apply, but Corollary 21 does, giving $\Omega(2^d/d^{1.5})$. Let $v_1, v_2, \ldots, x_m$ be a sequence of vertices traversed by GSA. The point x_m cannot be a neighbor of any x_j, $j < m-1$, otherwise x_m would have been selected at the jth step instead of x_{j+1}. Applying the same reasoning to the whole sequence, we get the general condition

$$x_j \in \mathcal{N}(x_i) \Rightarrow j = i \pm 1. \tag{1}$$

Corollary 21 implies there exists a sequence $x_1, \ldots, x_m$ satisfying condition (1),where $m \sim c2^d/d^{1.5}$. This sequence is nearly a snake-in-box (SIB). An SIB in a graph is a cycle satisfying condition (1). If we increase d by 2 and connect two copies of the path, we can turn it into a cycle of the same order. Our interest is in the reverse direction, however, since the best of the known bounds on SIBs in hypercubes are better. Evdokimov [1969] gives the first $\Omega(2^d)$ construction, and Röpling-Lenhart [1988] constructs SIBs of length $> (5/24)/2^n - 44$. From this we get Theorem 23, which applies to any variant of the standard algorithm, not only GSA.

Theorem 23 *Any version of the standard algorithm requires, in the worst case, at least*

$$\tfrac{5}{24} 2^d - 45$$

iterations to find a local optimum of a unimin function on Q^d.

Proof Let $v_1, \ldots, v_m$ be any path in Q^d satisfying condition (1). Assign to these vertices the smallest m values of f. By favoring vertices that are closer to the path, assign larger f values to the other vertices to make f unimin. Then the standard algorithm, starting at the mth smallest value, will have only one improving choice for the next $m-1$ iterations. □

Before proceeding further we observe a few properties of the results described so far. First, the snake-in-box result and the Hammer et al. result (though they do not make this claim) apply to any variant of the standard algorithm. That is, there exist instances and starting points such that any decreasing path to the minimum is exponentially long. Second, these results are all constructive. The other results are fully constructive and indeed require only polynomial time. This is seen by inspection of Röpling-Lenhart [1988] and Hammer et al. [1988].

3.2 Structured functions on the hypercube

All these results carry over to the more restricted class of uniminmax functions, i.e., simultaneously unimin and unimax. The idea is to use one extra dimension to

put in a nicely behaved unimax function whose values are all larger than the pathological unimin function in the lower dimension.

Theorem 24 *Theorems* 18 *and* 23 *and Theorem* 6 *of Bagchi & Williams* [1989] *are true for uniminmax functions on the hypercube* Q^{d+1}.

Proof Let f be a unimin function $Q^d \to \mathbb{Z}$ with the properties necessary to prove one of the theorems cited. Define $\bar{f}(x,0) = f(x)$ and $\bar{f}(x,1) = M + 1 + \sum_i x_i$; $x \in Q^d$, where $M = \max_{x \in Q^d} |f(x)|$. If f is unimin then $\bar{f}$ is uniminmax. □

More restrictions on f have been studied. The aim is to make f structured more like a realistic optimization problem. One of the specific motivations for this topic is to try to find a simplex method pivoting rule that has polynomial-time worst-case complexity. How might we accomplish this? First, we need a combinatorial model of linear programming, possibly a family of functions of graphs that correspond to linear objectives on polyhedra. Second, we would have to solve the local optimization problem on this combination of function family and graph class.

It appears that both of these tasks are quite difficult, based on the meagreness of results to date. The first would at the least require us to fully understand the combinatorial path properties of polyhedra, which we don't. The most famous illustration here is the Hirsch conjecture (and the related d-step conjecture), unsolved for decades and reputedly difficult. These are conjectures that the diameters of the graphs of polytopes are small. As we shall see, even the most studied choice of family/graph class has resisted polynomial-time solution, though its graph class consists of the hypercube.

The value of the graphical abstraction is unknown, with regard to these questions. If it leads to a disproof or proof of the Hirsch conjecture, or a polynomial-time simplex variant, it will have panned out very well. But it is quite possible that these questions will be resolved via noncombinatorial tools.

One of the early graph classes studied arises from abstract polytopes, a structure proposed by Dantzig [Adler & Dantzig, 1974]. Results on this class include the existence of decreasing paths of length $\sim 2^{d/2}$ [Adler & Saigal, 1976, Theorem 2]. This is a weaker result than the Klee–Minty example because abstract polytopes strictly contain actual polytopes as a class.

Theorem 24 tells us that constraining f to have a single local minimum and maximum is not enough to gain subexponential worst-case performance of the standard algorithm. Perhaps this can be obtained if we constrain every lower-dimensional restriction of f. These functions are called *completely unimin*, and they are the most studied subclass of unimin functions.

Definition 6 *A function f on Q^d is* completely unimin (unimax) *if and only if f has a unique minimum (maximum) on every face of Q^d. (A k-face of Q^d is a k-dimensional subcube formed by fixing $d-k$ of the vector components at* 0 *or* 1).

Hammer et al. [1988] provide a surprising, elegant characterization of this class.

Theorem 25 *For a function f on Q, the following are equivalent:*

1. *f is completely unimin.*
2. *f is completely unimax.*
3. *f has a unique local minimum when restricted to each 2-face of Q.*
4. *f has a unique local maximum when restricted to each 2-face of Q.*

Proof A 2-face of Q is a 4-cycle. Inspection shows (3)$\equiv$(4). By symmetry it suffices to show (3)$\Rightarrow$(1) to complete the proof. Inductively, suppose f has unique minimum on every k-face of Q. For any $(k+1)$-face F suppose without loss of generality $0^{k+1} \in F$ and is the global minimum on F. Combinatorially F is a $(k+1)$-cube. For any $x \in F$, except $x = 1^{k+1}$, there is a k-dimensional face F' containing 0^{k+1} and x.

Inductively there is a decreasing path in F', hence in F, from x to 0 and x can't be a local minimum in F. But 1^{k+1} can't be a local minimum in F either. For if it were, it would be a local minimum in all k-faces containing it, so inductively $f(1) < f(x)$ $\forall x \in F$ except 0 and 1. But now consider the $x^* \in F \setminus \{0^{k+1}, 1^{k+1}\}$, where f attains its third smallest value. This vertex must be in $f^{-1}(0)$, otherwise it would be a local minimum in a face common to x^* and 0^{k+1}. Similarly $x^* \in f^{-1}(1^{k+1})$. This is impossible for $k+1 \geqslant 3$. □

Williamson Hoke [1988] derives further elegant characterizations of completely unimin functions f on Q. If $f(y) < f(x)$ for y adjacent to x, we say y is a successor of x and x is a predecessor of y. With each $x \in I$ associate an $s(x) \in I$, the successor-tuple of x, defined as $s_i(x) = 1$ if the vertex obtained from x by complementing x_i is a successor of (has smaller f value than) x.

Theorem 26 *The following are equivalent for injective f on Q^d:*

1. *f is completely unimin.*
2. *$\forall k = 0, \ldots, d\ \exists$ exactly $\binom{d}{k}$ vertices with exactly k successors.*
3. *The mapping s: $I \to I$ is injective (hence bijective).*
4. *$\forall k = 0, \ldots, d\ \exists$ exactly $\binom{d}{k}$ vertices with exactly k predecessors.*

Proof sketch (3)$\Rightarrow$(2) immediately. (2)$\Leftrightarrow$(3) by Proposition 1 of Williamson Hoke [1988] or Chapter 3 of Bronsted [1983]. This is derived from properties of the numbers of faces of different dimensions of the dual polytope. The basic idea is that $s(x)$ tells how many and on which faces x is the (unique) local minimum of f. (1, 2)$\Rightarrow$(3) by induction on d. Verify the base case $d = 2$ by inspection. By contradiction, suppose $s(x) = s(y)$ and let $\sum_i s_i(x) \geqslant 1$ be minimal among such pairs. By the inductive hypothesis $x = \bar{y}$. Hence no $v \in I$ has $H(s(v), s(x)) = 1$. In particular there are $\sum s_i(x) - 1$ forbidden successor values, which by the minimality of $\sum s_i(x)$ contradicts (2). (5) follows from (3) and part (2) of Theorem 25. □

How easy is it to minimize completely unimin functions? Hammer et al. [1988] give a tantalizing property of these functions:

Theorem 27 *Suppose f is totally unimin on Q with minimum $\hat{x}$. Then $\forall x \in Q^d$ there exists a decreasing path from x to $\hat{x}$ of length at most d.*

Proof Inductively it suffices to show the existence of a decreasing path of length $\leqslant k+1$ from 1^{k+1} to minimum 0^{k+1} in the subcube. As in the proof of Theorem 25, 1^{k+1} is not a local minimum. Then there exists in the subcube a vertex $y \in N(1^{k+1})$ with $f(y) < f(1^{k+1})$, while inductively there is a decreasing path from y to 0 of length at most k. □

Bagchi & Williams [1989, Theorem 6] generalize Theorem 27 to neighborhoods of Hamming distance k. They show there always exists a decreasing path to the minimum, of length of most

$$2\left\lfloor \frac{n}{k+1} \right\rfloor + \left\lceil \frac{n(\mathrm{mod}\ k+1)}{k} \right\rceil .$$

It is an open problem to find a polynomial-time algorithm that minimizes completely unimin functions on Q, under the oracle model, or to show that none exists. This problem may have considerable importance, depending on the applicability of the abstract model. Theorem 27 shows that this function class satisfies the corresponding statement of the Hirsch conjecture. A positive solution to the problem might lead to a combinatorial polynomial-time pivoting rule for linear programming, or solve a useful class of pseudo-Boolean optimization problems. Alternatively, a negative solution might rule out a class of simplex pivot rules as inadequate.

3.3 Variations on the standard algorithm

If our goal is to optimize worst-case performance, much of the theoretical evidence in this and the preceding chapter points away from the standard algorithm. As we shall see in Section 4, the best theoretical support for the standard algorithm lies in its average, not worst-case performance. But even from the worst-case point of view, in some cases all hope for the standard algorithm is not lost. Here we describe some variations that may hold some promise.

Omniscient algorithms

An omniscient standard algorithm has perfect information at no cost, but is still required to travel from vertex to improving at each iteration. Here the question is whether there exists any short improving path to the optimum, to provide a bound on how well any variant of the standard algorithm could perform.

We have already covered all the known results of this type. The negative results are Theorem 13 of Chapter 2, giving exponential bounds for PLS-complete problems via 'tight' reductions, including polynomial-time computable functions on Q^d; Theorems 23 and 24 for uniminmax functions on Q^d; the positive results are Theorem 27 and its generalization by Bagchi & Williams giving $O(d)$ bounds for completely unimin functions on Q^d.

Greedy versus random standard algorithm

When more than one neighbor of $i \in S$ has lower cost than $f(i)$, how should the standard algorithm choose among them? Two of the most popular implementations of the standard algorithm are the greedy, which selects the least-cost neighbor, and the random, which selects randomly among the improving choices. For example, simulated annealing is equivalent to the latter at temperature $c = 0$, except for allowing neighbors of equal cost.

Which variant is better? From the point of view of worst-case number of steps, greed is superior, since any long sequence of greedy improving steps could also be taken by the random variant. A fairer comparison would take into account the different computational effort required per iteration, and the average rather than worst-case number of steps. We defer discussion of these issues to Section 4.3.

Randomized algorithms

Kelly [1981] proves that the random standard algorithm, which selects at random among improving neighbors, defeats the Klee–Minty example in expected $O(n^2)$ iterations. Gärtner & Ziegler [1994] show there exist starting vertices for which the random standard algorithm requires $\Omega(n^2/\log n)$ iterations. Williamson Hoke [1988] generalizes Kelly's upper bound to a subclass of the unimin functions. Call a $(d-1)$-face of Q^d *absorbing* if all its vertices have smaller f value than their neighbors on the complementary face. No variant of the standard algorithm can ever leave an absorbing face. Then we say that an injective function $f: Q^d \to \mathbb{R}$ is *fully absorbing* if $d = 1$ or Q^d has an absorbing facet that is fully absorbing.

Theorem 28 [Kelly, 1981; Williamson Hoke, 1988] *The expected number of iterations of the random standard algorithm, from any starting point, is at most* $\binom{d+1}{2}$ *for any fully absorbing* $f: Q^d \to \mathbb{Z}$.

Proof Let F denote the absorbing $(d-1)$-face of Q^d. Since f is injective, every $x \in \bar{F}$ has a neighbor $y \in F$ with $f(y) < f(x)$. Intuitively, since the probability is at least $1/d$ of selecting this neighbor, we should expect to leave the $\bar{F}$ face forever in $O(d)$ steps. Let M_c equal the maximum expected number of steps from any vertex in the c-dimensional fully absorbing face of Q^d, $c = 0, \ldots, d$. Then

$$M_c \leqslant 1 + \frac{M_{c-1}}{b} + \frac{b-1}{b} M_c,$$

where b is the number of successors (improving neighbors) at the vertex defining M_c. Since $M_{c-1} \leqslant M_c$ and $b \leqslant c$, we have $M_c \leqslant M_{c-1} + c$, whence $M_d \leqslant \sum_{c=0}^{d} c = \binom{d+1}{2}$. $\square$

Hoke makes the following conjecture.

Conjecture 29 [Williamson Hoke, 1988] *Theorem 28 is true for all completely unimin functions.*

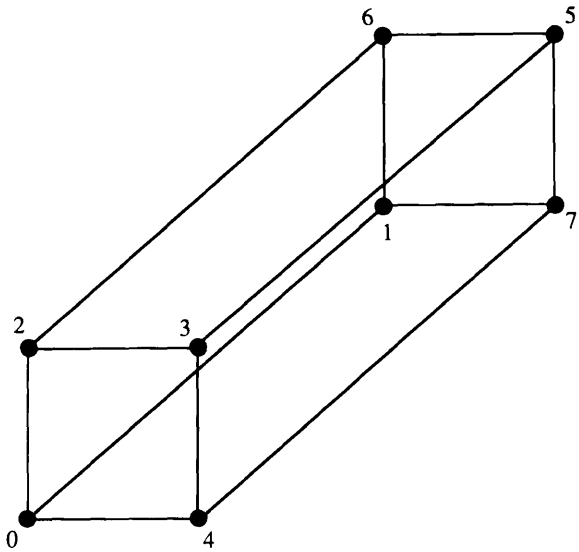

Figure 3.3 Fully absorbing function with two local maxima

Williamson Hoke proves that every Hamiltonian saddle-free f is fully absorbing, and suggests that fully absorbing implies completely unimin. However, note that every fully absorbing function is unimin but not necessarily uniminmax (Figure 3.3) and hence not necessarily completely unimin. Theorem 24 shows that Theorem 28 does not hold for all uniminmax functions.

Perhaps the main interest of the randomized algorithm is as a simplex pivot rule. No one has found a polyhedron for which the randomized pivot simplex method has a superpolynomial expected number iterations. Liebling [1981] conjectures that this algorithm runs in worst-case polynomial expected time. The conjecture is true for simplices (cyclic polytopes) [Liebling, 1981; Ross, 1982] and for assignment polytopes [Fathi & Tovey, 1986]. The graphical abstraction is useful in the former case, but not in the latter.

Lindberg & Ólafsson [1984] initiate the study of the randomized algorithm on the assignment polytope. They analyze the assignment problem from a combinatorial point of view, in an attempt to explain the good observed performance of simplex on assignment problems, and to provide evidence for their conjecture that the randomized simplex algorithm has expected polynomial performance, at least on assignment polytopes. They generate random instances and study the lengths of improving paths, and conclude that a suitable amount of randomization of the simplex method may lead to polynomial growth in the expected number of iterations.

However, subsequent analysis of the assignment problem by Orlin [1985], and independently by Dantzig, does not support this conclusion. Orlin's analysis is not related to the combinatorial abstraction. Instead, Orlin analyzes the amount

of change in objective function value achieved by best gradient pivots. He shows that these pivots improve the objective function by a great enough amount to guarantee only polynomially many pivots. Indeed, the algorithm need not select the best gradient pivot at each iteration. If $1/m$ of the pivots are best gradient, Orlin's bound on the expected number of iterations need only be multiplied by m. Fathi & Tovey [1986] use this property to prove Lindberg & Ólafsson's conjecture. They show the expected number of iterations required by the randomized simplex method on an n by n assignment problem is $O(n^7 \log n)$. The idea is that, some fraction of the time and with extremely high probability, randomized pivoting will coincide with the best gradient pivot.

So in this case the graphical abstraction doesn't do so well in explaining the performance of the algorithm. There are many examples where the best of the known performance bounds for an algorithm are derived by analyzing changes in the objective function value (or some related function). The graphical abstraction approach tends to depend only on the relative values of the function at different points – not always the most crucial information.

Affirmative action algorithms

Affirmative action algorithm, like the randomized variations on the standard method, were devised to defeat Klee–Minty. The main idea is to favor elements that have not been selected as much as others have in previous iterations. For example, the LU (least used) rule selects, among all eligible (improving) elements, one that has been selected the least number of times in the past. Zadeh [1980] found that the LU selection rule defeats the Klee–Minty example in polynomially many steps. Fathi & Tovey [1986] identify two other rules with the same property: LRU (least recently used) which selects an eligible element that has been chosen least recently and LRC (least recently considered) which cycles through the elements, selecting the first eligible element it encounters each step, and continuing its cycling from the element just selected. The LRC rule applied to the shortest path problem, viewed in the dual linear programming formulation, gives the theoretically fastest known fully polynomial algorithm for graphs with negative arc lengths, but is not computationally effective on random linear programs with general structure.

The worst-case performance of the simplex method according to any of these rules is unknown. Analysis of these affirmative action rules is not easy; constructions are especially difficult because of their anti-inductive, dimension- seeking nature. Each variable corresponds to a dimension in the search space. The idea of affirmative action is to remember which dimensions have already been explored, and to favor eligible unexplored dimensions. In contrast, when we construct an example inductively, we typically combine several low-dimensional copies of the construction to form a single higher-dimensional example. Then we understand the behavior of an algorithm on the higher-dimensional example, by already knowing inductively how it will behave within each of the lower-dimensional copies. But an affirmative action algorithm makes this kind of analysis impossible.

For the linear complementarity and maximum flow problems, these rules cycle or have poor worst-case performance [Fathi & Tovey, 1986].

3.4 Summary

Analysis on the hypercube suggests some interesting variations on the standard algorithm. Because of their generality, these variations have many possible applications, including network algorithms and linear programming. On the other hand, these variations can have pitfalls of their own. It is a tantalizing thought that there may be a variant that is guaranteed always to be very fast for some classes of functions, as suggested by Theorem 27.

4 AVERAGE-CASE ANALYSIS

We now study the average-case behavior of local improvement. The worst-case analysis in the previous section yields gloomy predictions about local improvement, as do the corollaries to the complexity analysis in Chapter 2 and the analyses of specific algorithms applied to specific problems cited earlier. But decades of empirical evidence yield the most cheerful assessment of local improvement. In practice, it finds a local optimum quickly, in a number of iterations small compared with the size of the search space.

Our principal goal is to understand why the standard algorithm meets with so much empirical success. In view of the worst-case results, the explanation must be probabilistic. We also get some confirmatory evidence for our abstract approach, and a rule of thumb comparing the random and greedy variants of the standard algorithm.

A *caveat* to the reader: there is no generally agreed satisfactory model of a random function on a graph. Nonetheless, under a few different random function models, the standard algorithm turns out to be quite fast on average. In addition we get some information about the relative speed of different variants of the standard algorithm. And we get the surprising insight that the *same* structural property of the neighborhood graph causes both the exponentially bad worst-case and the polynomially good average-case performance of the standard algorithm. The key isoperimetric structural property is nearly universal among graphs. This suggests that the coupling of poor worst-case and good average-case speeds is a nearly universal property of the standard algorithm.

Our first result applies to the greedy or random standard algorithm under the random function model [Tovey, 1985].

Theorem 30 *Let f be a random function on Q^d. Then the average number of iterations required by any variant of the standard algorithm is less than* $(7/5)\,ed$.

Proof Consider any fixed path P of edge length k in the graph Q^d. As the list of vertices ordered by function value, all permutations are equally likely to occur, so all $(k+1)!$ of the possible subsequences of vertices from P are equally likely to occur. Therefore the probability that f is increasing along P is $1/(k+1)!$. The

number of possible paths P of edge length k in the graph Q^d is less than $2^d d^k$. Then the probability that there exists an improving path of length k is less than

$$\frac{2^d d^k}{(k+1)!} < (0.49)^d$$

when $k = \frac{7}{5}ed$. No variant of the standard algorithm can take more than $k-1$ steps if no improving path of length k exists, and 2^d bounds the number of steps in any case. The average number of iterations is less than $\frac{7}{5}ed - 1 + 2^d(0.49)^d \leqslant \frac{7}{5}ed$. □

The key to the proof of Theorem 30 is the rapid growth of $k!$ compared with d^k. This gives us some slack and allows a relaxation of the probabilistic assumption. Also, the only properties of Q^d we used were that each vertex has degree at most d, and that the total number of vertices is 2^d. This allows for a generalization to the class of graphs having bounds of logarithmic degree. For any injective function f on the vertices of a graph, we define the *permutation induced by* f as the permutation of V in order from least to greatest f value.

Theorem 31 *Let $G = (V, E)$ be a graph with polylogarithmic maximum vertex degree $p(\log|V|)$, where $p(x) \geqslant x$ is a polynomial. Let μ be any probability measure on the set of functions $f: V \to \mathbb{R}$ such that no permutation of V is more than $2^{2\log|V|}$ times as likely as any other permutation to be induced by f. Then the expected number of iterations of any version of the standard algorithm is less than* $\mathrm{e}(\alpha+2)p(\log|V|)$.

The proof is essentially the same as before.

We now turn to unimin functions. These are a category of independent interest, especially as linear programming and some linear complementarity principal pivoting methods, as well as some pseudo-Boolean optimization functions, are members of this class. There is no universally accepted model of a random unimin function. The average-case behavior obviously depends on this, since from Theorem 23 trivially there exist measures with respect to which the average behavior is exponential. Indeed, the hitting time of a random walk is such a measure, and an elegantly defined one at that, as we know from Theorem 18.

4.1 The uniform boundary distribution on unimin functions

We now describe a natural and very different probability measure on unimin functions, called the *uniform boundary* distribution. This distribution is only concerned with the ordinal ranking induced by f.

As in Section 2, the boundary $\partial(S)$ of a set of vertices $S \subset V$ is the set of vertices adjacent to vertices in S, excluding S itself. Temporarily let us switch to maximization. This permits us to visualize a local optimum as a mountain peak, if we map f values on the vertical axis. A unimax function is a region with a single peak. Imagine now a unimax function that is hidden from view by a flood of randomizing water. As the flood waters of randomization recede, the first vertex revealed is the maximum (the highest point). The second point revealed is the vertex with

second largest f value. The second point must be a neighbor of the highest point, otherwise it would be a local optimum, violating the unimax property. In general, if S is the set of visible points, the next point revealed by the receding flood must be a member of $\partial(S)$.

The simplest assumption to make is that each member of $\partial(S)$ is equally likely to be revealed next, since all we know for sure is that some member of $\partial(S)$ will be revealed next.

Any probability measure on unimax (or unimin) functions corresponds to a rule for selecting among the members of $\partial(S)$, given the (ordered) set S. The *uniform boundary* distribution selects uniformly among the members of $\partial(S)$.

In the unimin case, define $v_{(i)}$ as the vertex in V with ith smallest f value, and let $V_{(i)} = \bigcup_{j=1}^{i} v_{(i)}$. In the boundary distribution

$$\mathbb{P}(v = v_{(i)} \mid V_{(i)} = S) = \frac{1}{|\partial(S)|} \forall v \in \partial(S).$$

The standard algorithm has fast average performance if f is sampled from the uniform boundary distribution. The average number of iterations is small for Q^d or any expander graph.

Theorem 32 [Tovey, 1986] *Let $G = (V, E)$ be a d-regular graph. Let f be drawn from the uniform boundary distribution. Let $\beta(i)$ be a lower bound on the cardinality $|\partial(S)|$ for $S \subset V: |S| = i$. Then the expected number of iterations required by any variant of the standard algorithm, from any starting point, is less than*

$$\sum_{j=1}^{|V|-1} ed/\beta(j).$$

From a random starting point under the same conditions, the expected number of iterations is less than

$$\frac{1}{|V|} \sum_{i=1}^{|V|-1} \sum_{j=1}^{i} ed/\beta(j).$$

Proof sketch Suppose f is generated according to the receding-flood method described above. As the points $v_{(1)}, v_{(2)}, \ldots$ are revealed, we can calculate the number of iterations, $h_{(i)}$, the standard algorithm will require to get from $v_{(i)}$ to the optimum $v_{(1)}$. For example, with the greedy standard algorithm this is 1 more than the h value of its best neighbor.

The key idea is illustrated in Figure 3.4 for a 4-regular graph. Supposing we know the values $h_{(1)}, \ldots, h_{(10)}$, what can we say about the value of $h_{(11)}$? Supposing further that $\beta(10) = 18$, there must be at least 18 possible choices for the eleventh-best vertex. Of these, at most 3 can be connected to $v_{(8)}$ (because every vertex has degree 4 and at least one edge is on the path to $v_{(1)}$). Now $h_{(8)} = 4$ and all other h values are smaller. So the probability is at most 3/18 that we will end up with $h_{(11)} = 5$. In the same way, at most 3 each can be connected to $v_{(7)}$ and $v_{(10)}$; and at most 3 each to $v_{(4)}$, $v_{(5)}$, $v_{(9)}$. These are the vertices with largest h values. So even at our most pessimistic view, the value of $h_{(11)}$ can't be any

i	$h_{(i)}$
1	0
2	1
3	1
4	2
5	2
6	1
7	3
8	4
9	2
10	3

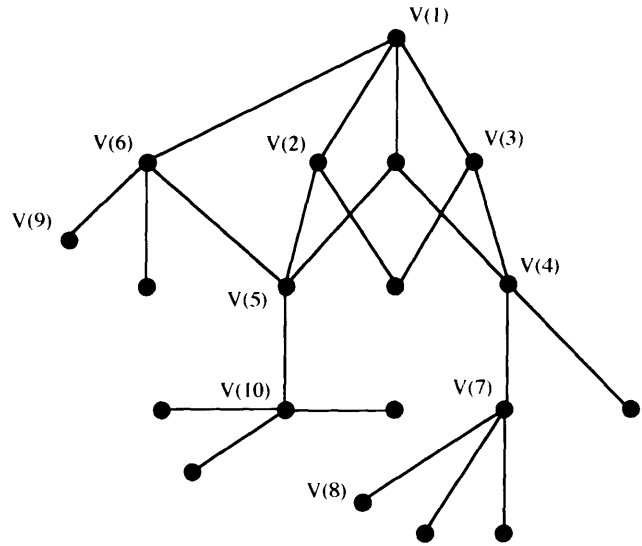

Figure 3.4 Bounding the growth of h

worse than if we randomly select a number from the multiset $\{4, 3, 3, 2, 2, 2\}$ and add 1.

In general, the value of $h_{(i+1)}$ is no worse than 1 plus a random selection from the $\lfloor \beta(i)/(d-1) \rfloor$ largest of $h_{(j)}$: $1 \leqslant j \leqslant i$. Aldous & Pitman [1983] proved that the largest value of the following process increases at average rate at most e/m: from a multiset of integers, repeatedly select one of the m largest at random, and add 1 plus its value to the multiset.

The expected value of the number of iterations required from the $(i+1)$st best value is therefore bounded by

$$\mathbb{E}(h_{i+1}) < \sum_{j=1}^{i} \frac{e(d-1)}{[\beta(j)]} < \sum_{j=1}^{i} ed/\beta(j)$$

and the theorem follows. □

Corollary 33 *The expected number of iterations required by any variant of the standard algorithm to find the local optimum of f on Q^d from a random starting point, with respect to the uniform boundary distribution, is less than ed^2.*

Proof Apply the isoperimetric bounds of Lemma 15 to Theorem 32; see Tovey [1986, Theorem 4.2] for details. □

Not only is the standard algorithm fast (on average) on Q^d, it also is fast on almost all typical neighborhood graphs.

Corollary 34 *Let $\mathscr{G}$ be a sequence of classes of the typical neighborhood graphs on n vertices, regular of degree $d(n)$. Then for almost every graph in $\mathscr{G}$, the average*

number of iterations of any variant of the standard algorithm, with respect to the uniform boundary distribution, is less than

$$(4 + o(1))\,ed \log n = O(d \log n).$$

Proof Recall that the *edge-boundary* $\partial^e(S)$ is the set of edges emanating from S, that is with one vertex in and one vertex not in S. Letting Δ_2 denote the second largest eigenvalue of the graph, as in Section 2.3, the edge-boundary size is bounded by

$$|\partial^e(S)| \geqslant \frac{(d - \Delta_2)|S|(n - |S|)}{n}.$$

See Alon, Spencer & Erdös [1992, p. 120]. Since each vertex is of degree d, we get

$$\beta(i) = \frac{(d - \Delta_2)i(n - i)}{dn} \geqslant \frac{(d - \Delta_2)i}{2d} \quad \text{for } i \leqslant n/2.$$

Since $\partial^e(S) = \partial^e(\bar{S})$, we get

$$\beta(i) \geqslant \frac{(d - \Delta_2)(n - i)}{2d} \quad \text{for } i > n/2.$$

Now applying Theorem 32 we have a bound of

$$ed \sum_{i=1}^{n} 1/\beta(i) \leqslant 2ed^2/(d - \Delta_2)\left[\sum_{i=1}^{n/2} 1/i + \sum_{i=n/2+1}^{n-1} 1/(n - i)\right] < \frac{4ed^2 \log n/2}{(d - \Delta_2)}$$

on the expected number of iterations. As in the proof of Theorem 16, almost every graph in $\mathscr{G}$ has second eigenvalue $\Delta_2 = O(\sqrt{d})$, and the corollary follows. □

4.2 The uniform edge-boundary distribution

If f is a random unimin function, every vertex in the boundary of $V_{(i)}$ might be $v_{(i+1)}$. The uniform boundary distribution assigns equal probability to each such vertex. An alternative distribution, the *uniform edge-boundary* distribution, gives to each vertex a weight proportional to the number of its neighbors in $V_{(i)}$. Thus vertices with many low-valued neighbors are more apt to be low-valued themselves.

All variants of the standard algorithm have fast average performance with respect to the uniform edge-boundary distribution. The random standard algorithm has a particularly simple interpretation when applied to this distribution: every edge in $\partial^e(V_{(i)})$ is equally likely to connect to $v_{(i+1)}$. This interpretation allows us to find smaller bounds for the random variant than for the greedy variant of the standard algorithm. The proofs are similar to Theorem 32 and its corollaries. For details see Tovey [1986].

Theorem 35 *The average number of iterations of any variant of the standard algorithm on Q^d with respect to the uniform edge-boundary distribution is less than* $2ed^2 \log d$.

Theorem 36 *The average number of iterations of the random standard algorithm on Q^d with respect to the uniform edge-boundary distribution is less than* $2ed \log d$.

Simulation results discussed in Section 4.3 confirm one's intuition that the greedy standard algorithm is actually faster than the random. We get a better theoretical bound on the random algorithm only because it is easier to analyze.

Theorem 37 *For almost all neighborhood functions $\mathcal{N}$ on $\mathcal{S}$, regular of degree $d \geqslant 4$, the average number of iterations of the random standard algorithm, with respect to the uniform edge-boundary distribution, is $O(d \log n)$, where $n = |\mathcal{S}|$.*

4.3 Other distributions and computational results

The *uniform unimin* is another natural distribution for unimin functions. Let f be a unimin function on search space $\mathcal{S}$, with respect to neighborhood function $\mathcal{N}$. All of the ordinal information about f is given by the induced permutation of the elements of $\mathcal{S}$. Since f is unimin, only some of the $|\mathcal{S}|$ permutations are possible. The uniform unimin distribution requires each of these permutations to be equally likely to occur.

The only known results regarding the uniform unimin distribution are computational. The results, for $d \leqslant 16$, are strikingly good. On the hypercube Q^d the expected number of iterations of both the greedy and random standard algorithm appear linear in d with coefficient less than 1 [Tovey, 1986]. Moreover, the uniform boundary and uniform-edge boundary distributions show virtually identical performance. This leads us to conjecture stronger bounds than given by Theorem 36.

Conjecture 38 *The expected number of iterations of both greedy and random standard algorithms is $O(d \log \log d)$ under the uniform unimin, uniform boundary, and uniform edge-boundary distributions.*

Based on the simulations of Tovey [1986], it appears that the random standard algorithm requires about 15% more iterations than the greedy standard algorithm to find a local optimum on Q^d. This 15% estimate is the same with respect to all three distributions. If $\mathcal{S}$ is the set of all $i \in \{0, 1\}^d : \sum_t i_t = k$, and i and j are neighbors if their Hamming distance is 2, then simulations predict a 50% difference between the random and greedy variants. These predictions are consistent with experimental results for integer programming and linear programming, respectively [Tovey, 1983].

Though the greedy standard algorithm takes fewer iterations on average, it is more costly per iteration. If the random standard algorithm samples the neighborhood $\mathcal{N}(i)$ without replacement, the expected effort per iteration is at most $(1 + |\mathcal{N}(i)|)/2$, except in the last iteration. This is because the expected number of samples required to find an improving step is $0.5 + |\mathcal{N}(i)|/2$ even when there is only one better-valued neighbor. If there are multiple improving steps the expected search time decreases. On the other hand, the greedy algorithm must evaluate all $|\mathcal{N}(i)|$ neighbors. Thus we can predict that the random standard

algorithm will require about 60% or fewer function evaluations than the greedy. For problems with multiple local optima, however, this is still not the whole story. It seems plausible that the greedy algorithm will tend to reach better-quality local optima than the random will. There is some anecdotal evidence in support of this presumption, but careful empirical or theoretical evaluation is still lacking.

The theoretical bounds in the previous section all extend to classes of distributions on nonunimin functions. To generalize the uniform boundary distribution, we require that the distribution treats all members of the boundary with equal probability. Any point not in the boundary may have arbitrary probability. All the bounds in the previous section remain valid. This is because selecting a point not in the boundary introduces a new local optimum, with best possible h value 0.

As a corollary to Corollary 33, we have Corollary 39.

Corollary 39 *Let f be a random function on Q^d. Then the average number of iterations required by any variant of the standard algorithm is less than en^2, but this is not as good a bound as given in Theorem 30.*

Corollary 33 contrasts very strongly with Theorem 18. In Theorem 18 the hitting times of the random walk form a probability distribution on unimin functions. Every version of the standard algorithm has expected performance $\Omega(2^{d/2})$ with respect to this distribution. Why is the distribution derived from hitting times so different from the boundary distribution? Figure 3.5 provides some intuition. Let S denote the set of exposed points (the ones with best f values). The ordering formed by hitting times will tend to consist of paths emerging from S, traveling at random, and reentering S. Hence the paths are apt to be like snakes-in-boxes,

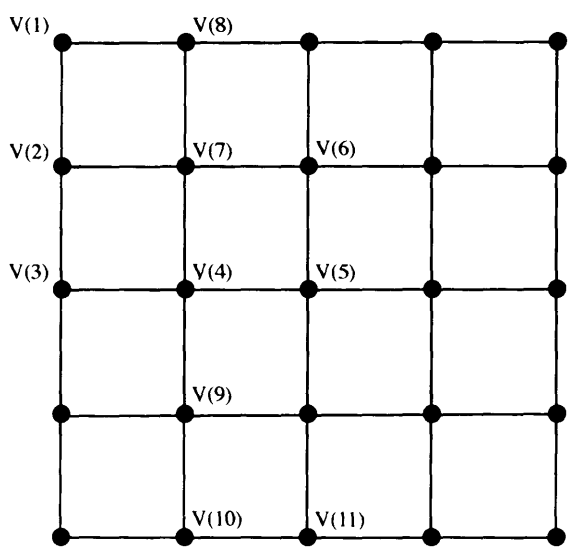

Figure 3.5 Hitting times make the standard algorithm slow

rather than gliding along the boundary of S. The path lengths of vertices on these paths will be likely to increase by one at each step, resulting in very long path lengths. So the hitting times put enormous probability on the last vertex revealed being the parent of the next vertex revealed, almost the opposite of the equal probabilities for the boundary distribution.

Since decades of experience with many optimization problems are consistent with the boundary distribution and inconsistent with the hitting time distribution, we can infer that the hitting time distribution is not a realistic model for random instances of an optimization problem.

4.4 Summary

The standard algorithm is fast on average, both for arbitrary and unimin functions. We have developed some appealingly simple models that help explain these phenomena and can make some useful predictions as well. In particular, we have some estimates of the relative speeds of the greedy and random variants.

Theorem 16 together with Corollaries 34 and 39 suggest that the bad worst-case and good average-case behavior of the standard algorithm are nearly universal properties of neighborhood functions. It is surprising, perhaps, that these two strongly contrasting behaviors are almost universal. It is remarkable that they stem from the *same* graphical property of being an expander.

5 MULTIPLICITY AND QUALITY OF LOCAL OPTIMA

Most of the results in this chapter have revolved around the question, 'How long does it take to find a local optimum?' When minimizing a unimin function, this is the only important question. But for general functions, questions about the quantity and quality of local optima become essential. How many local optima are there, and how good are they?

Unfortunately there are almost no results in this area, and the few that we have suggest that our general approach doesn't lend much insight. This is especially true about questions regarding the quality of local optima. The discrete models of this chapter emphasize the ordinal information about function values, whereas quantities such as performance ratios may be lost [Grover, 1992; Chandra, Karloff & Tovey, 1997].

As a preliminary, suppose we have a function f and we wish to determine whether or not it has multiple local optima. If f is arbitrary and can only be accessed by an oracle evaluating f at given $x \in Q$, it is simple to show that many evaluations are required in the worst case.

Proposition 40 *Under the function evaluation model, determining if an arbitrary injective $f: Q^d \mapsto \mathbb{Z}$ is unimin requires at least $2^d - d$ queries in the worst case.*

Proof Let x^i be the ith vertex queried. An adversary can set $f(x^1) = 0$ and $f(x^i) = HD(x^i, x^1)$ for $i < 2^2 - d$. When the last vertex $x \notin N(x^1)$ is queried,

the adversary may respond 0, making f not unimin, or $HD(x, x^1)$, making f unimin. □

What if f is known to be in a certain explicitly representable class? There are several results of the following form: it is coNP-complete to determine that a function from a certain class is unimin. One result has to do with a class of functions on arbitrary graphs. For a graph $G = (V, E)$, Bandelt considers a class of functions $F_W: V \to \mathbb{Z}$ parameterized by $W \subset V$. For particular W the function $F_W(v) = \sum_{w \in W} d(w, v)$, where d denotes the distance in the graph G. Bandelt proves the following theorem by reduction from exact cover.

Theorem 41 [Bandelt, 1989] *The question, 'Is F_W unimin for all $W \subset V$?' is co*NP*-complete, even if $G = (V, E)$ is restricted to be chordal.*

Another result due to Crama pertains to a class of functions on the hypercube. Define a pseudo-Boolean function of order k as a mapping $p: Q^d \mapsto \mathbb{R}$ where $p(x_1, \ldots, x_d)$ is a polynomial of degree k. Every pseudo-Boolean function is equal to one of order d or less, but even the cubics exhibit 'bad' properties.

Theorem 42 [Crama, 1989] *It is co*NP*-complete to determine that a given third-order pseudo-Boolean function is unimin.*

Proof The reduction is from an instance K, $c_1, \ldots c_m$ of Partition (Is there a subset of the integers c_i that sums to K?) to the complementary problem. Define for any Boolean vector $x \in Q^m$ the function $c(x) = \sum_{i=1}^{m} c_i x_i - K$. So $c^2(x) \geqslant 1 \forall x \in Q^m$ unless the Partition instance has answer 'yes'. Let $d = m + 2$ and let the cubic function, where $x \in Q^m$, be

$$f(x) = 16c^2(x) + 4x_{n+1}x_{n+2} - 4c(x) - (8c^2(x) + 4)x_{n+1} - (8c^2(x) + 3)x_{n+2}.$$

On the one hand, if x^* solves Partition, both $(x^*, 0, 1)$ and $(x^*, 1, 0)$ are local minima. On the other hand, if $c^2(x) \geqslant 1 \forall x$ then $(1, \ldots, 1)$ is the unique local minimum of f on Q^d. □

Bagchi & Williams [1989] extend Theorem 42 to the neighborhood $\mathcal{N}^k(x)$ consisting of all points at Hamming distance at most k from x.

Theorem 43 [Bagchi & Williams, 1989] *For every integer $k \geqslant 2$ determining if a given pseudo-Boolean function of order $k + 1$ is unimin, with respect to the neighborhood $\mathcal{N}^k$, is co*NP*-complete.*

Now let us face more squarely the question of multiplicity of local optima on the graph $G = (V, E)$. If f is not injective then f constant makes every vertex $v \in V$ a local minimum. The worst case for injective f is also immediate: the maximum number of local minima of an injective f on G is $\alpha(G)$, the independence number of G. On the hypercube Q^d there can be at most 2^{d-1} local minima.

These observations are in rough agreement with some analyses of specific problems [Papadimitriou & Steiglitz, 1978; Tovey, 1985; Rödl & Tovey, 1987], which construct instances containing exponentially many local optima.

How many local optima occur on average? This question, too, is immediately answered. Indexing the vertices of G arbitrarily, let d_i denote the degree of the ith vertex.

Proposition 44 *Under the random function model, the expected number of local minima equals* $\sum_{i=1}^{|V|} 1/(d_i+1)$. *For d-regular graphs the expected number is* $|V|/(d+1)$.

One of the fundamental problems in local search is to understand the properties of different neighborhoods. In the spirit of Bagchi & Williams [1989] let us extend Proposition 44 to $\mathcal{N}^k$ neighborhoods.

Corollary 45 *A random function on* Q^d *has expected number*

$$\frac{2^d}{\sum_{j=0}^{k}\binom{d}{j}} < \frac{2^d}{(d/k)^k}$$

local minima with respect to $\mathcal{N}^k$ *neighborhoods.*

Proof The resulting graph is regular with degree $\sum_{j=1}^{k}\binom{d}{k}$. Apply Proposition 44. □

Still, none of the results about multiplicity of optima are very satisfactory. It is not clear they tell us much about the multiplicity of local optima for any specific optimization problem. The difficulty with the random function model is that real problems have much more correlation in the function values of neighboring vertices. They also have fewer local optima, on average, than predicted by Proposition 44. It seems likely that there is a causal relation between these two differences. It is an open research problem to find a more satisfactory general model of a random function on an arbitrary or d-regular graph, where there is some correlation between function values of neighboring vertices.

Let us try to turn a failure of our random function model to our advantage. If results obtained in practice are better than the predictions, we examine the model to see where it differs from reality. The differences can give us clues as to how to design better local search neighborhoods.

Compare the 2-opt neighborhood for the m-city TSP with an arbitrary randomly generated $m(m-3)/2$-regular neighborhood. In the random case the expected number of local optima is as in Proposition 44, with the randomization coming from the graph. But in the 2-opt case empirical studies tell us the expected number of local optima is smaller. The 2-opt neighborhood is superior to a randomly generated neighborhood in the sense of having fewer local optima. Indeed, Savage [1976] conjectures that 2-opt is the best neighborhood of its degree, over all data-independent neighborhoods, in terms of maximizing the probability that the resulting cost function is unimin. Along the same lines one might also conjecture that 2-opt has the smallest expected number of local optima.

The 2-opt neighborhood can be characterized as the neighborhood (of its degree) that induces the greatest correlation between function values of neighbor-

ing tours, because neighboring tours differ in the minimum possible four edges, and the cost is the sum of the edge costs. We speculate that 2-opt's superiority is due to that property. If this is true, it suggests that we should try to maximize correlation when designing (data-independent) neighborhoods for other optimization problems.

We conclude by pointing out that there are other more accurate questions to ask. But there is no hope at present of answering them. For example, for an injective function f defined on the vertices of Q, let $r(x): x \in Q$ denote the local optimum the greedy standard algorithm would reach if initiated at x. Then $\mathbb{E}_x(f(r(x)))$ is not the same as the average value of f taken over the local optima, since different minima will have different size regions of attraction. Although intuitively one would expect $\mathbb{E}_x(f(r(x)))$ to be smaller, because deeper minima ought to have larger regions of attraction, a rigorous analysis is beyond present-day scope.

ACKNOWLEDGMENTS

I thank Ingo Althöfer, Imre Leader, Béla Bollobás, Kathy Hoke, Peter Hammer and colleagues at RUTCOR, Nick Pippinger, Elisheva Sachs, Pete Winkler, and John Vande Vate for comments and correspondence. The author was supported in part by NSF grant DDM-9215467.

Local Search in Combinatorial Optimization
Edited by E. Aarts and J. K. Lenstra
© 1997 John Wiley & Sons Ltd.

4

Simulated annealing

Emile H. L. Aarts, Jan H. M. Korst
Philips Research Laboratories, Eindhoven

Peter J. M. van Laarhoven
Eindhoven University of Technology

1 INTRODUCTION 91
2 THRESHOLD ALGORITHMS 92
3 A QUALITATIVE PERFORMANCE ANALYSIS 94
4 THE PHYSICS ANALOGY 96
5 MARKOV MODELS 98
6 A HOMOGENEOUS MODEL 101
7 AN INHOMOGENEOUS MODEL 104
8 ASYMPTOTIC BEHAVIOR 109
9 COOLING SCHEDULES 111
9.1 Optimal schedules 113
9.2 Heuristic schedules 113
10 ISSUES FROM PRACTICE 116
11 SPEEDING UP 118
12 COMBINED APPROACHES 120

1 INTRODUCTION

Simulated annealing belongs to a class of local search algorithms that are known as threshold algorithms. These algorithms play a special role within local search for two reasons. First, they appear to be quite successful when applied to a broad range of practical problems, which has given them quite a reputation among practitioners. Second, some threshold algorithms such as simulated annealing have a stochastic component, which facilitates a theoretical analysis of their asymptotic convergence, and this has made them very popular to mathematicians.

The emphasis of our presentation is on the mathematics of threshold algorithms, with special attention paid to simulated annealing. We discard the application of threshold algorithms to specific problems, since this issue is discussed in detail in Chapters 8 to 13 of this volume. However, we mention some general aspects of the algorithms' practical use to give the reader a feeling of what he might expect from their application.

2 THRESHOLD ALGORITHMS

Let $(\mathcal{S}, f)$ be an instance of a combinatorial minimization problem with solution set $\mathcal{S}$ and cost function $f: \mathcal{S} \to \mathbb{R}$. Furthermore, let $\mathcal{N}: \mathcal{S} \to \mathcal{P}(\mathcal{S})$ be a neighborhood function, which defines for each $i \in \mathcal{S}$ a set $\mathcal{N}(i) \subseteq \mathcal{S}$ of neighboring solutions. The question is to find an optimal solution $i^* \in \mathcal{S}$ that minimizes the cost of all solutions over $\mathcal{S}$. Consider the class of *threshold algorithms* given by the pseudocode of Figure 4.1.

The procedure INITIALIZE selects a start solution from $\mathcal{S}$, the procedure GENERATE selects a solution from the neighborhood of a current solution, and the procedure STOP evaluates a stop criterion that determines termination of the algorithm. Threshold algorithms continually select a neighbor of a current solution and compare the difference in cost between these solutions to a threshold. If the cost difference is below the threshold, the neighbor replaces the current solution. Otherwise, the search continues with the current solution. The sequence $(t_k | k = 0, 1, 2, \ldots)$ denotes the thresholds, where t_k is used at iteration k of the local search algorithm.

We distinguish between the following three types of threshold algorithms depending on the nature of the threshold:

- *Iterative improvement*: $t_k = 0$, $k = 0, 1, 2, \ldots$ Clearly, this is a variant of the classical greedy local search in which only cost-reducing neighbors are accepted.
- *Threshold accepting*: $t_k = c_k$, $k = 0, 1, 2, \ldots$, where $c_k \geqslant 0, c_k \geqslant c_{k+1}$, and $\lim_{k \to \infty} c_k = 0$. Threshold accepting uses a nonincreasing sequence of deterministic thresholds. Due to the use of positive thresholds, neighboring solutions with larger costs are accepted in a limited way. In the course of the algorithm's execution, the threshold values are gradually lowered, eventually to 0, in which case only improvements are accepted.

```
procedure THRESHOLD_ALGORITHM;
begin
    INITIALIZE (i_start);
    i := i_start;
    k := 0;
    repeat
        GENERATE (j from N(i));
        if f(j) - f(i) < t_k then i := j;
        k := k + 1;
    until STOP;
end;
```

Figure 4.1 Pseudocode of a class of threshold algorithms

- *Simulated annealing*: t_k = a random variable with expected value $\mathbb{E}(t_k) = c_k \in \mathbb{R}^+$, $k = 0, 1, 2, \ldots$. The t_k's follow a probability distribution function F_{c_k} over $\mathbb{R}^+$. Simulated annealing uses randomized thresholds with values between zero and infinity, and the probability of a threshold t_k being at most $y \in \mathbb{R}^+$ is given by $\mathbb{P}_{c_k}\{t_k \leqslant y\} = F_{c_k}(y)$. This implies that each neighboring solution can be chosen with a positive probability to replace the current solution. In practice the function F_{c_k} is chosen such that solutions corresponding to large increases in cost have a small probability of being accepted, whereas solutions corresponding to small increases in cost have a larger probability of being accepted.

As an important matter we remark that, in the original simulated annealing version, Kirkpatrick, Gelatt & Vecchi [1983] and Černý [1995] take for F_{c_k} the negative exponential distribution with parameter $1/c_k$. This choice is identical to the following *acceptance criterion*. For any two solutions $i, j \in \mathscr{S}$ the probability of accepting j from i at the kth iteration is given by

$$\mathbb{P}_{c_k}\{\text{accept } j\} = \begin{cases} 1 & \text{if } f(j) \leqslant f(i), \\ \exp\left(\dfrac{f(i) - f(j)}{c_k}\right) & \text{if } f(j) > f(i). \end{cases} \tag{1}$$

The parameter c_k is used in the simulated annealing algorithm as a *control parameter*, and it plays an important role in the convergence analysis of the algorithm. We will drop the subscript k of the control parameter if it is not explicitly needed.

Some preliminary convergence results

For iterative improvement it is not possible to give nontrivial convergence results. For *multistart iterative improment*, which consists of single runs of iterative improvement that are repeated with different start solutions, it is easily verified that an optimal solution is found with probability 1 if an infinite number of restarts is allowed.

Threshold accepting was introduced by Dueck & Scheuer [1990] as a deterministic version of simulated annealing. One of the major unresolved problems is the determination of appropriate values for the thresholds. Furthermore, as in the case of iterative improvement, no general convergence results can be proved, but one can do slightly better. Althöfer & Koschnick [1991] have related some convergence properties of threshold accepting to those of simulated annealing. The proofs of the convergence results are not constructive. They make use of the fact that, in some sense, simulated annealing generalizes threshold accepting. For instance, one of their propositions states that if there is a finite sequence of thresholds for simulated annealing, leading to an optimal solution with probability $1 - \varepsilon$ for some $\varepsilon > 0$, then there also exists a finite sequence of thresholds for threshold accepting, leading to an optimal solution with probability $1 - \varepsilon$. Furthermore, they give a simple example for which suboptimal solutions can be reached with threshold sequences of any length, demonstrating that even asymptotically the algorithm can get trapped in local minima.

Finally, for simulated annealing there exist general convergence results, which state that under certain mild conditions an optimal solution is found with probability 1. These results are obtained by analyzing the algorithm in terms of Markov chains. This central issue is the subject of Section 5 and subsequent sections. Before we present these quantitative performance results, we first discuss some qualitative results.

3 A QUALITATIVE PERFORMANCE ANALYSIS

To analyze the performance of the threshold algorithms introduced in the previous section, we consider the following combinatorial optimization problem, which is a simplified version of a problem introduced by Lundy & Mees [1986]. Let the set of solutions be given by $\mathscr{S} = \{0, 1, \ldots, N\}$, with $1 \ll N$, and let the cost function $f: \mathscr{S} \to \mathbb{R}$ be given by

$$f(i) = i - \lfloor i/n \rfloor \delta,$$

with $n \in \mathbb{N}$, $1 \ll n \ll N$, $\delta \in \mathbb{R}$, and $1 < \delta < n$. Figure 4.2 illustrates the cost function for $\delta = 2$. The problem is to find an element in $\mathscr{S}$ with minimum cost.

This problem formulation can be generalized to solution sets of d dimensions as follows. If $\mathscr{S} = \{0, 1, \ldots, N\}^d$, the cost function is given by $f\{i_0, i_1, \ldots, i_{d-1}\} =$

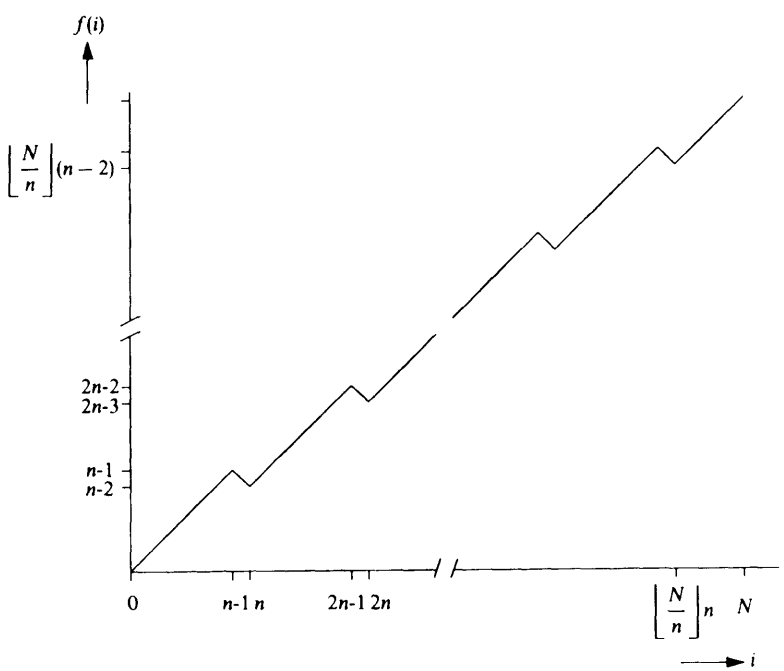

Figure 4.2 The cost function f for $\delta = 2$

$r - \lfloor r/n \rfloor \delta$, with $r = \max\{i_0, i_1, \ldots, i_{d-1}\}$. Lundy & Mees [1986] discuss a two-dimensional version of this problem.

Furthermore, let $\mathcal{N}$ be a neighborhood function defined by $\mathcal{N}(i) = \{i-1, i, i+1\}$, for $i \in \{1, 2, \ldots, N-1\}$, $\mathcal{N}(0) = \{0, 1\}$, and $\mathcal{N}(N) = \{N-1, N\}$. We now consider the performance of the threshold algorithms on this problem, for two choices of δ, namely $\delta \approx 1$ and $\delta \gg 1$, corresponding to the situations where a small or a large deterioration is required to escape from a local minimum, respectively.

The behavior of multistart iterative improvement is identical for both cases. A single iteration is described as follows. It randomly selects an initial solution i from $\mathcal{S} = \{0, 1, \ldots, N\}$ and terminates in a local optimum given by $i' = \lfloor i/n \rfloor n$. Clearly, the algorithm only finds a global optimum in a given iteration if the initial solution is in $\{0, 1, \ldots, n-1\}$. The expected number of iterations to obtain a global optimum can be shown to be $O(M)$, with $M = N/n$, for the one-dimensional case, and $O(M^d)$ for the case with d dimensions. This illustrates that multistart iterative improvement can always find a global optimum, but that the required effort may be very large – equivalent to enumerating all solutions in $\mathcal{S}$.

We now consider the behavior of threshold accepting and simulated annealing for the case that $\delta \approx 1$. For threshold accepting, we consider three situations, depending on the value of the thresholds t_k. If $t_k \geqslant 1$, each transition from a solution i to one of its neighbors $i-1$, $i+1$ is accepted, and the algorithm performs a random walk in the solution space $\mathcal{S}$. If $\delta - 1 \leqslant t_k < 1$, the algorithm only accepts transitions from a solution i to its neighbor $i-1$. Hence, in that case the algorithm performs as an optimization algorithm. If $t_k < \delta - 1$, the algorithm performs as iterative improvement and terminates in the local minimum i' that corresponds to the current solution i. From this example we observe that threshold accepting may perform very well, provided that the thresholds t_k are properly chosen. If $\delta - 1 \leqslant t_k < 1$, the algorithm finds a global optimum in $O(N)$ transitions or, assuming that n is constant, in $O(M)$ transitions. Note that this number of transitions does not depend on the number of dimensions. Consequently, provided the threshold is chosen properly, threshold accepting may do considerably better than multistart iterative improvement.

For the case that $\delta \approx 1$, simulated annealing behaves similarly to threshold accepting. This can be seen as follows. If $c \geqslant 1$, simulated annealing more or less performs a random walk, but having a slight preference for solutions with a small cost. If $\delta - 1 \leqslant c \ll 1$, the algorithm nearly only accepts transitions from a solution i to its neighbor $i-1$. In that case the algorithm also performs as an optimization algorithm, requiring $O(N)$ transitions to obtain a global optimum. If $t_k < \delta$, the algorithm probably terminates in the local minimum i' that corresponds to the current solution i, although there is a small probability of obtaining better solutions.

Next we consider the case that $\delta \gg 1$. In that case, threshold accepting cannot find on average better solutions than (single-start) iterative improvement. This can be shown as follows. If $t_k \geqslant \delta - 1$, the algorithm accepts any proposed transition. Consequently, as long as $t_k \geqslant \delta - 1$, threshold accepting performs

a random walk on the set of solutions. If $t_k < \delta - 1$, the algorithm behaves at least as bad as (single-start) iterative improvement. Let i be the current solution at that moment. Then the best obtainable solution is $i' = \lfloor i/n \rfloor i$. The algorithm even has a positive probability of terminating in a local minimum j' with $j' > i'$. Since, on average, the initial solution is identical to the solution obtained after the random walk, we conclude that, on average, threshold accepting cannot find better solutions than (single-start) iterative improvement. Hence, even if the algorithm is given enough time and the thresholds are carefully chosen, threshold accepting cannot guarantee to find a global optimum, whereas simulated annealing always has a positive probability of reaching a global optimum, when given enough time. As we show in the next section, simulated annealing asymptotically finds a global optimum with probability 1.

Summarizing, we have the following conclusions.

- For some optimization problems, the expected number of transitions necessary for reaching a global optimum is much smaller for simulated annealing and threshold accepting than for multistart iterative improvement.
- For some optimization problems, the expected cost of a final solution obtained by threshold accepting is not better than the expected cost of a final solution obtained by (single-start) iterative improvement.

Hence, the interest in simulated annealing can be motivated by the fact that, compared with multistart iterative improvement and threshold accepting, its performance is less dependent on the specific topology of the 'cost function landscape'.

4 THE PHYSICS ANALOGY

The origin of simulated annealing and the choice of the acceptance criterion lie in the physical annealing process [Kirkpatrick, Gelatt & Vecchi, 1983; Černý, 1985]. In *condensed matter physics*, *annealing* is a thermal process for obtaining low-energy states of a solid in a *heat bath*. It consists of the following two steps: (1) the temperature of the heat bath is increased to a maximum value at which the solid melts; (2) the temperature is carefully decreased until the particles of the melted solid arrange themselves in the ground state of the solid. In the liquid phase all particles of the solid arrange themselves randomly. In the ground state the particles are arranged in a highly structured lattice and the energy of the system is minimal.

The physical annealing process can be modeled successfully by computer simulation methods based on *Monte Carlo techniques*. An introductory overview of the use of these techniques in statistical physics is given by Binder [1978]. Here, we discuss one of the early techniques proposed by Metropolis et al. [1953], who gave a simple algorithm for simulating the evolution of a solid in a heat bath to *thermal equilibrium*. Their algorithm is based on Monte Carlo techniques and generates a sequence of states of the solid in the following way. Given a current state i of the solid with *energy* E_i, a subsequent state j is generated by applying a perturbation

mechanism, which transforms the current state into the next state by a small distortion, for instance by displacement of a single particle. The energy of the next state is E_j. If the *energy difference*, $E_j - E_i$, is less than or equal to 0, the state j is accepted as the current state. If the energy difference is greater than 0, the state j is accepted with a probability given by

$$\exp\left(\frac{E_i - E_j}{k_B T}\right),$$

where T denotes the *temperature* of the heat bath and k_B is a physical constant known as the *Boltzmann constant*. The acceptance rule described above is known as the *Metropolis criterion*, the corresponding algorithm as the *Metropolis algorithm*.

If the temperature is lowered sufficiently slowly, the solid can reach thermal equilibrium at each temperature. Thermal equilibrium is characterized by the *Boltzmann distribution*, which relates the probability of the solid of being in a state i with energy E_i to the temperature T, and is given by

$$\mathbb{P}_T\{\mathbf{X} = i\} = \frac{\exp(-E_i/k_B T)}{\sum_j \exp(-E_j/k_B T)},$$

where $\mathbf{X}$ is a random variable denoting the current state of the solid and the summation extends over all possible states. As we show in the following sections, the Boltzmann distribution plays an essential role in the analysis of the simulated annealing algorithm.

Returning to simulated annealing, we can apply the Metropolis criterion to generate a sequence of solutions of a combinatorial optimization problem. For this purpose we assume an analogy between a physical many-particle system and a combinatorial optimization problem based on the following equivalences:

- Solutions in a combinatorial optimization problem are equivalent to states of a physical system.
- The cost of a solution is equivalent to the energy of a state.

Next we introduce a parameter that plays the role of the temperature. This parameter is the same as the control parameter used in equation (1).

A characteristic feature of simulated annealing is that, besides accepting improvements in cost, it also accepts to a limited extent deteriorations in cost. Initially, at large values of c, large deteriorations are accepted; as c decreases, only smaller deteriorations are accepted, and as the value of c approaches 0, no deteriorations are accepted at all. Furthermore, there is no limitation on the size of a deterioration with respect to its acceptance, such as occurs in threshold accepting. In simulated annealing, arbitrarily large deteriorations are accepted with positive but small probability.

5 MARKOV MODELS

The simulated annealing algorithm can be mathematically modeled using the theory of finite Markov chains [Feller, 1950; Isaacson & Madsen, 1976; Seneta, 1981].

Definition 1 *Let $\mathcal{C}$ denote a set of possible outcomes of a sampling process. A* Markov chain *is a sequence of* trials, *where the probability of the outcome of a given trial depends only on the outcome of the previous trial. Let* $\mathbf{X}(k)$ *be a random variable denoting the outcome of the kth trial. Then the* transition probability *at the kth trial for each pair of outcomes $i, j \in \mathcal{C}$ is defined as*

$$P_{ij}(k) = \mathbb{P}\{\mathbf{X}(k) = j \mid \mathbf{X}(k-1) = i\}. \tag{2}$$

The matrix $P(k)$, whose elements are given by equation (2), is called the *transition matrix*. A Markov chain is called *finite* if the set of outcomes is finite. It is called *inhomogeneous* if the transition probabilities depend on the trial number k. If they do not depend on the trial number, the Markov chain is called *homogeneous*.

Let $a_i(k)$ denote the probability of outcome $i \in \mathcal{C}$ at the kth trial, i.e.,

$$a_i(k) = \mathbb{P}\{\mathbf{X}(k) = i\}. \tag{3}$$

Then for all $i \in \mathcal{C}$, $a_i(k)$ is given as

$$a_i(k) = \sum_{l \in \mathcal{C}} a_l(k-1) P_{li}(k).$$

Definition 2 *An n-vector x is called* stochastic *if its components x_i satisfy the conditions*

$$x_i \geqslant 0, i = 1, \ldots, n, \quad \textit{and} \quad \sum_{i=1}^{n} x_i = 1.$$

An $n \times m$ matrix X is called stochastic *if its components X_{ij} satisfy the conditions*

$$X_{ij} \geqslant 0, i = 1, \ldots, n, j = 1, \ldots, m, \quad \textit{and} \quad \sum_{j=1}^{m} X_{ij} = 1, i = 1, \ldots, n.$$

In the case of simulated annealing, a trial corresponds to a transition, and the set of outcomes is given by the finite set of solutions. Furthermore, the outcome of a trial only depends on the outcome of the previous trial. Consequently, we can safely apply the concept of finite Markov chains.

Definition 3 (transition probability) *Let $(\mathcal{S}, f)$ be an instance of a combinatorial optimization problem and $\mathcal{N}$ a neighborhood function. Then the transition probabilities for the simulated annealing algorithm are defined as*

$$\forall i, j \in \mathcal{S}: P_{ij}(k) = \begin{cases} G_{ij}(c_k) A_{ij}(c_k) & \textit{if } i \neq j, \\ 1 - \sum_{l \in S, l \neq i} G_{il}(c_k) A_{il}(c_k) & \textit{if } i = j, \end{cases} \tag{4}$$

where $G_{ij}(c_k)$ denotes the generation probability, i.e., the probability of generating a solution j from a solution i, and $A_{ij}(c_k)$ denotes the acceptance probability, i.e., the probability of accepting a solution j that is generated from solution i.

Note that the matrix P of equation (4) is stochastic. The $G_{ij}(c_k)$'s and $A_{ij}(c_k)$'s of (4) are conditional probabilities, i.e., $G_{ij}(c_k) = \mathbb{P}_{c_k}\{\text{generate } j|i\}$ and $A_{ij}(c_k) = \mathbb{P}_{c_k}\{\text{accept } j|i,j\}$. The corresponding matrices $G(c_k)$ and $A(c_k)$ are the *generation matrix* and *acceptance matrix*, respectively, and need not be stochastic.

In the original version of simulated annealing the following probabilities are used.

Definition 4 *The* generation probability *is defined by*

$$\forall i, j \in \mathscr{S}\colon\ G_{ij}(c_k) = G_{ij} = \frac{1}{\Theta}\chi_{(\mathscr{N}(i))}(j), \tag{5}$$

where $\Theta = |\mathscr{N}(i)|$, *for all* $i \in \mathscr{S}$. *The characteristic function* $\chi_{(X')}$ *for a subset* X' *of a given set* X *is a mapping of the set* X *onto the set* $\{0, 1\}$, *such that* $\chi_{(X')}(x) = 1$ *if* $x \in X'$, *and* $\chi_{(X')}(x) = 0$ *otherwise.*

The acceptance probability *is defined by*

$$\forall i, j \in \mathscr{S}\colon\ A_{ij}(c_k) = \exp\left(-\frac{(f(j) - f(i))^+}{c_k}\right), \tag{6}$$

where, for all $a \in \mathbb{R}$, $a^+ = a$ *if* $a > 0$, *and* $a^+ = 0$ *otherwise.*

Thus, the generation probabilities are chosen to be independent of the control parameter c and uniform over the neighborhoods $\mathscr{N}(i)$, where it is assumed that all neighborhoods are of equal size, i.e., $|\mathscr{N}(i)| = \Theta$, for all $i \in \mathscr{S}$.

The above definitions apply to most combinatorial optimization problems, and close examination of the literature reveals that in many practical applications these definitions – or minor variations – are indeed used. But it is also possible to formulate a set of conditions guaranteeing asymptotic convergence for a more general class of acceptance and generation probabilities. We return to this subject below.

We now concentrate on the asymptotic convergence of simulated annealing. A simulated annealing algorithm finds an optimal solution with probability 1 if, after a possibly large number of trials, say k, we have

$$\mathbb{P}\{\mathbf{X}(k) \in \mathscr{S}^*\} = 1,$$

where $\mathscr{S}^*$ denotes the set of optimal solutions. In the following sections we show that under certain conditions the simulated annealing algorithm converges asymptotically to the set of optimal solutions, i.e.,

$$\lim_{k \to \infty} \mathbb{P}\{\mathbf{X}(k) \in \mathscr{S}^*\} = 1.$$

An essential property in the study of Markov chains is that of *stationarity*. Under certain conditions on the transition probabilities associated with a Markov chain there exists a unique stationary distribution; see Feller [1950] and Isaacson & Madsen [1976].

Definition 5 (stationarity) *A* stationary distribution *of a finite homogeneous Markov chain with transition matrix* P *on a set of outcomes* $\mathscr{O}$ *is defined as the stochastic*

$|\mathcal{C}|$-vector q, whose components are given by

$$q_i = \lim_{k \to \infty} \mathbb{P}\{\mathbf{X}(k) = i \,|\, \mathbf{X}(0) = j\}, \quad \textit{for all } j \in \mathcal{C}.$$

If such a stationary distribution q exists, we have $\lim_{k \to \infty} a_i(k) = q_i$, where $a_i(k)$ is given by equation (3). Furthermore, it follows directly from the definitions that $q^T = q^T P$. Thus, q is the probability distribution of the outcomes after an infinite number of trials and the left eigenvector of P with eigenvalue 1. In the case of the simulated annealing algorithm, as P depends on c, q depends on c, i.e., $q = q(c)$.

Before we can prove the existence of a stationary distribution for the simulated annealing algorithm, we need the following definitions.

Definition 6 (irreducibility) *A Markov chain with transition matrix P on a set of outcomes $\mathcal{C}$ is* irreducible *if for each pair of outcomes $i, j \in \mathcal{C}$ there is a positive probability of reaching j from i in a finite number of trials, i.e.,*

$$\forall i, j \in \mathcal{C}\ \exists n \in \mathbb{Z}^+ : (P^n)_{ij} > 0.$$

Definition 7 (aperiodicity) *A Markov chain with transition matrix P is* aperiodic *if for each outcome $i \in \mathcal{C}$ the greatest common divisor* $\gcd(\mathcal{D}_i) = 1$, *where $\mathcal{D}_i$ is the set of all integers $n > 0$ with $(P^n)_{ii} > 0$.*

The integer gcd $(\mathcal{D}_i)$ is called the *period* of i. Thus, aperiodicity requires all solutions to have period 1. As a corollary we have that for an irreducible Markov chain aperiodicity holds if

$$\exists j \in \mathcal{C} : P_{jj} > 0. \tag{7}$$

We now come to the following important theorem; see Feller [1950] and Isaacson & Madsen [1976] for its proof.

Theorem 1 *Let P be the transition matrix associated with a finite homogeneous Markov chain on a set of outcomes $\mathcal{C}$, and let the Markov chain by irreducible and aperiodic. Then there exists a unique stationary distribution q whose components q_i are uniquely determined by*

$$\sum_{j \in \mathcal{C}} q_j P_{ji} = q_i \quad \textit{for all } i \in \mathcal{C}.$$

As a corollary we have that any probability distribution q that is associated with a finite, irreducible and aperiodic homogeneous Markov chain and that satisfies the equations

$$q_i P_{ij} = q_j P_{ji} \quad \text{for all } i, j \in \mathcal{C}, \tag{8}$$

is the unique stationary distribution in the sense mentioned in Theorem 1. The equations of (8) are called the *detailed balance equations*, and a Markov chain for which they hold is called *reversible*.

6 A HOMOGENEOUS MODEL

We now can prove asymptotic convergence of simulated annealing based on a model in which the algorithm is viewed as a sequence of Markov chains of infinite length. In this case we say that the value of the control parameter is independent of k, i.e., $c_k = c$ for all k. This leads to the following result.

Theorem 2 *Let $(\mathscr{S}, f)$ be an instance of a combinatorial optimization problem, $\mathscr{N}$ a neighborhood function, and $P(k)$ the transition matrix of the homogeneous Markov chain associated with the simulated annealing algorithm defined by* (4), (5), *and* (6), *with $c_k = c$ for all k. Furthermore, let the following condition be satisfied:*

$$\forall i, j \in \mathscr{S}\ \exists p \geqslant 1\ \exists l_0, l_1, \ldots, l_p \in \mathscr{S}$$

with

$$l_0 = i,\ l_p = j, \quad \text{and} \quad G_{l_k l_{k+1}} > 0, \quad k = 0, 1, \ldots, p-1.$$

Then the associated homogeneous Markov chain has a stationary distribution $q(c)$, whose components are given by

$$q_i(c) = \frac{\exp(-f(i)/c)}{\sum_{j \in \mathscr{S}} \exp(-f(j)/c)} \quad \text{for all } i \in \mathscr{S}, \tag{9}$$

and

$$q_i^* \stackrel{\text{def}}{=} \lim_{c \downarrow 0} q_i(c) = \frac{1}{|\mathscr{S}^*|} \chi_{(\mathscr{S}^*)}(i), \tag{10}$$

where i^* denotes an optimal solution, and $\mathscr{S}^*$ the set of optimal solutions.

The proof of the theorem follows directly from the previous results. Indeed, the condition in the theorem guarantees that the Markov chain is irreducible; see Definition 6. The transition probabilities given by (4) with (5) and (6) guarantee aperiodicity through (7); see Definition 7. Hence, according to Theorem 1, there exists a unique stationary distribution, and the correctness of the components of (9) follows directly from the detailed balance equations of (8), which proves the first part of the theorem. The second part follows directly from (9). The distribution given by (9) is the equivalent of the Boltzmann distribution in the Monte Carlo simulations of the physical annealing process mentioned in Section 4. It is characteristic of simulated annealing and, as we show below, it plays an important role in the analysis of the algorithm.

As a result of Theorem 2 we have that

$$\lim_{c \downarrow 0} \lim_{k \to \infty} \mathbb{P}_c\{\mathbf{X}(k) \in \mathscr{S}^*\} = 1. \tag{11}$$

This result reflects the basic property of the simulated annealing algorithm, i.e., the guarantee that the algorithm asymptotically finds an optimal solution. Furthermore, (11) expresses the characteristic of the homogeneous model for simulated annealing: first take the limit of the homogeneous Markov chain for an infinite number of trials, then take the limit for the control parameter to zero. In

the inhomogeneous model of Section 7 these two limits are combined into a single limit.

The convergence properties discussed above strongly depend on the original choice of the transition probabilities of (4), (5), and (6). Several authors have investigated convergence properties for a broader class of probabilities. This has led to the following formulation.

Theorem 3 *Let $(\mathscr{S}, f)$ be an instance of a combinatorial optimization problem, $\mathscr{N}$ a neighborhood function, and $P(k)$ the transition matrix of the homogeneous Markov chain associated with the simulated annealing algorithm defined by* (4), *with $c_k = c$ for all k. Furthermore, let the following conditions hold:*

(G1) $\forall c > 0\ \forall i, j \in \mathscr{S}\ \exists p \geqslant 1\ \exists l_0, l_1, \ldots, l_p \in \mathscr{S}$ *with* $l_0 = i$, $l_p = j$, *and* $G_{l_k l_{k+1}}(c) > 0, \quad k = 0, 1, \ldots, p-1$,

(G2) $\forall c > 0\ \forall i, j \in \mathscr{S}$: $\quad G_{ij}(c) = G_{ji}(c)$,

(A1) $\forall c > 0\ \forall i, j \in \mathscr{S}$: $\quad A_{ij}(c) = 1 \quad$ *iff* $f(i) \geqslant f(j)$,
$\quad A_{ij}(c) \in (0, 1)$ *iff* $f(i) < f(j)$,

(A2) $\forall c > 0\ \forall i, j, k \in \mathscr{S}$: $A_{ij}(c) A_{jk}(c) A_{ki}(c) = A_{ik}(c) A_{kj}(c) A_{ji}(c)$,

(A3) $\forall i, j \in \mathscr{S}$ *with* $f(i) < f(j)$: $\lim_{c \downarrow 0} A_{ij}(c) = 0$.

Then the Markov chain has a unique stationary distribution $q(c)$, whose components are given by

$$q_i(c) = \frac{1}{\sum_{j \in \mathscr{S}} (A_{ij}(c)/A_{ji}(c))} \quad \textit{for all } i \in \mathscr{S}, \tag{12}$$

and

$$\lim_{c \downarrow 0} q_i(c) = q_i^*,$$

where the q_i^* are defined by (10).

Condition (G1) again guarantees irreducibility and aperiodicity of the corresponding Markov chain, conditions (G2), (A1) and (A2) guarantee reversibility, and condition (A3) guarantees that stationary distributions concentrate on the set of optimal solutions as c approaches 0. In general terms, the conditions require the following. Condition (G1) requires that each solution can be reached from any other solution by generating a finite sequence of neighboring solutions. To ensure this, the corresponding neighborhood graph should be strongly connected. Condition (G2) requires symmetry of the generation matrix. Conditions (A1) through (A3) require that the acceptance matrix is well behaved, i.e., improvements are always accepted and deteriorations are accepted with positive probability for $c > 0$ (condition (A1)) and with zero probability for $\lim c \downarrow 0$ (condition (A3)); the factorization required by condition (A2) guarantees detailed balance as defined by (8).

Conditions (G1) through (A3) are sufficient but not necessary. Thus, there may be acceptance and generation matrices not satisfying these conditions and still ensuring the existence of the stationary distribution. An example of such an

acceptance matrix is

$$A_{ij}(c) = \frac{1}{1 + \exp(-(f(i) - f(j))/c)}. \tag{13}$$

This acceptance matrix does not satisfy conditions (A1) and (A2), but it can be shown to lead to the stationary distribution of (9) by using the detailed balance equations of (8).

Furthermore, several authors have addressed the generality issue of the acceptance probability. We discuss this issue following the lines of the work of Schuur [1997].

Theorem 4 *Let $P(k)$ be the transition matrix of the homogeneous Markov chain associated with the simulated annealing algorithm defined by* (4), *with $c_k = c$ for all k, and let conditions* (G1) *and* (G2) *of Theorem 3 hold. Furthermore, let the acceptance probabilities be defined as follows. Given are two functions $\phi:(0,\infty) \times \mathbb{R} \to (0,\infty)$ and $H:(0,\infty) \times \mathbb{R} \times \mathbb{R} \to (0,1]$, such that for $c > 0$, and $x, y \in \mathbb{R}: H(c,x,y) = H(c,y,x)$, and*

$$A_{ij}(c) = H(c, f(i), f(j)) \min\left(1, \frac{\phi(c, f(j))}{\phi(c, f(i))}\right),$$

and

$$\forall x, y \in \mathbb{R}: x > y \Rightarrow \lim_{c \downarrow 0} \frac{\phi(c,y)}{\phi(c,x)} = 0. \tag{14}$$

Then the Markov chain has a unique stationary distribution $q(c)$, whose components are given by

$$q_i(c) = \frac{\phi(c, f(i))}{\sum_{j \in \mathscr{S}} \phi(c, f(j))} \quad \textit{for all } i \in \mathscr{S}, \tag{15}$$

and

$$\lim_{c \downarrow 0} q_i(c) = q_i^*,$$

where the q_i^* are again given by (10).

As a corollary to Theorem 4 it is argued that the only well-behaved function $\phi(c, f(j))$ that satisfies (14) is of the form

$$\phi(c, f(j)) = \exp(\gamma(c) f(j)), \tag{16}$$

where $\gamma: (0,\infty) \to (0,\infty)$ and $\lim_{c \downarrow 0} \gamma(c) = \infty$.

Kesidis [1990] and Romeo & Sangiovanni-Vincentelli [1991] provide arguments for the assertion that the fastest convergence to the stationary distribution of (15) with (16) and $\gamma = c^{-1}$ is given by the acceptance probabilities of (6).

Below we give alternatives for conditions (G1) and (G2), respectively.

Condition (G1) can be replaced by a more general condition. This condition guarantees that the Markov chain associated with the generation matrix G is irreducible. If this is not the case, asymptotic convergence to a subset of the set of globally optimal solutions can still be proved if condition (G1) is replaced by the following necessary and sufficient condition:

$$(G1') \quad \forall i \in \mathscr{S}\ \exists i^* \in \mathscr{S}^*,\ p \geqslant 1\ \exists l_0, l_1, \ldots l_p \in \mathscr{S}$$
$$\text{with } l_0 = i,\ l_p = i^*, \text{ and } G_{l_k l_{k+1}} > 0, \quad k = 0, 1, \ldots, p-1. \tag{17}$$

According to this condition it should be possible to construct a finite sequence of transitions with nonzero generation probability, leading from an arbitrary solution i to some optimal solution i^*. For the proof of the validity of this condition, a distinction must be made between *transient* and *recurrent* solutions, where a solution is called transient if the probability that the Markov chain ever returns to that solution equals zero, and recurrent if the Markov chain may return to the solution with a positive probability [Feller, 1950]. Furthermore, the stationary distribution of (12) does not apply any more and should be replaced by a *stationary matrix* $Q(c)$ whose elements q_{ij} denote the probability of finding a solution j after an infinite number of transitions, starting from a solution i. A more detailed treatment is beyond the scope of this chapter; the reader is referred to Connors & Kumar [1987], Gidas [1985], Van Laarhoven [1988], and Van Laarhoven, Aarts & Lenstra [1992].

In practice one does not want to bother about the requirement of condition (G2) that the generation matrix must be symmetric. Easier to implement is a uniform distribution over the neighborhoods, similar to that used in the original version of simulated annealing. Lundy & Mees [1986] show that for the choice of the generation probabilities given by

$$G_{ij} = \frac{1}{|\mathscr{N}(i)|}\chi_{\mathscr{N}(i)}(j) \quad \text{for all } i, j \in \mathscr{S}, \tag{18}$$

condition (G2) is no longer needed to guarantee asymptotic convergence, and the components of the stationary distribution are then given by

$$q_i(c) = \frac{|\mathscr{N}(i)|}{\sum_{j \in \mathscr{S}} |\mathscr{N}(j)| A_{ij}(c)/A_{ji}(c)}.$$

Moreover, it follows directly that these components again converge to the q_i^* of (10) as $c \downarrow 0$. Finally, we mention that a generation matrix satisfying both condition (G2) and (18) implies that $|\mathscr{N}(i)|$ is independent of i.

7 AN INHOMOGENEOUS MODEL

In the previous section it was shown that, under certain conditions on the generation and acceptance matrices, the simulated annealing algorithm converges to a global minimum with probability 1, if for each value of the control parameter c_k, $k = 0, 1, 2, \ldots$, the corresponding homogeneous Markov chain is

infinitely long and the sequence $(c_k | k = 0, 1, 2, \ldots)$ eventually converges to 0 as $k \to \infty$. In this section we discuss conditions to guarantee asymptotic convergence for the case where each Markov chain is of finite length. Thus, the simulated annealing algorithm is modeled as an inhomogeneous Markov chain with the following transition probabilities:

$$P_{ij}(c_k) = \begin{cases} G_{ij}(c_k) A_{ij}(c_k) & j \neq i, \\ 1 - \sum_{l \in \mathscr{S}, l \neq i} G_{il}(c_k) A_{il}(c_k) & j = i. \end{cases} \tag{19}$$

Furthermore, we assume that the sequence $(c_k | k = 0, 1, 2, \ldots)$ satisfies the conditions

$$\lim_{k \to \infty} c_k = 0, \text{ and} \tag{20}$$

$$c_k \geqslant c_{k+1}, \; k = 0, 1, \ldots. \tag{21}$$

Thus, c_k is kept constant during a number of transitions, in which case we obtain an inhomogeneous Markov chain consisting of an infinite number of homogeneous Markov chains of finite length each.

We show that, under certain conditions on the rate of convergence of the sequence $(c_k | k = 0, 1, 2, \ldots)$, the inhomogeneous Markov chain associated with the simulated annealing algorithm converges in distribution to q^*, whose components are given by (10). In other words we prove that

$$\lim_{k \to \infty} \mathbb{P}\{\mathbf{X}(k) \in \mathscr{S}^*\} = 1.$$

To discuss the convergence of inhomogeneous Markov chains we need the following definitions; see Seneta [1981].

Definition 8 *Let $P(k)$ be the transition matrix associated with an inhomogeneous Markov chain on a set of outcome $\mathscr{O}$. Then the matrix $U(m, k)$ is defined as*

$$U(m, k) = \prod_{n=m}^{k} P(n), \quad 0 < m \leqslant k.$$

In other words the components of $U(m, k)$ are equal to

$$U_{ij}(m, k) = \mathbb{P}\{\mathbf{X}(k) = j | \mathbf{X}(m-1) = i\} \quad \textit{for all } i, j \in \mathscr{O}.$$

Definition 9 (ergodicy) *A finite inhomogeneous Markov chain on a set of outcomes $\mathscr{O}$ is* weakly ergodic *if*

$$\forall i, j, l \in \mathscr{O}, \forall m > 0: \lim_{k \to \infty} (U_{il}(m, k) - U_{jl}(m, k)) = 0.$$

It is strongly ergodic *if there exists a stochastic vector q^* such that*

$$\forall i, j \in \mathscr{O}, \forall m > 0: \lim_{k \to \infty} U_{ij}(m, k) = q_j^*.$$

Thus, for a given m, weak ergodicity implies that $\mathbf{X}(k)$ becomes independent of

$\mathbf{X}(m)$ as $k \to \infty$, whereas strong ergodicity implies *convergence in distribution*, i.e., for any stochastic vector $a(m)$ denoting the probabilities of the outcomes of the mth trial we have that

$$\lim_{k \to \infty} a^T(m-1) \prod_{n=m}^{k} P(n) = (q^*)^T,$$

or

$$\lim_{k \to \infty} \mathbb{P}\{\mathbf{X}(k) = j\} = \lim_{k \to \infty} \left(\sum_{i \in \mathcal{C}} U_{ij}(m,k)\, \mathbb{P}\{\mathbf{X}(m-1) = i\} \right) = q_j^* \quad \text{for all } i,j \in \mathcal{C}.$$

The difference between weak and strong ergodicity can be understood from the following example. Let the transition probabilities $P_{ij}(k)$ of an inhomogeneous Markov chain be independent of j. Then the Markov chain is clearly weakly ergodic, but it is not strongly ergodic if the $P_{ij}(k)$ vary forever with k for a given i. Note that for a homogeneous Markov chain there is no distinction between weak and strong ergodicity.

The following two theorems provide conditions for weak and strong ergodicity of inhomogeneous Markov chains. The proofs can be found in Isaacson & Madsen [1976] and Seneta [1981].

Theorem 5 *Let $\tau_1(X)$ denote the coefficient of ergodicity of the $n \times n$ stochastic matrix X defined as*

$$\begin{aligned} \tau_1(X) &= \frac{1}{2} \max_{i,j=1,\ldots,n} \sum_{l=1}^{n} |X_{il} - X_{jl}| \\ &= 1 - \min_{i,j=1,\ldots,n} \sum_{l=1}^{n} \min(X_{il}, X_{jl}). \end{aligned}$$

Then an inhomogeneous Markov chain is weakly ergodic if and only if there is a strictly increasing sequence of positive numbers $(k_i \mid i = 0, 1, 2, \ldots)$ such that

$$\sum_{i=0}^{\infty} (1 - \tau_1(X(k_i, k_{i+1}))) = \infty. \tag{22}$$

Theorem 6 *A finite inhomogeneous Markov chain is strongly ergodic under the following conditions:*

(C1) *The Markov chain is weakly ergodic.*

(C2) *For all k there exists a stochastic vector $q(k)$ such that $q(k)$ is the left eigenvector of $P(k)$ with eigenvalue 1.*

(C3) *The eigenvectors $q(k)$ satisfy*

$$\sum_{k=1}^{\infty} \| q(k) - q(k+1) \|_1 < \infty, \tag{23}$$

where the 1-norm of an n-vector x is defined as $\| x \|_1 = \sum_{i=1}^{n} |x_i|$.

Moreover, if $q^* = \lim_{k \to \infty} q(k)$, then q^* is the vector of Definition 9.

To prove convergence in distribution for the simulated annealing algorithm it must be shown that the associated inhomogeneous Markov chain is strongly ergodic.

Theorem 7 *Let $(\mathscr{S}, f)$ be an instance of a combinatorial optimization problem, $\mathscr{N}$ a neighborhood function, and $P(k)$ the transition matrix of the inhomogeneous Markov chain associated with the simulated annealing algorithm defined by* (19), (5), *and* (6). *Furthermore, let the following conditions be satisfied:*

(D1) $$\forall i,j \in \mathscr{S}\ \exists p \geqslant 1\ \exists l_0, l_1, \ldots, l_p \in \mathscr{S}$$

$$\text{with } l_0 = i,\ l_p = j, \quad \text{and } G_{l_k l_{k+1}} > 0, \quad k = 0, 1, \ldots, p-1.$$

(D2) $$c_k \geqslant \frac{\Gamma}{\log(k + k_0)}, \quad k = 0, 1, \ldots, \tag{24}$$

for some value of $\Gamma > 0$ and $k_0 > 2$.

Then the Markov chain converges in distribution to the vector q^*, with components given by (10), *or in other words*

$$\lim_{k \to \infty} \mathbb{P}\{\mathbf{X}(k) \in \mathscr{S}^*\} = 1. \tag{25}$$

Theorem 7 can be proved by showing that conditions (D1) and (D2) are sufficient to satisfy conditions (C1), (C2), and (C3) of Theorem 5, along the following lines.

Condition (D1) guarantees the existence of the left eigenvector $q(k)$ of $P(k)$, given by $q(k) = q(c_k)$, i.e., the stationary distribution of the homogeneous Markov chain with transition matrix $P = P(k)$; see Theorem 2. The components of the eigenvectors are given by (9). Furthermore, from (20) and (9) we have $\lim_{k \to \infty} q(c_k) = \lim_{c \downarrow 0} q(c) = q^*$, where the components of q^* are given by (10). So condition (C2) of Theorem 6 holds and, by using the explicit form of the eigenvectors $q(k)$, condition (C3) can be shown to hold. What remains to be shown is that the Markov chain is weakly ergodic, as stipulated by condition (C1). This can be done by using Theorem 5 and condition (D2).

The latter proof is quite technical, and several authors have come up with different approaches, which vary predominantly in the estimates of the values of the constant Γ.

One of the first results was obtained by Mitra, Romeo & Sangiovanni-Vincentelli [1986]. To discuss this we need the following definition.

Definition 10 *The* distance *$d(i,j)$ between two solutions $i,j \in \mathscr{S}$ is defined as the length d of the shortest sequence of solutions $(l_0, l_1, \ldots, l_d)$, with $l_0 = i$, $l_d = j$, and $P_{l_m l_{m+1}}(k) > 0$, $l_m \in \mathscr{S}$, $m = 0, 1, \ldots, d-1$.*

Mitra, Romeo & Sangiovanni-Vincentelli [1986] found that $\Gamma \geqslant r\Delta$ with

$$\Delta = \max_{i \in \mathscr{S}} \max_{j \in \mathscr{N}(i)} \{|f(j) - f(i)|\}, \tag{26}$$

and

$$r = \min_{i \in \mathscr{S} \setminus \hat{\mathscr{S}}} \max_{j \in \mathscr{S}} d(i,j), \tag{27}$$

where $\hat{\mathscr{S}}$ denotes the set of all locally minimal solutions.

Other values of Γ are given by Anily & Federgruen [1987a], Gelfand & Mitter [1985], Geman & Geman [1984], Gidas [1995], and Holley & Stroock [1988]; for an overview see Romeo & Sangiovanni-Vincentelli [1991].

The conditions for asymptotic convergence given above are *sufficient* but not *necessary*. Necessary and sufficient conditions are derived by Hajek [1988]. The difference with the conditions presented above again lies in the difference in value of the constant Γ. To discuss Hajek's result we need the following definitions.

Definition 11 *Let $i, j \in \mathscr{S}$, then j is* reachable *at height h from i if $i = j$ and $f(i) \leqslant h$, or $\exists p \geqslant 1\ \exists l_0, \ldots, l_p \in \mathscr{S}$ with $l_0 = i$ and $l_p = j$ such that $G_{l_k, l_{k+1}} > 0$ and $f(l_k) \leqslant h$ for all $k = 0, \ldots, p-1$.*

Definition 12 *Let $\hat{\imath}$ be a local minimum. Then the* depth *$d(\hat{\imath})$ of $\hat{\imath}$ is the smallest number x, $x > 0$, such that there is a solution $j \in \mathscr{S}$ with $f(j) < f(\hat{\imath})$ that is reachable at height $f(\hat{\imath}) + x$ from $\hat{\imath}$. By definition, for an optimal solution i^*, $d(i^*) = \infty$.*

We now can formulate the results obtained by Hajeck.

Theorem 8 *Let $(c_k | k = 0, 1, \ldots)$ be a sequence of values of the control parameter defined as*

$$c_k = \frac{\Gamma}{\log(k+2)}, \quad k = 0, 1, \ldots,$$

for some constant Γ. Then asymptotic convergence of the simulated annealing algorithm, using the transition probabilities of (19), (5), *and* (6), *is guaranteed if and only if*

- *the Markov chain is irreducible,*
- *i is reachable from j at height h if and only if j is reachable from i at height h, for arbitrary $i, j \in \mathscr{S}$ and h, and*
- *the constant Γ satisfies $\Gamma \geqslant D$, where*

$$D = \max_{\hat{\imath} \in \hat{\mathscr{S}} \setminus \mathscr{S}^*} d(\hat{\imath}), \tag{28}$$

i.e., D is the depth of the deepest local, nonglobal minimum.

Kern [1993] has addressed the problem of calculating the value of D. In particular, he showed for a number of problems how it is unlikely that D can be calculated in polynomial time for arbitrary instances of a combinatorial optimization problem. He also presents bounds on the value of D for several combinatorial optimization problems.

Under certain conditions, asymptotic convergence of the inhomogeneous Markov chain associated with the simulated annealing algorithm can also be

proved for general conditions on the generation and acceptance probabilities. This result was first proved by Anily & Federgruen [1987b] and can be formulated as follows.

Theorem 9 *Let the transition probabilities of the inhomogeneous Markov chain associated with the simulated annealing algorithm be defined by* (19), *and let the generation probabilities* $G_{ij}(c)$ *and acceptance probabilities* $A_{ij}(c)$ *satisfy conditions* (G1) *through* (A3) *of Theorem 3. Furthermore, let*

$$\underline{A}(c) = \min_{i,j} \{ A_{ij}(c) \mid i \in \mathscr{S}, j \in \mathscr{N}(i) \}.$$

Then

$$\lim_{k \to \infty} \mathbb{P} \{ \mathbf{X}(k) \in \mathscr{S}^* \} = 1,$$

if

$$\sum_{k=0}^{\infty} (\underline{A}(c_k))^n = \infty, \tag{29}$$

where n denotes the maximum number of steps needed to reach an optimal solution from any arbitrary solution, and the constant Γ *satisfies*

$$\Gamma \geqslant n\Delta, \tag{30}$$

where Δ *is given by* (26).

Finally, we mention that Gelfand & Mitter [1985] derived sufficient conditions for convergence to an arbitrary set of solutions. These conditions are similar to those given above.

8 ASYMPTOTIC BEHAVIOR

We have shown that, under mild conditions, the simulated annealing algorithm converges in probability to the set of optimal solutions, or in other words, asymptotically the algorithm finds an optimal solution with probability 1. As a result of the limits of (11) or (25), asymptotic convergence to the set of optimal solutions is achieved only after an infinite number of transitions. In any finite-time implementation one must resort to approximations of the asymptotic convergence.

With respect to the approximation of the stationary distribution we have the following two properties; see Seneta [1981].

Property 10 *Let* $P(k)$ *denote the transition matrix of the homogeneous Markov chain associated with the simulated annealing algorithm defined by* (4), *and let* $q(c)$ *denote the corresponding stationary distribution given by the left eigenvector with eigenvalue* 1 *of* P. *Then, as* $k \to \infty$, *we have*

$$\| a(k) - q(c) \|_1 = O(k^s |\lambda_2(c)|^k), \tag{31}$$

where $a(k)$ denotes the probability distribution of the outcomes after k trials, $\lambda_2(c)$ $(0 < |\lambda_2(c)| < 1)$ denotes the second largest eigenvalue of $P(k)$ with multiplicity m_2, and $s = m_2 - 1$.

Hence, the speed of convergence to the stationary distribution is determined by $\lambda_2(c)$. Unfortunately, computation of $\lambda_2(c)$ is impracticable, due to the large size of matrix $P(k)$. Approximation of the norm in (31) leads to the following property; see Aarts & Van Laarhoven [1985b].

Property 11 *Let ε denote an arbitrarily small positive number. Then*

$$\| a(k) - q(c) \|_1 < \varepsilon, \tag{32}$$

if

$$k > K\left(1 + \frac{\ln(\varepsilon/2)}{\ln(1 - \gamma^K(c))}\right), \tag{33}$$

where $\gamma(c) = \min_{i,j \in \mathscr{S}} P_{ij}^{+}(c)$ and $K = |\mathscr{S}|^2 - 3|\mathscr{S}| + 3$.

Hence, (32) and (33) indicate that the stationary distribution is approximated arbitrarily closely, only if the number of transitions is at least quadratic in the size of the solution space. Moreover, the size $|\mathscr{S}|$ is for most problems exponential in the size of the problem itself; for instance, in the n-city traveling salesman problem, $|\mathscr{S}| = (n-1)!$. Thus, the analysis presented above indicates that approximating the stationary distribution arbitrarily closely results in an exponential-time execution of the simulated annealing algorithm.

With respect to the asymptotic convergence of the inhomogeneous Markov chain associated with the simulated annealing algorithm, we have the following result; see Mitra, Romeo & Sangiovanni-Vincentelli [1986].

Property 12 *Let the transition probabilities of the inhomogeneous Markov chain associated with the simulated annealing algorithm be defined by* (19), (5), *and* (6), *and let the sequence $(c_k | k = 0, 1, \ldots)$ be given by* (24), *with $\Gamma > r\Delta$, where r and Δ are defined as in* (26) *and* (27). *Furthermore, let q^* be the uniform probability distribution on the set of optimal solutions defined by* (10). *Then for $k \to \infty$,*

$$\| a(k) - q^* \|_1 < \varepsilon,$$

for an arbitrarily small positive number ε, if

$$k = O(\varepsilon^{-\max(a,b)}),$$

where

$$a = \frac{r^{r\Delta/\Gamma}}{w^r}, \quad \text{and} \quad b = \frac{r\Delta}{\hat{f} - f^*},$$

with $\hat{f} = \min_{i \in \mathscr{S}/\mathscr{S}^*} f(i)$ and $w = \min_{i \in \mathscr{S}} \min_{j \in \mathscr{N}(i)} G_{ij}$.

Evaluation of this bound for particular problems typically leads to a number of transitions that is larger than the size of the solution space and thus

to an exponential-time execution for most problems. For instance, in the case of the traveling salesman problem, Aarts & Korst [1989a] show that

$$k = O(n^{n^{2n}}).$$

Note that $|\mathscr{S}| = (n-1)!$ Hence, complete enumeration of all solutions would take less time than approximating an optimal solution arbitrarily closely by the simulated annealing algorithm.

Summarizing, we have shown that optimal simulated annealing algorithms require an infinite number of transitions and that the rate of convergence is logarithmic, leading to exponential time complexities for arbitrarily close approximation of an optimal solution.

Several authors have investigated possibilities of speeding up the convergence of optimal simulated annealing for specific problems by taking into account the combinatorial structure of the problem at hand. For instance, Sorkin [1991] proved that, if the neighborhoods of a problem exhibit certain fractal properties, the time complexity of optimal simulated annealing is polynomial. More specifically, he showed that, for problems with properly scaled cost functions between 0 and 1, and a fractal neighborhood structure, a solution of expected cost no greater than ε can be found in a time bounded by a polynomial in $1/\varepsilon$, where the exponent of the polynomial depends on the fractal.

Stander & Silverman [1994] discuss a simple global optimization problem and propose an optimal method for lowering the value of the control parameter based on the use of dynamic programming techniques. The resulting time complexity is still exponential but the method provides optimal choices for the initial and final values of the control parameter. Christoph & Hoffmann [1993] address the scaling behavior of optimal annealing. They found that *dominating barriers* exist at which the value of the control parameter must be lowered more slowly than in between the barriers.

Rajasekaran & Reif [1992] obtained improved convergence rate of optimal annealing by exploiting a special property of the cost function, if present, which they call *small-separability*. Based on this concept, they developed an algorithm called *nested annealing*, which is a simple modification of the classical simulated annealing algorithm obtained by assigning different control parameter values to different regions. For a specific class of problems in computer vision and circuit layout, they proved that the time complexity of their optimal simulated algorithm is $2^{O(\sqrt{n})}$ instead of $2^{\Omega(n)}$, where n refers to the size of the problem instance at hand.

9 COOLING SCHEDULES

We now leave the issue of optimal annealing and turn to finite-time implementations of the algorithm. Earlier sections have indicated that finite-time implementations can no longer guarantee to find an optimal solution, but may result in much faster executions of the algorithm without significantly compromising the solution quality. Ingber [1993] refers to such implementations as *simulated quenching*.

A *finite-time* implementation of the simulated annealing algorithm is obtained by generating a sequence of homogeneous Markov chains of finite length at descending values of the control parameter. For this, a set of parameters must be specified that govern the convergence of the algorithm. These parameters are combined in what is called a cooling schedule.

Definition 13 *A* cooling schedule *specifies a finite sequence of values of the control parameter, and a finite number of transitions at each value of the control parameter. More precisely, it is specified by*

- *an* initial *value of the control parameter* c_0,
- *a* decrement function *for lowering the value of the control parameter*,
- *a* final value *of the control parameter specified by a stop criterion*,
- *a* finite length *of each homogeneous Markov chain.*

Central to the discussion of cooling schedules is the concept of quasi-equilibrium. Let L_k denote the length of the kth Markov chain and c_k the corresponding value of the control parameter. Then *quasi-equilibrium* is achieved if the probability distribution $a(L_k, c_k)$ of the solutions, after L_k trials of the kth Markov chain, is 'sufficiently close' to the stationary distribution at c_k, i.e.,

$$\| a(L_k, c_k) - q(c_k) \|_1 < \varepsilon, \tag{34}$$

for some specified positive value of ε.

From Property 11 we recall that a number of transitions is required that is quadratic in the size of the solution space in order to satisfy (34) for arbitrarily small values of ε, which leads to an exponential-time execution for most problems. Thus, in order to be of practical use, a less rigid quantification of the quasi-equilibrium concept is needed than that of (34). For this one may resort to the following arguments. For the acceptance probabilities of (6), and well-behaved generation probabilities, the stationary distribution is of the form given by (9). For $c \to \infty$, the stationary distribution is given by a uniform distribution on the set of solutions $\mathscr{S}$, i.e., if

$$\lim_{c \to \infty} q_i(c) = \frac{1}{|\mathscr{S}|}. \tag{35}$$

Thus, at sufficiently large values of c_k – allowing acceptance of virtually all proposed transitions – quasi-equilibrium is obtained by definition, since all solutions occur with equal probability given by the uniform distribution of (35). Next, the decrement function and the Markov chain lengths must be chosen such that quasi-equilibrium is restored at the end of each individual Markov chain. In this way the equilibrium distributions for the various Markov chains are 'closely followed', so as to arrive eventually, as $c_k \downarrow 0$, close to q^*, the uniform distribution on the set of optimal solutions given by (10).

It is intuitively clear that large decrements in c_k require larger Markov chain lengths in order to restore quasi-equilibrium at the next value c_{k+1} of the control parameter. Thus, there is a trade-off between large decrements of the control

parameter and small Markov chain lengths. Usually, one chooses small decrements, in c_k to avoid extremely long chains, but one could use large values for L_k in order to be able to make large decrements in c_k.

The search for adequate cooling schedules has been the subject of many studies over the past year. Reviews are given by Van Laarhoven & Aarts [1987], Collins, Eglese & Golden [1988], and Romeo & Sangiovanni-Vincentelli [1991].

9.1 Optimal schedules

Recently, researchers have been investigating optimal finite-time schedules, where optimal refers to the best average cost obtained in finite time. Strenski & Kirkpatrick [1991] analyzed a small instance of a graph partitioning problem and used an approach based on evaluating exactly the probability distributions of outcomes of the Markov chain associated with the simulated annealing algorithm. They found that different schedules, including iterative improvement, may be optimal depending on the employed schedule length. When a sufficiently long schedule is employed, annealing replaces iterative improvement as the optimal schedule. Furthermore, they observed that optimal schedules may be nonmonotone. This result was rather unexpected since the convergence proofs of simulated annealing suggest a monotone lowering of the control parameter value; see for instance Aarts & Korst [1989a]. Nevertheless, it was in accordance with earlier theoretical results obtained by Hajek & Sasaki [1989], who found for a small artificial problem that the control parameter values of an optimal annealing schedule are all either 0 or ∞.

The approach of Strenski & Kirkpatrick [1991] has been further pursued by Boese & Kahng [1994]. They introduce the concept of *best-so-far* versus *where-you-are*. More specifically, they use an acceptance criterion based on the cost of the best solution found so far, instead of the cost of the current solution. They determine optimal cooling schedules for two small instances of the traveling salesman problem and the graph partitioning problem and found that optimal sequences of control parameter values may not be monotone. The analysis of optimal finite-time schedules is interesting, but the results obtained so far are only proved to hold for extremely small instances. At present it is not clear which impact they have on larger instances. One might argue that the whimsical structure of small instances may introduce artifacts that are absent from the more regularly structured large instances. In that case the nonmonotonicity results would only hold for a specific class of small problem instances.

9.2 Heuristic schedules

Most of the existing work on cooling schedules presented in the literature deals with heuristic schedules. We distinguish between two broad classes: static schedules and dynamic schedules. In a *static* cooling schedule the parameters are fixed: they cannot be changed during execution of the algorithm. In a *dynamic*

cooling schedule the parameters are adaptively changed during execution of the algorithm. Below we present some examples.

Static cooling schedules

The following simple schedule is known as the *geometric schedule*. It originates from the early work on cooling schedules by Kirkpatrick, Gelatt & Vecchi [1983], and is still used in many practical situations.

Initial value of the control parameter. To ensure a sufficiently large value of c_0, one may choose $c_0 = \Delta f_{\max}$, where $\Delta f_{\max}$ is the maximal difference in cost between any two neighboring solutions. Exact calculation of $\Delta f_{\max}$ is quite time-consuming in many cases. However, one often can give simple estimates of its value.

Lowering the control parameter value. A frequently used decrement function is given by

$$c_{k+1} = \alpha \cdot c_k, \; k = 0, 1, \ldots,$$

where α is a positive constant smaller than but close to 1. Typical values lie between 0.8 and 0.99.

Final value of the control parameter. The final value is fixed at some small value, which may be related to the smallest possible difference in cost between two neighboring solutions.

Markov chain length. The length of Markov chains is fixed by some number that may be related to the size of the neighborhoods in the problem instance at hand.

Dynamic cooling schedules

There exist many extensions of the simple static schedule presented above that lead to a dynamic schedule. For instance, a sufficiently large value of c_0 may be obtained by requiring that the *initial acceptance ratio* χ_0 – defined as the number of accepted transitions at c_0 – is close to 1. This can be achieved by starting off at a small positive value of c_0 and multiplying it with a constant factor, larger than 1, until the corresponding value of χ_0, which is calculated from a number of generated transitions, is close to 1. Typical values of χ_0 lie between 0.9 and 0.99.

An adaptive calculation of the final value of the control parameter may be obtained by terminating the execution of the algorithm at a c_k value for which the value of the cost function of the solution obtained in the last trial of a Markov chain remains unchanged for a number of consecutive chains. Clearly such a value exists for each local minimum that is found. The length of a Markov chain may be determined by requiring that at each value c_k a minimum number of transitions is accepted. However, since transitions are accepted with decreasing

probability, one would obtain $L_k \to \infty$ for $c_k \downarrow 0$. Therefore, L_k is usually bounded by some constant $L_{\max}$ to avoid extremely long Markov chains for small values of c_k.

In addition to this basic dynamic schedule, the literature presents a number of more elaborate schedules. Most of them are based on a statistical analysis of the simulated annealing process, thus allowing a more theoretical estimation of the parameters. For the transition probabilities of (4), (5), and (6), the statistical analysis leads to a model for the cost distribution that resembles an exponential distribution at low c values and a normal distribution at high c values. Within this model, the first two moments of the resulting distribution are given by Aarts, Korst & Van Laarhoven [1988]:

$$\mathbb{E}_c(f) = \mathbb{E}_\infty(f) - \frac{\sigma_\infty^2(f)}{c}\left(\frac{\gamma c}{\gamma c + 1}\right) \tag{36}$$

and

$$\sigma_c^2(f) = \sigma_\infty^2(f)\left(\frac{\gamma c}{\gamma c + 1}\right)^2,$$

where γ is given by

$$\gamma = \frac{\mathbb{E}_\infty(f) - f^*}{\sigma_\infty^2(f)}.$$

To compute $\mathbb{E}_c(f)$ and $\sigma_c^2(f)$, values for $\mathbb{E}_\infty(f)$ and $\sigma_\infty^2(f)$ can be approximated by the average cost value of the solutions and the corresponding standard deviation, respectively.

The analysis given above is used by several authors to derive adaptive parameter estimates. As an example, we discuss the schedule proposed by Huang, Romeo & Sangiovanni-Vincentelli [1986], since it is quoted in the literature as the most efficient one among those that require only a modicum of sophistication. The schedule of Lam & Delosme [1986] is conjectured to be even more efficient but its intricacy generally hinders practical use.

Initial value of the control parameter. From (36) it follows directly that $\mathbb{E}_c(f) \approx \mathbb{E}_\infty(f)$ for $c \gg \sigma_\infty^2(f)$. Hence c_0 may be chosen as

$$c_0 = K\,\sigma_\infty^2(f),$$

where K is a constant typically ranging from 5 to 10.

Lowering the control parameter value. Here the concept of quasi-equilibrium is quantified by requiring that the average cost difference for two consecutive Markov chains is small, i.e., $\mathbb{E}_{c_{k+1}}(f) - \mathbb{E}_{c_k}(f) = -\varepsilon$ for some small positive number ε. Next, by using

$$\frac{\partial}{\partial \ln c}\mathbb{E}_c(f) = \frac{\sigma_c^2(f)}{c}, \tag{37}$$

and replacing the left-hand side of (37) with the differential quotient, we obtain

$$\frac{\mathbb{E}_{c_{k+1}}(f) - \mathbb{E}_{c_k}(f)}{\ln c_{k+1} - \ln c_k} = \frac{\sigma^2_{c_k}(f)}{c_k}.$$

This results in a decrement rule given by

$$c_{k+1} = c_k \exp\left(-\frac{\varepsilon c^*}{\sigma^2_{c_k}(f)}\right), \tag{38}$$

where, for practical purposes, $\sigma_{c_k}(f)$ is approximated by the measured deviation. In their original paper, Huang, Romeo & Sangiovanni-Vincentelli [1986] replace ε by $\lambda\sigma_{c_k}$, $\lambda < 1$, which gives only a slight modification of (38).

Final value of the control parameter. Execution is terminated if at the end of a Markov chain

$$f'_{\max} - f'_{\min} = \Delta f'_{\max}, \tag{39}$$

where $f'_{\max}$ and $f'_{\min}$ denote the maximum and minimum cost value, respectively, and $\Delta f'_{\max}$ the maximum cost difference of the solutions accepted during the generation of that chain. If (39) holds, c is set to 0, and the execution is concluded with a simple local search to ensure local optimality of the final solution.

Markov chain length. Statistical analysis leads to the observation that, in equilibrium, the fraction of solutions generated with cost values within a certain range ε from the expected cost reaches a stationary value κ. Assuming a normal distribution of the cost values, Huang, Romeo & Sangiovanni-Vincentelli [1986] show that $\kappa = -\operatorname{erf}(\varepsilon/\sigma_c(f))$, where $\operatorname{erf}(x)$ is the *error function*; see Abramowitz & Stegun [1970]. The Markov chain length is determined by the number of trials L_k for which

$$L_k^* = p\kappa,$$

where p is a parameter depending on the size of the problem instance, and L_k^* is defined as the number of accepted solutions with a cost value within the interval $(\mathbb{E}_c - \varepsilon, \mathbb{E}_c + \varepsilon)$. An additional bound on L_k is introduced to avoid extremely long Markov chains.

10 ISSUES FROM PRACTICE

Four basic ingredients are needed to apply simulated annealing in practice: a concise problem representation, a neighborhood function, a transition mechanism, and a cooling schedule. The algorithm is usually implemented as a sequence of homogeneous Markov chains of finite length, generated at descending values of the control parameter specified by the cooling schedule. As for the choice of the cooling schedule, we have seen in the previous section that there exist some general guidelines. However, no general rules are known that guide the choice of

the other ingredients. The way they are handled is still a matter of experience, taste, and skill left to the annealing practitioner, and we expect that this will not change in the near future.

During the years of its existence, simulated annealing has been applied to a large variety of problems, ranging from practical real-life situations to theoretical tests. Two appealing examples of real-life applications are the scheduling of the Australian state cricket season by Willis & Terrill [1994] and the design of keyboards for typewriters by Light & Anderson [1993]. VLSI design, atomic and molecular physics, and picture processing are the three problem areas in which simulated annealing is most frequently applied. The set of theoretical test problems includes almost all the well-known problems in discrete mathematics and operations research, such as coding, graph coloring, graph partitioning, sequencing and scheduling problems; see the references given below and Chapters 8 through 13 of this book.

So far, about a thousand papers have been published reporting applications of the algorithm. Many of these studies have led to modifications of the algorithm such as the use of penalty functions, alternative generation and acceptance probabilities, implementation-specific aspects, parallel versions, etc. Due to the large variety of approaches and the many different implementation details, it is virtually impossible to give a balanced overview of the experience that has been gathered. Therefore, we restrict ourselves to some general statements and appropriate references.

We start with the references. General overviews of applications of simulated annealing are given by Aarts & Korst [1989a], Collins, Eglese & Golden [1988], Dowsland [1993], Van Laarhoven & Aarts [1987], and Vidal [1993]. Overviews of applications in operations research are given by Eglese [1990] and Koulamas, Antony & Jaen [1994]. Studies emphasizing performance issues for theoretical test problems are given by several authors. One of the most elaborate studies is presented by Johnson et al. [1989, 1991], who report on an extensive numerical study for several combinatorial optimization problems, including graph partitioning, graph coloring and number partitioning. This work provides many practical findings that in our opinion reflect the general experience of annealing practitioners.

Perhaps the most striking element is the observed performance ambivalence. For the graph partitioning problem, simulated annealing seems to outperform all existing approximation algorithms, whereas for the number partitioning problem the performance is hopelessly poor. Although this bad performance for the number partitioning problem can be understood from analytical arguments, there seems no way to adapt the algorithm in order to improve it. A similar ambivalence is encountered in the area of code design, where for the football pool problem, simulated annealing is able to improve on the best known results; see Van Laarhoven et al. [1989] and Chapter 13 of this book. On the other hand, Beenker, Claasen & Hermens [1985] found that for problems related to the design of binary sequences, the algorithm is inferior to simple constructive methods.

Furthermore, the literature presents results of studies in which the performance of simulated annealing is compared with that of other local search algorithms. Results for the job shop scheduling problem are presented by Van Laarhoven, Aarts & Lenstra [1992], Aarts et al. [1994], and Vaessens, Aarts & Lenstra [1996], and for the traveling salesman problem by Ulder et al. [1991].

With some restraint one may conclude from these studies that simulated annealing, if large running times are allowed, can outperform about all other algorithms with respect to effectiveness. More general conclusions cannot be drawn due to the many different quality measures that can be applied. To illustrate this we mention the performance studies for quadratic assignment. Pardalos, Murty & Harrison [1993] report that simulated annealing can find acceptable solutions with fewer iterations than tabu search. Battiti & Tecchiolli [1994] question this conclusion and argue that it no longer holds for difficult problem instances if high quality solutions are required.

Broadly speaking, simulated annealing can find good solutions for a wide variety of problems, but often at the cost of substantial running times. Consequently, the true merits of the algorithm become obvious in industrial problem settings, where running times are of little or no concern. As an example we mention design problems, since in those cases one is primarily interested in finding high-quality solutions, whereas design time often plays only a minor role. A well-known successful simulated annealing area in this respect is VLSI design [Sechen & Sangiovanni-Vincentelli, 1985; Shahookar & Mazumder, 1991; Wong, Leong & Liu, 1988].

The success of simulated annealing can be explained from the fact that the algorithm is easy to implement and capable of handling almost any optimization problem and any constraint, either by appropriate neighborhoods or by relaxation through the use of penalty functions. These properties are, however, not unique for simulated annealing. They also hold for simple local search algorithms. The main advantage of simulated annealing is that it is able to improve upon the relatively poor performance of local search by simply replacing the deterministic (strict improvement) acceptance criterion by a stochastic criterion, thus circumventing the need of an in-depth study of the problem structure in order to construct more effective neighborhoods, or to design more tailored algorithms. It almost goes without saying how this is a great advantage in an industrial environment, since often the required expertise is unavailable and, even more important, it saves development time.

11 SPEEDING UP

The literature presents many variations on the basic simulated annealing approach presented in the previous sections. Many of these variations concentrate on alternatives that should reduce the potentially burdensome running times required by simulated annealing to converge to near-optimal solutions. Roughly speaking, the existing approaches fit into three categories: fast sequential algo-

rithms, hardware acceleration, and parallel algorithms. We mention a few examples.

Szu & Hartley [1987] present an annealing algorithm for the optimization of continuous-valued functions, using a generation mechanism given by a Cauchy distribution instead of the frequently used Gaussian distribution. They claim that their generation mechanism leads to an inverse linear cooling rate, instead of an inverse logarithmic cooling rate as was found for the Gaussian distribution. This approach has been further refined by Ingber [1989], who proposes a technique he calls *very fast simulated reannealing*, permitting an exponential cooling rate. The application of these approaches is limited to the optimization of continuous real-valued functions, which prohibits their use in the many existing combinatorial optimization problems.

Greene & Supowit [1986] introduce the *rejectionless method* as an example of a deterministic simulated annealing approach based on an improved generation mechanism. They propose to generate new solutions with a probability proportional to the effect of a transition on the cost function. In this way, a subsequent solution is directly chosen from the neighborhood of a given solution, i.e., no rejection of solutions takes place. This method leads to shorter Markov chains for a number of problems. However, the efficient use of the method depends strongly on some additional conditions on the neighborhood function, which unfortunately cannot be met by many combinatorial optimization problems. Fox [1993, 1994] further elaborates on this issue. He introduces the concept of *self-loop elimination* and shows how it not only speeds up simulated annealing, but also causes the algorithm to be more efficient than multistart iterative improvement with random restarts. This contradicts Ferreira & Žerovnik [1993], who asserted the opposite.

Parallel simulated annealing algorithms aim at distributing the execution of the various parts of a simulated annealing algorithm over a number of communicating parallel processors. This is a promising approach to the problem of speeding up the execution of the algorithm, but it is by no means a trivial task, due to the intrinsic sequential nature of the algorithm. Over the years a large variety of approaches have been proposed, leading to algorithms that are generally applicable and to tailored algorithms. For overviews we refer to Aarts & Korst [1989a], Azencott [1992], Boissin & Lutton [1993], Greening [1990], and Verhoeven & Aarts [1996]. A special approach to parallel simulated annealing is provided by the use of neural network models. To this end, the optimization problem at hand is cast into a 0–1 programming formulation and the values of the decision variables are associated with the states of the neurons in the network. This has led to randomized approaches such as the *Boltzmann machine* [Aarts & Korst 1989a, 1989b; Aarts & Korst, 1991], and to deterministic approaches such as the *mean field method* [Peterson & Söderberg, 1989]; see also Chapter 7 of this book. In addition to the speedup obtained by parallel execution, neural networks also offer a speedup through their hardware implementation. This has led to fast VLSI implementations of simulated annealing [Lee & Sheu, 1991] and even to optical implementations [Lalanne et al., 1993].

12 COMBINED APPROACHES

Recent approaches to local search concentrate on the combined use of different local search algorithms, known as *multilevel approaches* [Vaessens, Aarts & Lenstra, 1992]. Simulated annealing is used in several of these approaches, and we mention some examples. Martin, Otto & Felten [1991, 1992] propose a successful simulated annealing algorithm for the traveling salesman problem, which uses a restricted 4-exchange neighborhood, combined with a simple local search algorithm using a 3-exchange neighborhood. Eiben, Aarts & Van Hee [1991] present a stochastic search procedure that combines elements of population genetics with those of simulated annealing. They prove that their stochastic approach exhibits convergence properties similar to those of simulated annealing. Lin, Kao & Hsu [1994] introduce a genetic approach to simulated annealing using population-based transitions, genetic-operator based quasi-equilibrium control, and Metropolis-criterion selection operations in the jargon of genetic algorithms. They find empirically that their approach works quite well for the zero–one knapsack, set partitioning, and traveling salesman problems.

Clearly, the issue of combined approaches opens many possibilities for the design of new variants of local search algorithms. However, one should be careful not to propose these variants as new algorithmic concepts. Research on local search has been fascinating over the past 10 years. It has also suffered from considerable confusion, created by so-called new concepts, which, after their fancy names had been demystified, turned out to be only coarse or well-known heuristic rules.

Local Search in Combinatorial Optimization
Edited by E. Aarts and J. K. Lenstra
© 1997 John Wiley & Sons Ltd.

5

Tabu search

Alain Hertz
Swiss Federal Institute of Technology, Lausanne

Eric Taillard
IDSIA, Lugano

Dominique de Werra
Swiss Federal Institute of Technology, Lausanne

1 INTRODUCTION 121
2 BASIC IDEAS OF TABU SEARCH 123
3 EFFICIENCY OF ITERATIVE SOLUTION METHODS 127
3.1 Effective modeling 127
3.2 Effective computing 128
4 EFFICIENT USE OF MEMORY 129
4.1 Variable tabu list size 129
4.2 Intensification of the search 130
4.3 Diversification 130
5 SOME APPLICATIONS OF TABU SEARCH 131
5.1 The quadratic assignment problem 131
5.2 The graph coloring problem 133
5.3 The maximum independent set problem 133
5.4 The vehicle routing problem 134
5.5 The course scheduling problem 135
6 CONCLUSIONS 136

1 INTRODUCTION

Engineering and technology have been continuously providing examples of difficult optimization problems. In this chapter we shall present the tabu search technique, which with its various ingredients may be viewed as an engineer-designed approach: no clean proof of convergence is known but the technique has shown a remarkable efficiency on many problems.

The roots of tabu search go back to the 1970s. It was first presented in its current form by Glover [1986]; the basic ideas have also been sketched by Hansen [1986]. Additional efforts of formalization are reported by Glover [1989, 1990] and de Werra & Hertz [1989]. Many computational experiments have shown that tabu search has now become an established approximation

technique, which can compete with almost all known techniques and which, by its flexibility, can beat many classical procedures. Up to now, there has been no formal explanation of this good behavior. Recently, theoretical aspects of tabu search have been investigated [Faigle & Kern, 1992; Glover, 1992; Fox, 1993].

Faigle & Kern [1992] have proposed a probabilistic version of tabu search that converges almost surely to a global optimum. They showed that one can choose probabilities of moving from a current solution to a neighbor in a very wide range without losing the probabilistic convergence properties. This probabilistic model is in fact a refined simulated annealing procedure, in which the introduction of a memory in the search process has the effect of biasing the probability of moving from one solution to another. More precisely, they consider a simulated annealing scheme where, at each temperature c, the probabilities of considering j as a potential successor of i are given by a stochastic matrix $A(c) = (a_{ij}(c))$ having the property that there exists an ε such that for each $c > 0$, $a_{ij}(c) > 0$ implies $a_{ij}(c) \geqslant \varepsilon$ whenever $i \neq j$.

Once a solution j has been chosen as a potential successor of a current solution i, it is accepted as the new current solution with probability $b_{ij}(c)$. The transition matrix $P(c) = (p_{ij}(c))$ is then defined by

$$p_{ij}(c) = a_{ij}(c) b_{ij}(c) \quad \forall i, j, i \neq j.$$

Let $G(A(c))$ be the graph of transitions defined by taking an arc (i, j) whenever $a_{ij}(c) > 0$, and let f be the objective function to be minimized. It is assumed that for all temperatures c the graphs $G(A(c))$ are the same, and that for any real value u the connected components of the graph $G(A(c))$ restricted to the solutions i for which $f(i) < u$ are strongly connected.

Denoting the stationary distribution of $P(c)$ by $\pi(c)$ and the limit distribution of $\pi(c)$ as $c \to 0$ by π^*, it is shown that under the above assumptions $\pi_i^* > 0$ only if i is an optimal solution.

Since tabu search can be viewed as a method where at each step the neighborhood of a solution varies (see later), we may consider that this amounts to changing some of the generation probabilities $a_{ij}(c)$. These changes may also take into account information gained in previous iterations. In fact, the result of Faigle & Kern [1992] shows that convergence properties are kept as long as the probabilities are modified within certain bounds $[\varepsilon, 1 - \varepsilon]$.

Besides these properties, which may be helpful from a computational point of view (as long as a probabilistic version of the algorithm is used), they also observed that deep valleys may be lifted in order to escape from local minima. Indeed when replacing the value $f(i)$ of a recognized local minimum i by the smallest value $f(j)$ of a neighbor j of i, we remove the local minimum without affecting the conditions on $G(A(c))$ necessary to ensure convergence.

Several pieces of work are based on this type of result. They are in fact slight modifications of simultated annealing rather than extensions of tabu search [Fox, 1993]. The meaning of the convergence result should nevertheless be recalled: 'The result does not say that the limit of the sequence of solutions generated by the algorithm is optimal, as some people mistakenly believe, but

only that the method will find itself *at an optimum* an increasing percentage of the time, relative to being elsewhere, as time grows. This requires that the process repeatedly, and increasingly, executes a self-loop that returns to the same solution encountered on the previous iteration' [Glover, 1992].

A didactic presentation of tabu search and a series of applications have been collected in a recent book [Glover et al., 1993]. Its interest lies in the fact that success with tabu search often implies that a serious effort of modeling be made from the beginning. The applications in Glover et al. [1993] provide many such examples together with a collection of references.

A huge collection of optimization techniques have been suggested by a crowd of researchers from different fields, and an infinity of refinements have made these techniques work on specific types of applications. All these procedures are based on some common ideas and, furthermore, are characterized by a few specific extras. Among the optimization procedures, the iterative techniques play an important role; for most optimization problems no general procedure is known to obtain an 'optimal' solution directly.

The general step of an iterative procedure consists of constructing from a current solution i a next solution j and checking whether one should stop there or perform another step. Neighborhood search methods are iterative procedures in which a neighborhood $\mathcal{N}(i)$ is defined for each feasible solution i, and the next solution j is searched among the solutions in $\mathcal{N}(i)$.

The most famous neighborhood search method that has been used for finding an approximation to the minimum value of a real-valued function f on a set $\mathcal{S}$ is the descent method.

Descent method

1. Choose an initial solution $i \in \mathcal{S}$.
2. Find a best $j \in \mathcal{N}(i)$, i.e., such that $f(j) \leqslant f(k)$ for any $k \in \mathcal{N}(i)$.
3. If $f(j) \geqslant f(i)$, then stop. Else set $i = j$ and go to Step 2.

Such a method clearly may stop at a local but not global minimum of f. In general, $\mathcal{N}(i)$ is not defined explicitly: we may search for j by exploring some directions from i (e.g., the coordinate axes).

Simulated annealing and tabu search can be considered as neighborhood search methods that are more elaborate than the descent method. The basic ingredients of tabu search are described in the next section.

2 BASIC IDEAS OF TABU SEARCH

In order to improve the efficiency of the exploration process, one needs to keep track not only of local information (like the current value of the obective function) but also of some information related to the exploration process. This systematic use of *memory* is an essential feature of tabu search (TS). Its role will be emphasized later on. Although most exploration methods keep in memory essentially the value $f(i^*)$ of the best solution i^* visited so far, TS will also keep

information on the itinerary through the last solutions visited. Such information will be used to guide the move from i to the next solution j to be chosen in $\mathcal{N}(i)$. The role of the memory will be to restrict the choice to some subset of $\mathcal{N}(i)$ by forbidding for instance moves to some neighbor solutions.

More precisely, we will see that the structure of the neighborhood $\mathcal{N}(i)$ of a solution i varies from iteration to iteration. It may therefore be more appropriate to include TS in a class of procedures called *dynamic neighborhood search techniques*.

Formally let us consider an optimization problem in the following way: given a set $\mathcal{S}$ of feasible solutions and a function $f: \mathcal{S} \to \mathbb{R}$, find some solution i^* in $\mathcal{S}$ such that $f(i^*)$ is acceptable with respect to some criterion (or criteria). Generally a criterion of acceptability for a solution i^* would be to have $f(i^*) \leqslant f(i)$ for every i in $\mathcal{S}$. Then TS would be an exact minimization algorithm provided the exploration process were to guarantee that, after a finite number of steps, such an i^* would be reached.

In most contexts, however, no guarantee can be given that such an i^* will be obtained so TS could be viewed simply as a very general heuristic procedure. Since TS will include in its own operating rules some heuristic techniques, it may be characterized as a *metaheuristic*. Its role will most often be to guide and to orient the search of another (more local) search procedure.

As a first step towards the description of TS, we reformulate the classical descent method as follows:

1. Choose an initial solution $i \in \mathcal{S}$.
2. Generate a subset V^* of solutions in $\mathcal{N}(i)$.
3. Find a best $j \in V^*$, i.e., such that $f(j) \leqslant f(k)$ for any $k \in V^*$, and set $i = j$.
4. If $f(j) \geqslant f(i)$, then stop. Else go to Step 2.

In a straightforward descent method we generally take $V^* = \mathcal{N}(i)$. However, this may often be too time-consuming; an appropriate choice of V^* may often be a substantial improvement.

The opposite case is to take $|V^*| = 1$; this drops the step of choosing a best j. Simulated annealing can be integrated within such a framework. A solution j will be accepted if $f(j) \leqslant f(i)$, otherwise it will be accepted with a certain probability depending upon the value of f at i and j as well as upon a parameter called temperature.

There is no temperature in TS. However, the choice of V^* is crucial. In order to define it at each step, one uses memory systematically to exploit knowledge extending beyond the function f and the neighborhood $\mathcal{N}(i)$.

Except for some special cases of convexity, the use of descent procedures is generally frustrating. This is because we are likely to be trapped in a local minimum that may be far from a global minimum (with respect to the value of f).

So any iterative exploration process should sometimes accept nonimproving moves from i to $j \in V^*$, i.e., with $f(j) > f(i)$, if one wants to escape from a local minimum. Simulated annealing does this also, but it does not guide the choice of j. TS in contrast chooses a best $j \in V^*$.

As soon as nonimproving moves are possible, there is the risk of visiting a solution again and more generally of cycling. It is then that the use of memory helps to forbid moves that might lead to recently visited solutions. If such a memory is introduced, we may consider that the structure of $\mathcal{N}(i)$ will depend upon the itinerary and hence upon the iteration k; so we may refer to $\mathcal{N}(i, k)$ instead of $\mathcal{N}(i)$. With these modifications in mind, we may attempt to formalize an improvement of the descent algorithm in a way that will bring it closer to the general TS procedure. Writing i^* for the best solution found so far and k for the iteration counter it can be stated as follows:

1. Choose an initial solution $i \in \mathcal{S}$. Set $i^* = i$ and $k = 0$.
2. Set $k = k + 1$ and generate a subset V^* of solutions in $\mathcal{N}(i, k)$.
3. Choose a best $j \in V^*$ with respect to f or to some modified function $\tilde{f}$, and set $i = j$.
4. If $f(i) < f(i^*)$, then set $i^* = i$.
5. If a stopping condition is met, then stop. Else go to Step 2.

Observe that the classical descent procedure is included in this formulation: the stopping rule is simply $f(i) \geqslant f(i^*)$ and i^* is always the last solution. Notice also that we may consider the use of a modified $\tilde{f}$ instead of f in some circumstances to be described later.

Some of the immediate stopping conditions are as follows:

- $\mathcal{N}(i, k+1) = \varnothing$.
- k is larger than the maximum number of iterations allowed.
- The number of iterations since the last improvement of i^* is larger than a specified number.
- Evidence can be given that an optimum solution has been obtained.

Although these stopping rules may have some influence on the search procedure and on its results, the crucial factors are the definition of $\mathcal{N}(i, k)$ at each iteration k and the choice of V^*.

The definition of $\mathcal{N}(i, k)$ implies that some recently visited solutions are removed from $\mathcal{N}(i)$. They are considered as *tabu solutions*, which should be avoided in the next iteration. Such a memory based on recency will partially prevent cycling. For instance, keeping at iteration k a list T (tabu list) of the last $|T|$ solutions visited will prevent cycles of length at most $|T|$. Then we may take $\mathcal{N}(i, k) = \mathcal{N}(i) - T$. However, this list T may be extremely impractical to use. We will therefore describe the exploration process in $\mathcal{S}$ in terms of moves from one solution to the next. For each solution $i \in \mathcal{S}$, we define $M(i)$ as the set of moves m that can be applied to i in order to obtain a new solution j; notation $j = i \oplus m$. Then $\mathcal{N}(i) = \{j \mid \exists m \in M(i) \text{ with } j = i \oplus m\}$. In general we use moves that are reversible: for each m there exists a move m^{-1} such that $(i \oplus m) \oplus m^{-1} = i$. So instead of keeping a list T of the last $|T|$ solutions visited, we may simply keep track of the last $|T|$ moves or of the last $|T|$ reverse moves associated with the moves actually performed. It is clear that this restriction is a loss of information and that it does not guarantee that no cycle of length at most $|T|$ will occur.

For efficiency, it may be convenient to use several lists T_r at a time. Then some constituents t_r (of i or of m) will be given a tabu status to indicate that these constituents are currently not allowed to be involved in a move. Generally the tabu status of a move is a function of the tabu status of its constituents, which may change at each iteration. So we may formulate a collection of *tabu conditions* as follows:

$$t_r(i, m) \in T_r, \quad r = 1, \ldots, t.$$

A move m applied to a solution i will be a tabu move if *all* conditions are satisfied. The next few sections give examples to illustrate these concepts.

Another drawback of the simplification of the tabu list, due to replacing solutions by moves, is the fact that we may be led to giving a tabu status to solutions that may be unvisited so far. We are then compelled to some relaxation of the tabu status; we will overrule the tabu status when some tabu solutions look attractive. This will be performed by *aspiration level conditions.*

A tabu move m applied to a current solution i may appear attractive because it gives a solution better than the best found so far. We would like to accept m in spite of its status; we shall do so if it has an *aspiration level* $a(i, m)$ that is better than a threshold value $A(i, m)$.

Generally $A(i, m)$ can be viewed as a set of preferred values for the function $a(i, m)$. So conditions of aspiration can be written in the form

$$a_r(i, m) \in A_r(i, m), \quad r = 1, \ldots, a.$$

If at least one of these conditions is satisfied by the tabu move m applied to i, move m will be accepted in spite of its status. Illustrations are given below.

We have now described almost all basic ingredients of TS, but there is one more important feature. Since f may in some instances be replaced by another function $\tilde{f}$, we may introduce some intensification and diversification of the search.

Memory is used in different ways to guide the tabu search procedure. We have seen a short-term memory, whose role was to forbid some moves likely to drive us back to recently visited solutions. Memory is also present at a deeper level.

It is sometimes fruitful to intensify the search in some region of $\mathscr{S}$ because it may contain some acceptable solutions. Such *intensification* can be carried out by giving a high priority to solutions having features in common with the current solution. This may be achieved by introducing an extra term in the objective function, which will penalize solutions far from the current solution. Intensification should be carried out over a few iterations. Then it may be useful to explore another region of $\mathscr{S}$. This *diversification* will tend to spread the exploration effort over different regions of $\mathscr{S}$, and it may be forced by introducing a penalization term into the objective function. At some stage this term will penalize solutions that are close to the current solution. The intensification and diversification terms have weights that are modified throughout the search so that the process alternates from one to the other. The modified objective function is $\tilde{f}$, mentioned earlier in the algorithm: $\tilde{f} = f$ + intensification + diversification.

We have now briefly presented the essential characteristics of a TS procedure, which can be stated as follows:

Tabu search

1. Choose an initial solution $i \in \mathcal{S}$. Set $i^* = i$ and $k = 0$.
2. Set $k = k + 1$ and generate a subset V^* of solutions in $\mathcal{N}(i, k)$ such that either one of the tabu conditions $t_r(i, m) \in T_r$ is violated ($r = 1, \ldots, t$) or at least one of the aspiration conditions $a_r(i, m) \in A_r(i, m)$ holds ($r = 1, \ldots, a$).
3. Choose a best $j = i \oplus m \in V^*$ with respect to f or to the function $\tilde{f}$, and set $i = j$.
4. If $f(i) < f(i^*)$, then set $i^* = i$.
5. Update the tabu and aspiration conditions.
6. If a stopping condition is met, then stop. Else go to Step 2.

3 EFFICIENCY OF ITERATIVE SOLUTION METHODS

The efficiency of iterative solution methods depends mostly on the modeling. A fine tuning of parameters will never balance a bad choice of the neighborhood structure or of the objective function. On the other hand, an effective modeling should lead to robust techniques that are not too sensitive to different parameter settings. In this section we will give some general guidelines for designing efficient iterative solution methods.

3.1 Effective modeling

Iterative solution methods may be viewed as walks in a neighborhood graph $G(\mathcal{S}, A)$ where the vertex set $\mathcal{S}$ is the set of feasible solutions and there is an arc $(i, j) \in A$ from i to j if $j \in \mathcal{N}(i)$. Choosing an initial solution is equivalent to generating a vertex in G, and a step of the iterative procedure consists of moving from the current vertex i to a vertex adjacent to i. For a given optimization problem there are usually many ways to define the neighborhood graph G, and it is essential to choose one that satisfies the following condition:

$$\text{For any } i \in \mathcal{S}, \text{ there exists a path from } i \text{ to an optimal solution } i^*. \qquad (1)$$

If a solution that does not satisfy this condition is visited during the iterative process, an optimal solution will never be reached.

To illustrate this, let us consider the graph coloring problem: given a graph $H = (V, E)$ one has to find a coloring of its vertices with as few colors as possible such that no two vertices linked by an edge have the same color; such colorings are called feasible colorings. One can define the set $\mathcal{S}$ of feasible solutions as the set of feasible colorings that do not use more than a given number of colors; for example, this upper bound may be the number of colors used in the initial solution. Let us define the neighborhood $\mathcal{N}(i)$ of a solution i as the set of feasible colorings that can be obtained from i by changing the color of exactly one vertex. This induces a neighborhood graph that does not satisfy condition (1). Indeed,

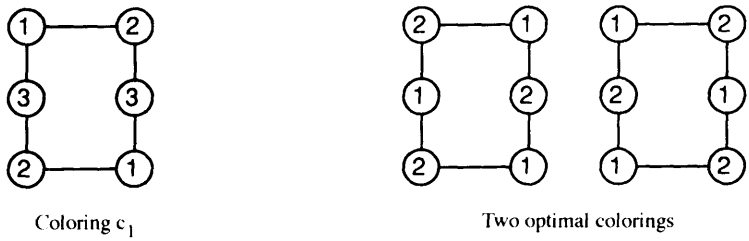

Figure 5.1 The graph coloring problem

consider the example in Figure 5.1. If the iterative solution method visits the feasible coloring c_1 and if the upper bound on the number of colors that may be used is 3, then an optimal coloring using only 2 colors will never be reached. A better definition of the neighborhood graph G will be given in Section 5.

Now note that the set $\mathscr{S}$ of feasible solutions is not necessarily the set of solutions of the original problem; sometimes it may be very difficult to define a neighborhood structure that satisfies condition (1). If the solutions of the original problem must satisfy a set C of constraints, it is sometimes judicious to define $\mathscr{S}$ as a set of solutions that satisfy a proper subset $C' \subset C$ of constraints. For the graph coloring problem, an efficient adaptation of TS will be described in Section 5, where the constraint that two vertices linked by an edge should receive different colors is relaxed. Of course, the violation of a relaxed constraint is penalized in the objective function.

Each vertex i of the neighborhood graph G has a value $f(i)$. Considering these values as altitudes, this induces a topology with valleys containing local optima. It is important to define f in such a way that there are not too many valleys or large plateaus. Too many valleys or large plateaus make it very difficult to guide the search towards an optimal solution, which is at the bottom of one of the deepest valleys.

For the graph coloring problem, if one defines the value $f(i)$ of a coloring i as the number of colors that are used in i, the consequence is to generate very large plateaus. This is because all solutions have an integer value that belongs to the small interval $[f(i^*), u]$, where $f(i^*)$ is the value of an optimal solution and u is the upper bound.

As mentioned in Section 2, the objective function f may be changed during the search. These changes correspond to a modification of the topology. During the diversification phase, for example, one tries to replace the valleys often visited by high mountains; hence the search is directed towards unexplored regions.

3.2 Effective computing

At each step of an iterative solution method, many solutions have to be evaluated, so it is important to perform this computation efficiently. For a solution i and a move m, it is often easier to compute $\Delta(i, m)$ defined as $f(i \oplus m) - f(i)$ than

$f(i \oplus m)$. Sometimes it is possible to store the values $\Delta(i, m)$ for all $m \in M(i)$. Then if a move m may be applied to two consecutive solutions i and j, i.e., $m \in M(i) \cap M(j)$, it is often possible to compute $\Delta(j, m)$ even more easily. In Section 5 we will describe an adaptation of TS to the quadratic assignment problem: given a solution i and a move $m \in M(i)$, the direct computation of $f(i \oplus m)$ needs time $O(n^2)$ where n is the size of the problem, whereas $\Delta(i, m)$ can be evaluated in time $O(n)$. Moreover, for two consecutive solutions i and j, if all values $\Delta(i, m)$ with $m \in M(i)$ have been stored, the values $\Delta(j, m)$ with $m \in M(j)$ can be computed in time $O(1)$ on average.

For some problems, given a solution i and a move $m \in M(i)$, it is difficult to compute $f(i \oplus m)$. For the vehicle routing problem (VRP), let us define a solution i as an assignment of the customers to the vehicles, and a move m as the transfer of exactly one customer from one vehicle to another. If the objective is to minimize the total distance traveled, then computing $f(i \oplus m)$ is equivalent to solving several traveling salesman problems (TSPs). It is well known that the TSP is an NP-hard problem.

In such cases it is necessary to use fast heuristic methods for evaluating the neighboring solutions. Once a best neighboring solution has been determined, a more elaborate technique may be used to improve it. For example, for the VRP, a simple insertion method may be used for finding the best neighbor $j \in N(i)$. All routes of i that have been modified for getting j are then reoptimized using improvement methods such as described by Lin & Kernighan [1973], or even exact methods when there are not too many customers on the routes.

When it is possible to store all the values $\Delta(i, m)$ for $m \in M(i)$, it is sometimes possible to keep a sorted list of these values with a small computational effort.

In Section 5 we will describe an adaptation of TS to the problem of finding an independent set of given size k in a graph with n vertices. The $O(nk)$ neighbors will be efficiently evaluated by storing two sorted lists containing pertinent information. One step of TS, i.e., finding the best neighbor and updating the sorted lists, can then be performed in time $O(n)$.

4 EFFICIENT USE OF MEMORY

Section 2 presented a general description of the TS procedure, where memory is an essential feature. The tabu conditions may usually be considered as a short-term memory that prevents cycling to some extent. This section describes some efficient policies for the management of tabu lists. We shall also show how the use of memory may help to intensify the search in 'good' regions or to diversify the search towards unexplored regions.

4.1 Variable tabu list size

We have seen that it may be convenient to use several tabu lists at a time. Although we shall restrict the following discussion to a unique tabu list, it may be extended to the general case.

The basic role of the tabu list is to prevent cycling. A list that is too short may not prevent cycling, but a list that is too long may create excessive restrictions; the mean value of the visited solutions appears to grow with an increase of the tabu list size. It is usually easy to determine the order of magnitude of the list size. But, for a given optimization problem, it is often difficult or even impossible to find a value that prevents cycling and does not excessively restrict the search for all instances of a given size.

An effective way of circumventing this difficulty is to vary the size of the tabu list. Each element of the list belongs to it for a number of iterations bounded by given maximum and minimum values.

4.2 Intensification of the search

In order to intensify the search in promising regions, we first have to come back to one of the best solutions found so far. Then the size of the tabu list may be simply decreased for a small number of iterations.

More elaborate techniques may sometimes be used. Some optimization problems can be partitioned into subproblems. Solving these subproblems optimally and combining the partial solutions leads to an optimal solution. The obvious difficulty is to find a good partition. For the VRP, any solution induces a partition of the customers into a certain number of routes. The search can then be intensified by solving subproblems that focus on subsets of routes.

For reasons related to computation time, fast heuristics and a neighborhood of reasonable size are used at each step of TS. Ways of intensifying the search use more elaborate heuristics, even exact methods, or they enlarge the neighborhood. For the VRP, the simple insertion technique used at each iteration may be replaced by the 2-opt technique or an exact solution method for the TSP. If a neighboring solution is obtained by moving a customer to a new route containing at least one of its p nearest neighbors (where p is a parameter), then p can be increased during the intensification phase.

It is also possible to perform an intensification based on long-term memory. Each solution or move can be characterized by a set of components. The components of good moves, or good solutions, are memorized. During the intensification phase the moves or solutions are evaluated, taking into account the amounts of their good components. For example, the best-visited solutions of a VRP might have some common feature, such as pairs of customers that are always on the same route. During the intensification phase moves that separate such pairs of customers are penalized, whereas those that put them together are favored. This long-term memory may be viewed as a kind of learning process.

4.3 Diversification

It is important to diversify the search, in order to avoid a large region of the neighborhood graph remaining completely unexplored. The simplest way is to

perform several random restarts. A different way to guarantee the exploration of unvisited regions is to penalize moves frequently performed or solutions visited often.

This penalty is set large enough to ensure escape from the current region. The modified objective function is used for a given number of iterations. As described in Section 5 for the VRP, it is also possible to use a penalty on frequently performed moves during the whole search procedure.

When the feasible solutions have to satisfy a set of constraints, these appear as unpassable mountains of infinite height. Relaxing these constraints and penalizing their violation corresponds to reducing the height of the mountains. It is then possible to pass this barrier and reach another valley in a few steps. During this diversification phase, the solutions visited are not necessarily feasible, since constraints have been relaxed. To obtain a feasible solution again, the penalty for the violation of the relaxed constraints is gradually increased.

5 SOME APPLICATIONS OF TABU SEARCH

Tabu search has been applied to many combinatorial optimization problems. Some examples may illustrate the ingredients of TS and indicate practical implementations. More details on these applications and references to other works may be found in Hertz & de Werra [1990] and Glover et al. [1993].

5.1 The quadratic assignment problem

We first present an adaptation of TS to the quadratic assignment problem [Taillard, 1991]. This application uses only basic concepts of TS and is easy to implement. Given n items (e.g., buildings) and flows a_{ij} from item i to item j, $i, j = 1, \ldots, n$ (e.g., the flow of people from building i to building j), given n locations (e.g., sites where buildings may be constructed) with distances b_{uv} between locations u and v, $u, v = 1, \ldots, n$, one searches to assign one location to each item such that the sum of the products of the flows and the distances is minimized. We have to determine a permutation π^* that minimizes

$$f(\pi) = \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij} b_{\pi(i)\pi(j)},$$

where $\pi(i)$ gives the ith component of the permutation π, i.e., the location of item i.

For this problem we define the set $\mathcal{S}$ of feasible solutions as the set of permutations of n items, and the neighborhood $\mathcal{N}(\pi)$ of a solution π as the set of permutations that can be obtained from π by exchanging two elements of π. A move m is entirely defined by both locations whose items are exchanged. Here the set of moves does not depend on the current solution.

The tabu status of a move is defined as follows: we choose to forbid the moves that place both concerned items back in locations they have already occupied in

the last $|T|$ iterations. We use the following aspiration conditions. The first one is classical and allows a tabu move to be performed if it improves the best solution π^* found so far. The second condition is designed to diversify the search; if a move has never been chosen during a large number A of iterations, it is performed, whatever is the value of the solution it leads to:

$A_1(\pi, m) = \{x \in \mathbb{R}, x < f(\pi^*)\}$,
$A_2(\pi, m) = \{m' \in M(\pi), m'$ has not been chosen during the A last iterations$\}$,
$a_1(\pi, m) = f(\pi \oplus m), \qquad a_2(\pi, m) = m.$

The best solution $\pi' \in V^*$ at a step of the search will be chosen considering the modified function

$$\tilde{f}(\pi \oplus m) = \begin{cases} f(\pi \oplus m) & \text{if } m \notin A_2(\pi, m), \\ -\infty & \text{if } m \in A_2(\pi, m). \end{cases}$$

In order to implement the tabu and aspiration conditions efficiently, an array T of size $n \times n$ is used whose elements T_{ik} correspond to the iteration number at which unit i has been moved to location k.

A move m_{ij} that exchanges the locations of units i and j is tabu if it satisfies the following two conditions:

$$T_{i\pi(j)} + \text{tabu list size} \geqslant \text{current iteration},$$
$$T_{j\pi(j)} + \text{tabu list size} \geqslant \text{current iteration}.$$

The tabu status of a move m_{ij} is canceled if the following aspiration condition is satisfied:

$$f(\pi \oplus m) \leqslant f(\pi^*) \text{ or}$$
$$\max\{T_{i\pi(j)} + A,\ T_{j\pi(i)} + A\} \leqslant \text{current iteration}.$$

Hence, testing whether or not a solution belongs to V^* can be performed in $O(1)$ time by using an additional memory, which is not greater than the memory needed to store the data of the problem.

Let m_{ij} be the move that swaps the items located on sites i and j, and let us denote by $\Delta(\pi, i, j)$ the value of the move m_{ij} applied to solution π, i.e., $\Delta(\pi, i, j) = f(\pi \oplus m_{ij}) - f(\pi)$. A simple calculation shows that

$$\begin{aligned}\Delta(\pi, i, j) = \sum_{k=1}^{n} \Big(& a_{ki}(b_{\pi(k)\pi(j)} - b_{\pi(k)\pi(i)}) + a_{kj}(b_{\pi(k)\pi(i)} - b_{\pi(k)\pi(j)}) \\ & + a_{ik}(b_{\pi(j)\pi(k)} - b_{\pi(i)\pi(k)}) + a_{jk}(b_{\pi(i)\pi(k)} - b_{\pi(j)\pi(k)}) \Big) \\ & + \Big(a_{ii}(b_{\pi(j)\pi(j)} - b_{\pi(i)\pi(i)}) + a_{ij}(b_{\pi(j)\pi(i)} - b_{\pi(i)\pi(j)}) \\ & + a_{ji}(b_{\pi(i)\pi(j)} - b_{\pi(j)\pi(i)}) + a_{jj}(b_{\pi(i)\pi(i)} - b_{\pi(j)\pi(j)}) \Big).\end{aligned}$$

Moreover, if $\pi' = \pi \oplus m_{uv}$, it is easy to see that for $i \neq u, v$ and $j \neq u, v$ we have

$$\Delta(\pi', i, j) = \Delta(\pi, i, j) + (a_{ui} - a_{uj} + a_{vj} - a_{vi})(b_{\pi'(v)\pi'(i)} - b_{\pi'(v)\pi'(j)} + b_{\pi'(u)\pi'(j)} - b_{\pi'(u)\pi'(i)})$$
$$+ (a_{iu} - a_{ju} + a_{jv} - a_{iv})(b_{\pi'(i)\pi'(v)} - b_{\pi'(j)\pi'(v)} + b_{\pi'(j)\pi'(u)} - b_{\pi'(i)\pi'(u)}).$$

Hence, by storing the values $\Delta(\pi, i, j)$ of all possible moves m_{ij} from the current solution π, one can perform each iteration of the tabu search procedure in time $O(n^2)$.

A set of parameters has been identified that works satisfactorily on a wide range of problems. The tabu list size is randomly and uniformly chosen in the interval $[\lfloor 0.9n \rfloor, \lceil 1.1n + 4 \rceil]$ and the aspiration parameter A is set equal to $5n^2$.

This adaptation of tabu search is not only one of today's most efficient approximation procedures for the quadratic assignment problem (QAP), it is also very easy to implement, needing only two pages of Pascal code. A more refined adaptation of tabu search to the QAP is described by Battiti & Tecchiolli [1992], in which each visited solution is stored in order to determine the tabu conditions. For some instances of the QAP, this new adaptation turns out to be more efficient than the version we have described, but it is more difficult to implement.

5.2 The graph coloring problem

Given a graph $G = (V, E)$ we have to find a coloring of its vertices with as few colors as possible [Hertz & De Werra, 1987]. The unique constraint is that two vertices linked by an edge should not receive the same color. In other words, a coloring of the vertices of G in k colors is a partition of the vertex set V into k independent sets $V_1, \ldots, V_k$. For defining the set $\mathcal{S}$ of feasible solutions, the unique constraint has been relaxed and TS is used to find if possible a coloring in a given number k of colors. In other words, a feasible solution is any partition $(V_1, \ldots, V_k)$ of V into k sets. By defining $E(V_r)$ as the set of edges having both endpoints in V_r, the objective is to minimize $\sum_{r=1}^{k} |E(V_r)|$.

Hence a solution of value 0 corresponds to a coloring of G in k colors. Define $\mathcal{S}$ as the set of all partitions of V into independent sets (i.e., the unique constraint of the problem is not relaxed) and the objective function as the number of colors used. This implies that a large number of moves (to change the color of a unique vertex) have the same value – very few moves are improving – which makes it difficult to guide the search.

In the efficient modeling described above, the neighborhood $\mathcal{N}(i)$ of a solution i consists of all solutions generated from i by the following local modification: move a vertex that is an endpoint of a monochromatic edge (both endpoints are in the same V_r) to a set V_q, $q \neq r$. When a vertex v is moved from V_r to V_q, the pair (v, r) is introduced in the tabu list T. This means that it is forbidden to move v to V_r during $|T|$ iterations. For more details, the reader is referred to Hertz & de Werra [1987].

5.3 The maximum independent set problem

Given a graph $G = (V, E)$ we have to determine a subset X of V as large as possible such that $E(X) = \emptyset$, where $E(X)$ is a set of edges having both endpoints in

X [Friden, Hertz & de Werra, 1989, 1990]. Adaptations of tabu search attempt to determine in G an independent set of a given size k. Once again, the unique constraint of the problem has been relaxed. A solution is any subset $X \subset V$ of size k, and the objective function is $|E(X)|$. The neighborhood consists of exchanging a vertex $v \in X$ with a vertex $w \in V - X$. The size of the neighborhood of a solution is equal to $k(n-k) = O(nk)$, where $n = |V|$.

For a vertex $v \in V$, let us denote by $\Gamma_X(v)$ the set of vertices $w \in X$ that are linked to v by an edge. The vertices $v \in X$ are sorted according to nonincreasing values of $\Gamma_X(v)$ and the vertices $w \in V - X$ are sorted according to nondecreasing values of $\Gamma_X(w)$. Keeping these two lists sorted requires time $O(n)$. Three tabu lists are used. The first one T_1 stores the last visited solutions; the second one T_2 and the third one T_3 contain the last vertices introduced into and removed from X, respectively. Friden, Hertz & de Werra [1989] have shown that by using hashing techniques for implementing T_1 and choosing a small constant value for $|T_2|$ and $|T_3|$, a best neighbor may be found in almost constant time. Hence one step of TS takes time $O(n)$ instead of $O(nk)$.

A completely different adaptation of TS to the maximum independent set problem has been described by Friden, Hertz & de Werra [1990]. TS is used for getting bounds in a branch-and-bound algorithm. At each node of the enumerative algorithm, we define two sets I and O of vertices that are forced to be respectively inside and outside of a maximum independent set. Let X be the largest independent set found so far. TS is used for covering with $|X| - |I|$ cliques as many vertices as possible in the graph H induced by $U = V - \{I \cup O\}$. Let W be the set of vertices covered by the cliques. Backtracking occurs if $|W| = |U|$; this is because H cannot contain a stable set of size larger than $|X| - |I|$. Otherwise, we generate a new subproblem for each vertex $v \in U - W$ by adding v to I and by adding all vertices adjacent to v to O.

Using a less elaborate technique for covering the vertices of H by cliques requires less time, but the number of branches generated at each node of the branch-and-bound algorithm is generally much larger. It is worth spending time using TS at each node for reducing the number of branchings. More details can be found in Friden, Hertz & de Werra [1990].

5.4 The vehicle routing problem

Let $V = \{V_1, \ldots, V_n\}$ be a set of customers and V_0 a depot at which m identical vehicles are based. The VRP consists of designing a set of least-cost vehicle routes in such a way that each route starts and ends at the depot, and each customer of V is visited exactly once by exactly one vehicle [Gendreau, Hertz & Laporte, 1994; Taillard, 1993a; Semet & Taillard, 1993]. With every client is associated a demand, and the total demand of any vehicle route may not exceed the vehicle capacity. The total length of any route (travel plus service time) may not exceed a preset bound.

Many adaptations of TS have been designed for this problem [Willard, 1989; Pureza & França, 1991; Gendreau, Hertz & Laporte, 1994; Taillard, 1993a;

Osman, 1993; Semet & Taillard, 1993]. Most of them use a neighborhood function where customers are moved from one route to another. Details on other approaches can be found in Chapters 9 and 10 of this book.

If TS is implemented with a short-term memory only, it appears that the customers located close to the depot are moved more frequently than the others. In order to diversify the search, a penalty is added to the value of a move. This penalty is proportional to the frequency of the use of the move. For more details, the reader is referred to Taillard [1993a] and Gendreau, Hertz & Laporte [1994].

Another way of escaping local optima is to relax the capacity or total route length constraints and to add to the objective function components proportional to the violation of these constraints. The proportionality factor can be automatically updated by increasing it when the recently visited solutions are all violating the constraints, and by decreasing it when they are all feasible. More details can be found in Gendreau, Hertz & Laporte [1994].

5.5 The course scheduling problem

Course scheduling problems can be formulated as graph coloring problems: the courses are the vertices of a graph, the periods correspond to the colors, and two vertices are linked by an edge if the corresponding courses cannot be given at the same time [Hertz, 1991, 1992; Costa, 1994]. Many additional constraints have to be taken into account in real-life problems, and most optimization techniques can hardly deal with all the specific requirements of each school.

Tabu search has been successfully applied to this problem by extending its adaptation to the graph coloring problem. A feasible solution is defined as a schedule that satisfies a subset C of constraints, and a neighboring solution is obtained by modifying the schedule of one course. If all courses last one period, this is equivalent to changing the color of a vertex. In order to decide whether a specific constraint should be included in C, the following guidelines can be used: a course should never be assigned to a period if such an assignment induces only course schedules that do not satisfy all the constraints. Moreover, a conflict between two courses should not be a sufficient condition for hindering an assignment.

For example, let us consider two courses c_1 and c_2 involving common students. If c_1 cannot be scheduled at period p for some reason (maybe the teacher is not available at that time), we shall never visit a solution where c_1 is given at time p. However, if this constraint does not exist and c_2 is scheduled at time p, we are allowed to schedule c_1 at time p, even if this violates the constraint of no overlapping of courses involving common students. The reason is that c_1 is possibly given at time p in an optimal schedule and the violation of the constraint can be canceled by moving course c_2 to another period. As mentioned by Hertz [1992], such a model can handle timetabling problems with optional courses, time window and preassignment requirements, compactness, geographical and precedence constraints, and the courses can have different durations.

6 CONCLUSIONS

This chapter has illustrated the wide range of applications for tabu search. More examples can be found in Glover et al. [1993].

A simple technique, tabu search owes its efficiency to a rather fine tuning of an apparently large collection of parameters. In fact, experiments have shown that the degrees of freedom in most of these choices are not so restricted. Theoretical considerations based on a probabilistic model of tabu search seem partially to confirm this statement [Faigle & Kern, 1992]: when moves from a solution i to a solution j are performed according to a probability distribution p_{ij}, then under rather mild assumptions one can prove that convergence is obtained to an optimal solution for probabilities p_{ij} that are allowed to be chosen in a large subinterval of $[0, 1]$. This approach may lead to a better understanding of tabu search efficiency.

For now, suffice it to say that tabu search looks more like an engineering approach to large and difficult problems of optimization, less like an elegant algorithm of mathematical simplicity.

With tabu search, complexity is not only present in the problems but in the technique itself. This is clearly not an advantage. However, the situation may be viewed as an explanatory step in a new field of techniques, which we expect to become more elegant and simpler when memory, artificial intelligence and exploration procedures will be integrated more effectively.

Local Search in Combinatorial Optimization
Edited by E. Aarts and J. K. Lenstra
© 1997 John Wiley & Sons Ltd.

6

Genetic algorithms

Heinz Mühlenbein
GMD Schloss Birlinghoven

1 INTRODUCTION 137
2 EVOLUTIONARY ALGORITHMS 140
3 UNDERSTANDING GENETIC ALGORITHMS 142
4 A SURVEY OF POPULATION GENETICS 144
5 ARTIFICIAL SELECTION 146
6 NATURAL SELECTION 150
7 TWO LOCI 151
8 GENE POOL RECOMBINATION 153
9 SELECTION AND MUTATION 156
9.1 Mutation in small populations 157
9.2 Mutation in large populations 158
10 SUMMARY OF MAJOR RESULTS 160
11 MUTATION AND RECOMBINATION 162
12 STATISTICS AND GENETICS 163
13 APPLICATIONS OF THE THEORY 167
14 CONCLUSION 170

1 INTRODUCTION

Evolutionary algorithms which model natural evolution processes were already proposed for optimization in the 1960s. I cite just one representative example, the outstanding work of Bremermann [Bremermann, Roghson & Salaff, 1966]. He wrote:

> The major purpose of the work is the study of the effects of mutation, mating, and selection on the evolution of genotypes in the case of non-linear fitness functions. In view of the mathematical difficulties involved, computer experimentation has been utilized in combination with theoretical analysis.... In a new series of experiments we found evolutionary schemes that converge much better, but with no known biological counterpart.

These remarks are still valid. The designer of evolutionary algorithms should be inspired by nature, but he should not intend to copy. His or her major goal should be to develop powerful optimization methods. An optimization is powerful if it is able to solve difficult optimization problems. Furthermore the algorithm should be based on a solid theory. I object to popular arguments along the lines: 'this algorithm is a good optimization method because the principle is used in nature,'

and vice versa: 'this algorithm cannot be a good optimization procedure because you do not find it in nature.'

Modeling the natural evolution process and applying it to optimization problems is a challenging task. In principle, artificial selection of animals for breeding and selection of virtual animals on a computer are similar problems. Therefore the designer of an evolutionary algorithm can profit from the knowledge accumulated by human breeders. During the course of applying the algorithm to difficult fitness landscapes, the human breeder may also profit from the experience gained by using the algorithm.

Bremermann further notes:

> One of the results was unexpected. The evolution process may stagnate far from the optimum, even in the case of a smooth convex fitness function.... It can be traced to the bias that is introduced into the sampling of directions by essentially mutating one gene at a time. One may think that mating would offset this bias; however, in many experiments mating did little to improve convergence of the process.

Bremermann used the term *mating* for recombining two (or more) parent strings to create an offspring string. The *stagnation problem* will be solved in this chapter. Bremermann's algorithm contained most of the ingredients of a good evolutionary algorithm. But because of limited computer experiments and a missing theory, he did not find a good combination of the ingredients.

In the 1970s two different evolutionary algorithms independently emerged, the genetic algorithm (GA) of Holland [1975] and the evolution strategies of Rechenberg [1973] and Schwefel [1981]. Holland was not so much interested in optimization, but in adaptation. He investigated the genetic algorithm with decision theory for discrete domains. Holland emphasized the importance of recombination in large populations, whereas Rechenberg and Schwefel mainly investigated mutation in very small populations for continuous parameter optimization.

Both algorithms have been used by small groups for quite a time. Both groups developed a unique notation using a mixture of biological and computer science terms. In this chapter evolutionary algorithms are considered to be part of mathematical random search methods. But the theoretical analysis is unusual from a mathematical point of view. The analysis is based on classical population genetics and statistics. It provides some answers to the following questions:

- How should the selection be done?
- Given a selection and recombination scheme, what is the expected progress of the population?
- Given a selection and a mutation scheme, what is the expected progress of the population?
- How can selection, mutation, and recombination be combined in a synergistic manner?

This approach is opposite to the standard GA analysis initiated by Holland [1975], which starts with the schema theorem. The theorem predicts the progress

of schemata under a specific selection schedule, called proportionate selection. Mutation and recombination are later introduced as disruptions of the population. In contrast to this view, I regard mutation and recombination as constructive search operators. They have to be evaluated according to the probability that they create better solutions.

The search strategies of mutation and recombination are different. Mutation is based on chance. The progress for a single mutation step is almost unpredictable. Recombination is a more global search based on restricted chance. The bias is implicitly given by the population. Recombination only shuffles the substrings contained in the population. The substrings of the optimum have to be present in the population. Otherwise a search by recombination is not able to locate the optimum. Recombination needs a large population size. In principle, evolutionary algorithms using selection and mutation or selection and recombination alone are able to locate the optimum. Therefore the most difficult question of population genetics and evolutionary algorithms remains: Why use both search strategies together?

In the analysis I distinguish between empirical laws and theorems. Empirical laws are derived from carefully performed computer experiments. Theorems are obtained by purely mathematical reasoning. Empirical laws are by no means less true than theorems. They are laws explaining the results of numerical experiments. This procedure was and is successfully used in physics. The laws describing the movements of the planets are a historical example. Kepler derived his famous laws empirically. They explained all the available data and could be used for prediction. Newton was able to derive the same laws by postulating a gravitational force between the sun and the planets. Thus in Newton's theory Kepler's laws can be proven mathematically. In my terminology Newton converted an empirical law to a theorem by a theory. The main source of the confusion today is that the word *empirical* has one sense in which it refers to something based purely on observation without theoretical depth. But I use the word in its classical sense. Widespread availability of computers makes the empirical approach a viable alternative to the analytical approach.

The outline of this chapter is as follows. First some of the most popular evolutionary algorithms are broadly described. These algorithms differ in their emphasis on selection, mutation, and recombination. A theoretical understanding is required in order to evaluate the different algorithms. Unfortunately, it turns out that such an analysis is very difficult, even in the case of simple functions. In Section 3 I will briefly describe the early approaches to understanding genetic algorithms. This is followed by a very short survey of population genetics. The analysis starts with the equation for the *response to selection*, which predicts the expected average progress of the population. The equation is used to analyze the dynamic behavior of a genetic population with recombination and selection. In Section 7 the two loci case is analyzed in detail. This analysis leads to a new, more efficient recombination scheme, called *gene pool recombination*, which is discussed in Section 8. Selection and mutation are investigated in Section 9. After a summary of the major theoretical results, the search strategies of

mutation and recombination are compared in Section 11. Problems of population genetics had a major influence on the development of modern statistics. One problem, estimating the heritability, is described in Section 12. The last section shows how to apply the theory numerically.

The theory that will be presented was developed together with a number of researchers. In the following sections I will use *we* to acknowledge the common work.

2 EVOLUTIONARY ALGORITHMS

A previous survey of search strategies based on evolution has been done by Mühlenbein, Gorges-Schleuter & Krämer [1988]. Evolutionary algorithms for continuous parameter optimization are surveyed by Bäck & Schwefel [1993].

Algorithms that are driven mainly by mutation and selection have been developed by Rechenberg [1973] and Schwefel [1981] for continuous parameter optimization. Their algorithms are called evolution strategies (ES). The following strategy is the most famous one.

$(\mu + \lambda)$ Evolution strategy

1. Create an initial population of size λ.
2. Compute the fitness $F(x_i)$, $i = 1, \ldots, \lambda$.
3. Select the $\mu \leqslant \lambda$ best individuals.
4. Create λ/μ offspring of each of the μ individuals by small variation.
5. If not finished, return to Step 2.

An evolution strategy is a random search which uses selection and variation. The small variation is done by randomly choosing a number from a normal distribution with zero mean. This number is added to the value of the continuous variable. The algorithm adapts the amount of variation by changing the variance of the normal distribution. The most popular algorithm uses $\mu = \lambda = 1$.

In biological terms, evolution strategies model natural evolution by asexual reproduction with mutation and selection. Search algorithms that model sexual reproduction are called genetic algorithms (GAs). Sexual reproduction is characterized by recombining two parent strings into an offspring. This recombination is often called *crossover*. Genetic algorithms were invented by Holland [1975]. Recent surveys can be found in Goldberg [1989] and the proceedings of the International Conferences on Genetic Algorithms (ICGA) [Schaffer, 1989; Belew & Booker, 1991; Forrest, 1993].

Genetic algorithm

0. Define a genetic representation of the problem.
1. Create an initial population $P(0) = x_1^0, \ldots, x_N^0$. Set $t = 0$.
2. Compute the average fitness $\bar{f}(t) = \sum_i^N f(x_i)/N$. Assign each individual the normalized fitness value $f(x_i)/\bar{f}(t)$.

3. Assign each x_i a probability $p(x_i, t)$ proportional to its normalized fitness. Using this distribution, select N vectors from $P(t)$. This gives the set of the selected parents.
4. Pair all parents at random forming $N/2$ pairs. Apply crossover with a certain probability to each pair and other genetic operators such as mutation, forming a new population $P(t+1)$.
5. Set $t = t + 1$ and return to Step 2.

In the simplest case the genetic representation is just a bitstring of length n, the *chromosome*. The positions of the strings are called *loci* of the chromosome. The variable at a locus is called a *gene*, its value an *allele*. The set of chromosomes is called the *genotype*, which defines a *phenotype* (the individual) with a certain fitness.

The genetic operator *mutation* changes with a given probability m each bit of the selected string. The *crossover* operator works with two strings. If two strings $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$ are given, then the *uniform crossover* operator [Syswerda, 1989] combines the two strings as follows:

$$z = (z_1, \ldots, z_n),\ z_i = x_i \text{ or } z_i = y_i.$$

Normally x_i or y_i are chosen with equal probability. Many different crossover operators are used in genetic algorithms. We will only consider uniform crossover. In general we will use the more general term *recombination* for any method combining two or more strings.

A genetic algorithm is a parallel random search with centralized control. The centralized part is the selection schedule. The selection needs the average fitness of the population. The result is a highly synchronized algorithm, which is difficult to implement efficiently on parallel computers. In the *parallel genetic algorithm* (PGA) [Mühlenbein, 1991a], a distributed selection scheme is used. This is achieved as follows. Each individual does the selection by itself. It looks for a partner in its neighborhood only. The set of neighborhoods defines a spatial population structure.

The second major change can also be easily understood. Each individual is active and not acted on. It may improve its fitness during its lifetime by performing a local search. The parallel genetic algorithm PGA is described as follows.

Parallel genetic algorithm

0. Define a genetic representation of the problem.
1. Create an initial population and its population structure.
2. Each individual does local hill climbing.
3. Each individual selects a partner for mating in its neighborhood.
4. An offspring is created using genetic operators like recombination and mutation.
5. The offspring does local hill climbing. It replaces the parent if it is better than some criterion (acceptance).
6. If not finished, return to Step 3.

Note that each individual may use a different local hill-climbing method. This feature will be important for problems where the efficiency of a particular hill-climbing method depends on the problem instance.

In the PGA the information exchange within the whole population is a diffusion process because the neighborhoods of the individuals overlap. All decisions are made by the individuals themselves. Therefore the PGA is a totally distributed algorithm without any central control. The PGA models the *natural evolution process*, which organizes itself.

The *distributed breeder genetic algorithm* (DBGA) [Mühlenbein & Schlierkamp-Voosen, 1993] is inspired by the science of breeding animals. Each one of a set of virtual breeders has the task to improve its own subpopulation. Occasionally the breeder imports individuals from neighboring subpopulations. The DBGA models *rational controlled evolution*. We will describe the breeder genetic algorithm only.

Breeder genetic algorithm

0. Define a genetic representation of the problem.
1. Create an initial population $P(0)$ of size N. Set $t = 0$.
2. Each individual may perform local hill climbing.
3. The breeder selects $T\%$ of the population for mating. This gives the selected parents.
4. Pair the parents at random forming N pairs. Apply the genetic operators crossover and mutation, forming a new population $P(t + 1)$.
5. Set $t = t + 1$ and return to Step 2.

The major difference between the genetic algorithm and the breeder genetic algorithm is the method of selection. The breeders have developed many different selection strategies. We only want to mention *truncation selection*, which breeders usually apply for large populations. In truncation selection the $T\%$ best individuals of a population are selected as parents. The breeder genetic algorithm uses genetic representations that depend on the problem. Continuous functions are represented by continuous genes, discrete functions by discrete genes. The genetic operators mutation and recombination are different for the two representations. The theory outlined in this chapter can be used for both representations, but we focus on discrete functions.

3 UNDERSTANDING GENETIC ALGORITHMS

Previous studies on why and how the genetic algorithm works have proceeded along two different lines. In the first line of approach, the theoretical one, most arguments are based on the *schema theorem* [Goldberg, 1989]. Mühlenbein [1991a] mentioned that this theorem cannot be used to explain the search strategy of the genetic algorithm. The reason is that the schema theorem does not include the gene frequencies of the actual population. Furthermore, it describes

only the outcome of proportionate selection. Mutation and recombination are only introduced as perturbations.

We do not want to discuss all the wrong interpretations of this theorem, but only the most simple central failure. For simplicity we assume binary functions. Then a schema is defined over the ternary alphabet $\{0, 1, *\}$. A schema matches a particular string if at every location in the schema a 1 matches a 1 in the string, a 0 matches a 0, and a * matches either. Let $m(H, t)$ be the number of occurrences of schema H in the population at generation t. Then the schema theorem computes $m(H, t+1)$ for proportionate selection [Goldberg, 1989] by

$$m(H, t+1) = m(H, t)\,\frac{f(H, t)}{\bar{f}(t)}.$$

This equation immediately follows from the definition of proportionate selection. $f(H, t)$ is the average fitness of schema H at generation t, $\bar{f}(t)$ is the average fitness of the population. Both variables depend on the population at generation t. They can only be computed if the population at generation t is known. In order to circumvent this problem, Goldberg [1989, p. 30] argues as follows:

> Suppose that a particular schema remains above average an amount $c\bar{f}$ with c a constant. Then
>
> $$m(H, t+1) = m(H, t)\cdot(1 + c).$$
>
> Starting at $t = 0$ and assuming a stationary value of c, we obtain the equation
>
> $$m(H, t) = m(H, 0)\cdot(1 + c)^t.$$
>
> The effect of reproduction is now quantitatively clear; reproduction allocates exponentially increasing numbers of trials to above-average schemata.

The argument is used to claim that proportionate selection is allocating the trials in an optimal manner. The failure of this argument is simple: there can't be a schema remaining a constant c above average. The average fitness of the population is steadily increasing, and convergence of the algorithms means that there is only one genotype left in the population.

Because the schema theorem could not be used to predict the behavior of practical genetic algorithms, a second line of research has been tried. This line was experimental and top-down. A full-blown genetic algorithm was run and the parameters of the GA were optimized for a suite of test functions. An example is the work of Grefenstette [1986]. His research was extended by Schaffer and coworkers [1989]. Empirically the researchers found that the following formula gave an almost optimal parameter setting for their test suite:

$$\ln N + 0.93 \ln m + 0.45 \ln n = 0.56,$$

where

N = size of the population,

m = mutation rate,

n = length of the chromosome.

This equation can be approximated by

$$N m \sqrt{n} = 1.7.$$

The formula indicates that an optimal mutation rate should decrease with the size of the population. We will show that this is not the case. The failure of the statistical approach was to extend the regression results beyond the problem domain considered.

The major mistake of the early theoretical research was to underestimate the difficulty of the problem. Surprisingly the researchers did not seriously consider the theory that has been developed in a different field for understanding the evolution of genetic populations – population genetics.

4 A SURVEY OF POPULATION GENETICS

Population genetics tries to analyze the evolution of genetic populations. A genetic algorithm consists of an artificial genetic population. Therefore population genetics should also be applicable to genetic algorithms. Why has it taken so long to recognize this?

There are two major reasons. First, GA researchers believed in a new theory based on the schema theorem. The second reason can be found within population genetics itself. Population genetics consists of a diversity of different models and methods, which are not easily understood or transferred to the needs of genetic algorithms. The major obstacle lies in the fact that the theoretical analysis of evolution centered on understanding evolution in a natural environment. It tried to model *natural selection*. The term *natural selection* was informally introduced by Darwin in his famous book *On the Origins of Species by Means of Natural Selection*. He wrote, 'The preservation of favourable variations and the rejection of injurious variations, we call Natural Selection.' Modeling natural selection mathematically is difficult. Normally biologists introduce another term, the fitness of an individual, which is defined as the number of offspring of that individual. This fitness definition cannot be used for prediction. It can only be measured after the individual is not able to reproduce anymore. *Artificial selection* as used by breeders is seldom investigated in textbooks on quantitative genetics. It is described in more practical books aimed at the breeders. We believe that this is a mistake. Artificial selection is a controlled experiment, like an experiment in physics. It can be used to isolate and understand specific aspects of evolution. Individuals are selected by the breeder according to some trait. Predicting the outcome of a breeding programme plays a major role in artificial selection. Note that *proportionate selection*, as used by the simple genetic algorithm, is used in population genetics as a model for natural selection.

Unfortunately the behavior of genetic populations is very difficult to analyze mathematically. Therefore population genetics developed a set of models and a set of approaches that investigate specific aspects of genetic populations. Historically three different approaches have been tried:

- phenotypic, used by the biometricians (Galton, Pearson),
- genotypic, used by the Mendelians,
- statistical, used by breeders.

The biometricians mainly used the concept of *correlation* and *regression* to quantify the relation between offspring and parent. Their analysis centers on quantitative traits. The Mendelians use Mendel's *genetic chance model* to compute the change of the gene frequencies in the population. Mendel's model is restricted to discrete genes. The scientific way of breeding starts with the equation for the *response to selection*. It tries to predict the outcome of selection experiments. Modern textbooks about population genetics mainly describe the Mendelian approach.

For the theory of genetic algorithms all three approaches are useful. At least five parameters are required to describe the initial state and the evolution of an artificial population of a genetic algorithm:

- population size N,
- initial frequency of the alleles p_0,
- number of loci n,
- mutation rate m,
- intensity of selection I.

It would be futile to investigate the genetic algorithm with all five parameters variable. Therefore we will investigate simpler models with one or more parameters fixed. A similar approach has been used in population genetics.

We just summarize the most important models of population genetics. The interested reader should consult Crow & Kimura [1970] or Naglyaki [1992] for a more comprehensive survey.

- *The Fisher–Wright model.* This discrete stochastic model is based on Mendel's genetic chance model. It was analyzed by a Markov chain approach. Some analytical results have been obtained for one locus and proportionate selection.
- *The Kimura (diffusion) approach.* This continuous time model is a diffusion approximation of the Fisher–Wright model. It was used to obtain additional analytical results for the Fisher–Wright model. Kimura extended it to analyze evolution without selection (genetic drift).
- *Two loci.* Researchers in population genetics have not been able to compute analytical results for the Fisher–Wright model for more than one locus with Mendelian recombination. Therefore a specific model was defined. It assumes an infinite population, and the evolution of the genotype frequencies is described by deterministic difference equations.
- *Many loci (quantitative genetics).* This model has been developed by the biometricians. It is purely descriptive, no genetics is involved. It is also purely phenotypic. Deriving the equations of this model from an extension of Mendel's chance model was a challenge to population genetics, and remains a challenge even now.

After 100 years of hard research, some mathematical results have been obtained for these models. It is rather unlikely that GA researchers will find mathematical solutions to some of the problems which remain open. But progress is to be expected by carefully designed simulation experiments. Such a blending of mathematical analysis and simulation experiments will be emphasis of this chapter.

5 ARTIFICIAL SELECTION

In this section we will first investigate artificial selection using methods described by Falconer [1981]. Later we will use the same method the analyze natural selection modeled by proportionate selection. The change produced by selection that mainly interests the breeder is the *response to selection*, symbolized by R. R is defined as the difference between the population mean fitness of generation $t+1$ and the population mean of generation t. $R(t)$ measures the expected progress of the population:

$$R(t) = \bar{f}(t+1) - \bar{f}(t).$$

Breeders measure the selection with the *selection differential*, symbolized by S. $S(t)$ is defined as the difference between the mean fitness of the selected parents $\bar{f}_s(t)$ and the mean fitness of the population:

$$S(t) = \bar{f}_s(t) - \bar{f}(t).$$

The breeder tries to predict $R(t)$ from $S(t)$. This regression is shown in Figure 6.1. The curves represent the fitness distribution of the phenotypes at generations t and $t+1$.

Breeders often use *truncation selection or mass selection* (Figure 6.1). In truncation selection with threshold *Trunc*, the *Trunc* best individuals will be selected as parents. *Trunc* is normally chosen in the range 10–50%.

The prediction of the response to selection starts with

$$R(t) = b(t) \cdot S(t).$$

$b(t)$ is called the *realized heritability*. The breeder either measures $b(t)$ in previous generations or estimates $b(t)$ by different methods [Crow, 1986]. It is normally assumed that $b(t)$ is constant for a certain number of generations. This leads to

$$R(t) = b \cdot S(t), \tag{1}$$

a relation which involves no genetics; indeed, it goes back to a series of papers written by Pearson in the 1890s. The prediction of just one generation is only half the story. The breeder (and the GA user) would like to predict the cumulative response R_s for s generations of his or her breeding scheme,

$$R_s = \sum_{t=0}^{s} R(t).$$

The computation of R_s requires a second equation. Several approximate equa-

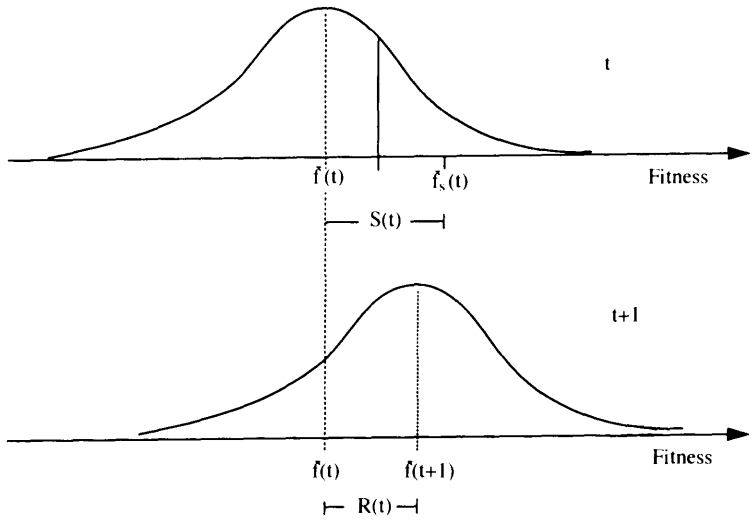

Figure 6.1 Regression of truncation selection: fitness distribution at (a) time t and (b) time $t+1$

tions for $S(t)$ are proposed in quantitative genetics [Bulmer, 1980; Falconer, 1981], but most of them are only valid for *diploid* organisms. Diploid organisms have two sets of chromosomes. Most genetic algorithms use one set of chromosomes – they deal with *haploid* organisms – so we can only apply the research methods of quantitative genetics, not the results.

Breeders have found that the dimensionless quantity

$$I = S(t)/\sigma(t) \tag{2}$$

provides a more convenient measure of the strength of selection; $\sigma(t)$ is the standard deviation of the fitness of the individuals and I is called the *selection intensity*. I can be computed analytically, if the fitness values are normally distributed. A derivation can be found in Bulmer [1980]. Simulations have shown that equation (2) is approximately valid for many practical applications, also when the fitness values are not normally distributed. In fact, for arbitrary distributions the following estimate has been proved by Nagaraja [1982]:

$$S(t)/\sigma(t) \leqslant \sqrt{(100 - Trunc)/Trunc}.$$

Table 6.1 shows the relation between the truncation threshold *Trunc* and the selection intensity I for a normal distribution. A decrease from 50% to 1% leads to an increase of the selection intensity from 0.8 to 2.66.

If we insert (2) into (1), we obtain the equation for the *response to selection* [Falconer, 1981]:

$$R(t) = h \cdot I \cdot \sigma(t).$$

Table 6.1 Selection intensity

Trunc	80%	50%	40%	20%	10%	1%
I	0.34	0.8	0.97	1.4	1.76	2.66

The science of artificial selection consists of estimating b and $\sigma(t)$. These estimates depend on the specific traits to be improved. The designer of a genetic algorithm has more freedom. In order to maximize the response, the designer can look for a recombination operator that *maximizes the product of the realized heritability* and the *standard deviation* of the offspring generation. This design goal is difficult to fulfill. An application of this design principle to a continuous fitness function can be found in Voigt, Mühlenbein & Cvetkovic [1995], where a fuzzy recombination operator is shown to be superior to other recombination operators.

The equation for the response to selection can also be used for analyzing selection methods. Given a certain class of selection methods, all of which have the same selection intensity I, the preferred selection method generates parent populations with the *highest standard deviation*. Blickle & Thiele [1995] used this method to compare tournament selection and truncation selection. They showed that tournament selection creates a parent population with a higher standard deviation than truncation selection with the same selection intensity.

We will now apply the response-to-selection equation to predict the behavior of a genetic algorithm. As an introductory example, we will use the binary ONEMAX function of size n. Here the fitness is given by the number of 1's in the binary string.

We will first estimate b. A popular method for estimating b is to make a regression of the midparent fitness value to the offspring. The midparent fitness value is defined as the average of the fitness of the two parents. We assume *uniform crossover* for recombination.

A simple calculation for the simple ONEMAX function shows that the probability of the offspring being better than the midparent is equal to the probability of them being worse. Therefore the average fitness of the offspring will be the same as the average of the midparents. But this means that the average of the offspring is the same as the average of the selected parents. This gives $b = 1$ for ONEMAX.

Estimating $\sigma(t)$ is more difficult. We make the assumption that uniform crossover is a random process which creates a binomial fitness distribution with probability $p(t)$, the probability that there is a 1 at a locus. Therefore the standard deviation is given by

$$\sigma(t) = \sqrt{np(t)(1 - p(t))}.$$

Theorem 1 *If the population is large enough to converge to the optimum and if the selection intensity I is greater than* 0, *the response to selection for the ONEMAX*

function is approximately given by

$$R(t) = \frac{I}{\sqrt{n}} \cdot \sqrt{p(t)(1 - p(t))}.$$

The number of generations needed until equilibrium is approximately

$$GEN_e = \left[\frac{\pi}{2} - \arcsin(2p_0 - 1)\right]\frac{\sqrt{n}}{I}. \tag{3}$$

$p_0 = p(0)$ *denotes the probability of the advantageous bit in the initial population.*

Proof Noting that $R(t) = n(p(t+1) - p(t))$ we obtain the difference equation

$$p(t+1) - p(t) = \frac{I}{\sqrt{n}} \cdot \sqrt{p(t)(1 - p(t))}.$$

The difference equation can be approximated by a differential equation

$$\frac{dp(t)}{dt} = \frac{I}{\sqrt{n}} \cdot \sqrt{p(t)(1 - p(t))}.$$

The initial condition is $p(0) = p_0$. The solution of the differential equation is given by

$$p(t) = 0.5\left(1 + \sin\left[\frac{I}{\sqrt{n}}t + \arcsin(2p_0 - 1)\right]\right).$$

The convergence of the total population is characterized by setting $p(GEN_e) = 1$. This gives equation (3). □

Remark Simulations show that the phenotypic variance is slightly less than given by the binomial distribution. The empirical data are better fitted if the binomial variance is reduced by a factor $\pi/4.3$. Using this variance one obtains the equations

$$\tilde{R}(t) = \frac{\pi}{4.3} \cdot \frac{I}{\sqrt{n}} \cdot \sqrt{p(t)(1 - p(t))},$$

$$\widetilde{GEN}_e = \frac{4.3}{\pi}\left[\frac{\pi}{2} - \arcsin(2p_0 - 1)\right]\left(\frac{\sqrt{n}}{I}\right).$$

These equations fit the results obtained from simulations very well. The number of generations needed until convergence is proportional to $\sqrt{n}$ and inversely proportional to the selection intensity. Note that the equations are only valid if the population is large enough to converge to the optimum. The most efficient breeder genetic algorithm runs with the *minimal pop-size* N^*, so that the population still converges to the optimum. N^* depends on the size of the problem n, the selection intensity I, and the probability of the advantageous bit p_0. The determination of N^* is outside the scope of this chapter. The interested reader is referred to Mühlenbein & Schlierkamp-Voosen [1994b].

6 NATURAL SELECTION

Natural selection is modeled by proportionate selection in quantitative genetics. It is defined as follows. Let $0 \leqslant q_i(t) \leqslant 1$ be the frequency of genotype i in a population of size N at generation t, m_i its fitness. Then the genotype frequency of the selected parents is given by

$$q_{i,S}(t) = q_i(t)\frac{m_i}{\bar{f}(t)},$$

where $\bar{f}(t) = \sum q_i(t)m_i$ is the average fitness of the population. Note that the above equation is just the schema theorem applied to strings (schemata of length n).

Theorem 2 *In proportionate selection the selection differential is given by*

$$S(t) = \frac{\sigma^2(t)}{\bar{f}(t)}. \tag{4}$$

Let the fitness function be ONEMAX(n). *It is assumed that the genotypes are binomially distributed,* $\sigma^2(t) \approx np(t)(1 - p(t))$. *Then the response to selection is given by*

$$R(t) = 1 - p(t). \tag{5}$$

If the population is large enough, the number of generations until $p(t) = 1 - \varepsilon$ *for large n is given by*

$$GEN_{1-\varepsilon} \approx n \cdot \ln\frac{1 - p_0}{\varepsilon}. \tag{6}$$

p_0 *is the probability of the advantageous allele in the initial population.*

Proof

$$\begin{aligned} S(t) &= \sum_{i=1}^{N} q_{i,S}m_i - \bar{f}(t) \\ &= \sum_{i=1}^{N} \frac{q_i(t)m_i^2 - q_i(t)\bar{f}^2(t)}{\bar{f}(t)} \\ &= \frac{1}{\bar{f}(t)} \sum_{i=1}^{n} q_i(t)(m_i - \bar{f}(t))^2 \\ &= \frac{\sigma^2(t)}{\bar{f}(t)}. \end{aligned}$$

For ONEMAX(n) we have $R(t) = S(t)$. Because $\bar{f}(t) = np(t)$, equation (5) is obtained. From $R(t) = n(p(t+1) - p(t))$ we get the difference equation

$$p(t+1) = \frac{1}{n} + \left(1 - \frac{1}{n}\right)p(t).$$

This equation has the solution

$$p(t) = \frac{1}{n}\left[1 + \left(1 - \frac{1}{n}\right) + \cdots + \left(1 - \frac{1}{n}\right)^{t-1}\right] + \left(1 - \frac{1}{n}\right)^{t} p_0.$$

This equation can be simplified to

$$p(t) = 1 - \left(1 - \frac{1}{n}\right)^{t}(1 - p_0).$$

By setting $p(GEN_{1-\varepsilon}) = 1 - \varepsilon$ equation (6) is easily obtained. □

Remark If we assume $R(t) = S(t)$ we obtain from equation (4) a version of Fisher's *fundamental theorem of natural selection* [Fisher, 1958].

By comparing truncation selection and proportionate selection one observes that proportionate selection gets weaker when the population approaches the optimum. An infinite population will need an infinite number of generations for convergence. This effect was observed early in practical applications of genetic algorithms. Therefore many different extensions to proportionate selection have been invented. The theory presented here shows that the progress of the genetic population only depends on the selection intensity and the variance of the population. The specific selection procedure does not matter. Up to now the only alternative to truncation selection has been *tournament selection* [Goldberg & Deb, 1991] with a reasonable tournament size. In fact, one can show that the tournament size may be mapped onto the selection intensity defined earlier for truncation selection.

With truncation selection, the population will converge in at most $O(\sqrt{n})$ generations, independent of the size of the population. Truncation selection, as used by breeders, is therefore a much more effective method of optimization than proportionate selection.

7 TWO LOCI

In this section we will derive exact equations for infinite populations. For simplicity we restrict the discussion to two loci and proportionate selection. In this case there are four possible genotypes: (0, 0), (0, 1), (1, 0), and (1, 1), which we index by $i = 0, 1, 2, 3$. We denote their fitness values by m_0, m_1, m_2, and m_3, respectively. Let $q_i(t)$ be the frequency of genotype i at generation t. We assume an infinite population and *uniform crossover*. The exact equations describing the evolution of the frequencies q_i can be derived easily for proportionate selection. These equations – known for diploid chromosomes in population genetics [Crow & Kimura, 1970] – are:

$$q_i(t+1) = \frac{m_i}{\bar{f}(t)} q_i(t) + \varepsilon_i \frac{D(t)}{2\bar{f}(t)^2}, \quad i = 0, 1, 2, 3 \tag{7}$$

with $\varepsilon = (-1, 1, 1, -1)$. $\bar{f}(t) = \sum_{i=0}^{3} m_i q_i(t)$ is the average fitness of the population.

$D(t)$ defines the deviation from linkage equilibrium:

$$D(t) = m_0 m_3 q_0(t) q_3(t) - m_1 m_2 q_1(t) q_2(t).$$

Note that $D(t) = 0$ if $q_0(t)q_3(t) = q_1(t)q_2(t)$ and $m_0 m_3 = m_1 m_2$. The first condition is fulfilled if the genotypes are binomially distributed. This assumption is called the *Hardy–Weinberg equilibrium* in population genetics. Although the general nonlinear difference equations have not yet been solved analytically (see the discussion by Naglyaki [1992]), it is possible to derive an exact expression for the realized heritability. By summation we obtain

$$R(t) = \bar{f}(t+1) - \bar{f}(t) = \frac{V(t)}{\bar{f}(t)} - (m_0 + m_3 - m_1 - m_2)\frac{D(t)}{2\bar{f}(t)^2},$$

where $V(t) = \sigma^2 = \sum q_i(t)(m_i - \bar{f}(t))^2$ denotes the variance of the population. Using $S(t) = V(t)/\bar{f}(t)$ we obtain the exact equation for the heritability,

$$b(t) = 1 - (m_0 - m_1 - m_2 + m_3)\frac{D(t)}{2\bar{f}(t)V(t)}.$$

In general, $b(t)$ depends on the genotype frequencies. Note that $b(t) = 1$ if $D(t) = 0$ or $m_0 + m_3 = m_1 + m_2$. The second assumption is fulfilled for the function ONEMAX(2), which has the fitness values $m_0 = 0, m_1 = m_2 = 1, m_3 = 2$. From (7) we obtain

$$\begin{aligned}\bar{f}(t+1) &= q_1(t+1) + q_2(t+1) + 2q_3(t+1)\\ &= \frac{q_1(t) + q_2(t) + 4q_3(t)}{\bar{f}(t)}\\ &= 1 + \frac{2q_3(t)}{\bar{f}(t)}.\end{aligned}$$

Let $p(t)$ denote the frequency of allele 1. By definition $\bar{f}(t) = 2p(t)$. Therefore we obtain

$$R(t) = 1 - p(t) + \frac{B_3(t)}{p(t)},$$

where $B_3(t)$ denotes how far $q_3(t)$ deviates from the frequency given by the binomial distribution,

$$B_3(t) = q_3(t) - p^2(t).$$

The exact difference equation for $p(t)$ can be written as

$$p(t+1) = p(t) + \frac{1}{2}(1 - p(t)) + \frac{B_3(t)}{2p(t)}.$$

This equation has two unknown variables, $p(t)$ and $q_3(t)$. Therefore $p(t)$ cannot be directly computed. Selection leads the population away from the binomial

distribution, and two-parent recombination (TPR) is not able to recreate a binomial distribution for the offspring population.

We now discuss a function where $D(t) = 0$ if the population starts in a Hardy–Weinberg equilibrium. An example is MULT(2) with fitness values $m_0 = 1$, $m_1 = m_2 = 2$, $m_3 = 4$. In this case the difference equation for $p(t)$ is given by

$$p(t+1) = p(t)\frac{2}{1+p(t)}, \tag{8}$$

which can be solved easily.

In summary, even linear fitness functions lead to difficult systems of difference equations. The genetic population moves away from Hardy–Weinberg equilibrium. A class of multiplicative fitness functions with $m_0 m_3 = m_1 m_2$ leads to simpler equations, because the population stays in Hardy–Weinberg equilibrium.

8 GENE POOL RECOMBINATION

The exact analysis of recombination together with selection leads to difficult nonlinear differential equations. Recombination of two genotypes creates a linkage between the genes at different loci. This linkage is very hard to describe mathematically. Therefore we decided to look for a recombination operator that leads to simpler equations, like those we used as an approximation. This operator must create a binomial distribution. Fortunately, there is a simple recombination scheme that fulfills this condition; we call it *gene pool recombination* (GPR).

Definition *In gene pool recombination the two 'parent' alleles of an offspring are randomly chosen with replacement from the gene pool given by the parent population selected before. Then the offspring allele is computed using any of the standard recombination schemes for TPR.*

For binary functions GPR is obviously a Bernoulli process. Let $p_i^s(t)$ be the frequency of allele 1 at locus i in the *selected* parent population. Then GPR creates offspring with allele frequency $p_i(t+1) = p_i^s(t)$ and variance $p_i(t+1)(1 - p_i(t+1))$ at locus i.

The idea of using more than two parents for recombination is not new. Mühlenbein [1989] has already used eight parents; the offspring allele was obtained by a majority vote. Multiparent recombination has also been investigated recently by Eiben, Raue & Ruttkay [1994], though their results are somewhat inconclusive. For binary functions the bit-based simulated crossover (BSC) of Syswerda [1993] is similar to GPR. However, his implementation merged selection and recombination. An implementation of BSC that separates selection and recombination was empirically investigated by Eshelman & Schaffer [1993]. GPR is an extension of BSC; it can be used for representation, discrete or continuous. Variants of GPR have also been successfully used for the traveling salesman problem; see Chapter 8.

In order to analyze GPR we will derive difference equations for the gene frequencies, valid for arbitrary fitness functions and infinite populations. As before, we restrict the analysis to the case of two loci and proportionate selection.

Let $q_i(t)$ be the frequency of genotype i at generation t. For $n = 2$ loci, the marginal gene frequencies $p_1(t)$ and $p_2(t)$ can be obtained from

$$p_1(t) = q_2(t) + q_3(t),$$
$$p_2(t) = q_1(t) + q_3(t).$$

We assume that the initial population has a binomial distribution. This means that

$$\begin{aligned} q_0(0) &= (1 - p_1(0))(1 - p_2(0)), \\ q_1(0) &= (1 - p_1(0))p_2(0), \\ q_2(0) &= p_1(0)(1 - p_2(0)), \\ q_3(0) &= p_1(0)p_2(0). \end{aligned}$$

Then the following theorem holds.

Theorem 3 *Let the initial population have a binomial distribution. For an infinite population with GPR and proportionate selection, the marginal frequencies $p_1(t)$ and $p_2(t)$ can be obtained from*

$$p_1(t+1) = p_1(t)\frac{m_2(1 - p_2(t)) + m_3 p_2(t)}{\bar{f}(t)}, \tag{9}$$

$$p_2(t+1) = p_2(t)\frac{m_1(1 - p_1(t)) + m_3 p_1(t)}{\bar{f}(t)}. \tag{10}$$

The realized heritability, $b(t)$, is given by

$$b(t) = 1 - (m_0 m_3 - m_1 m_2)(m_0 - m_1 - m_2 + m_3)\frac{p_1 p_2 (1 - p_1)(1 - p_2)}{V\bar{f}},$$

where p_1, p_2, V, and $\bar{f}$ depend on t.

Proof Proportionate selection selects the genotypes for the parents of population $t + 1$ according to

$$q_i^s(t) = \frac{m_i}{\bar{f}(t)} q_i(t).$$

From q_i^s the marginal frequencies p_1^s and p_2^s can be obtained from the two equations $p_1^s(t) = q_2^s(t) + q_3^s(t)$ and $p_2^s(t) = q_1^s(t) + q_3^s(t)$. For each locus, GPR is a Bernoulli process; therefore, the marginal gene frequencies of parents and offspring remain constant

$$p_1(t+1) = p_1^s(t), \quad p_2(t+1) = p_2^s(t).$$

Combining these equations gives equations (9) and (10). The expression for the realized heritability can be obtained after some manipulations. □

Remark Theorem 3 can be extended to arbitrary functions of size n, or genetically speaking, to n loci. This means that the evolution of an infinite genetic population with GPR and proportionate selection is fully described by n equations for the marginal gene frequencies. In contrast, for TPR one needs 2^n equations for the genotypic frequencies. For GPR one can in principle solve the difference equations for the marginal gene frequencies instead of running a genetic algorithm.

Note that $b(t) = 1$ if $m_0 m_3 = m_1 m_2$ or if $m_1 + m_2 = m_0 + m_3$. Let us first consider ONEMAX(2). The average fitness is given by $\bar{f}(t) = p_1(t) + p_2(t)$. If $p_1(0) = p_2(0)$, we have $p_1(t) = p_2(t) = p(t)$ for all t. From $\bar{f}(t) = 2p(t)$ we obtain

$$R(t) = 1 - p(t)$$

and

$$p(t+1) = p(t) + \frac{1}{2}(1 - p(t)).$$

This equation is similar to the equation obtained for TPR, which was solved in the previous section. Both equations become equal if $B_3(t) = 0$. This shows that GPR and TPR give similar results for linear fitness functions, but GPR has the slight advantage of converging faster.

Let us now turn to the function MULT(2). Combining $\bar{f}(t) = (1 + p(t))^2$ with equation (9), we obtain equation (8). For MULT(2) TPR and GPR lead to the same difference equation. One can show that in general for multiplicative functions ($m_0 m_3 = m_1 m_2$) TPR and GPR give the same gene frequencies.

For many loci the above analysis can easily be extended to fitness functions called *unitation functions*; their fitness values depend only on the number of 1's in the genotype. Again for simplicity, we consider only three loci. Let u_i denote the fitness of a genotype with i 1's. Under the assumption that $p_1(0) = p_2(0) = p_3(0)$, all marginal frequencies have the same value, which we denote by $p(t)$. Then for the marginal frequency $p(t+1) = c \cdot p(t)$ we obtain

$$c = \frac{u_1(1-p)^2 + 2u_2 p(1-p) + u_3 p^2}{u_0(1-p)^3 + 3u_1 p(1-p)^2 + 3u_2 p^2(1-p) + u_3 p^3},$$

where $p = p(t)$. If $c > 1$ the marginal frequency $p(t)$ increases, if $c < 1$ it decreases. As a specific example we analyze a 'deceptive' function of three loci; see Goldberg [1989]. Let the fitness values of this function be $u_0 = 28$, $u_1 = 26$, $u_3 = 30$. The global optimum is at 111, the local optimum at 000. The fitness value for u_2 can be varied. Depending on u_2 and the initial population, the genetic population will converge to 000 or 111. Take $u_2 = 0$ as an example. If $p_0 > 0.639$ the population will converge to 111. If $p_0 < 0.639$ the population will converge to 000. At $p_0 = 0.639$ we have $c = 1$. This point is an *unstable equilibrium point*.

Remark The analysis of unitation functions of three or more loci shows that a genetic algorithm using selection and recombination only is *not a global optimization* method. Depending on the frequency distribution of the genotypes and the fitness values, a genetic algorithm with infinite population size will

Table 6.2 Results for TPR and GPR for a bimodal function

t	REC	q_0	q_1	q_3	$\bar{f}$	Var
0	TPR	0.250	0.250	0.250	0.4975	0.2475
1	TPR	0.372	0.125	0.378	0.7463	0.1857
2	TPR	0.369	0.125	0.381	0.7463	0.1857
3	TPR	0.365	0.125	0.385	0.7464	0.1856
9	TPR	0.287	0.121	0.471	0.7556	0.1819
0	GPR	0.250	0.250	0.250	0.4975	0.2475
1	GPR	0.247	0.250	0.252	0.4975	0.2475
2	GPR	0.242	0.250	0.257	0.4977	0.2476
3	GPR	0.233	0.250	0.268	0.4983	0.2477
6	GPR	0.120	0.226	0.427	0.5457	0.2467
9	GPR	0.000	0.006	0.988	0.9883	0.0115

deterministically converge to one of the local optima. The equations derived in this chapter can be used to determine the optima to which the genetic algorithm will converge.

As a last example we will analyze a fitness function where the results of TPR and GPR are very different. For simplicity we take two loci. The fitness values are defined as $m_0 = 0.99$, $m_1 = 0$, $m_2 = 0$, and $m_3 = 1$.

Table 6.2 shows that GPR very slowly changes the gene frequencies at the beginning. In fact, if $m_0 = 1$ the population would stay in equilibrium. After three generations GPR changes the gene frequencies very quickly. In constrast, TPR dramatically changes the frequencies in the first generation. The population immediately goes to the equilibrium points for the symmetric fitness function $m_0 = m_3 = 1$, $m_1 = m_2 = 0$. It takes TPR a long time to leave this equilibrium point and march to the optimum.

Proportionate selection should not be used for a genetic algorithm; its selection intensity is far too low. Unfortunately, it is difficult to obtain the equations for the marginal gene frequencies of other selection methods. An approximate analysis can be performed for truncation selection by using the equation for the response to selection. Here one has to estimate the standard deviation of the phenotypes. Theorem 1 does this for the ONEMAX function; the theorem is exact for GPR because GPR leads to a binomial distribution. Our analysis shows that GPR converges for the ONEMAX function about 25% faster than TPR.

9 SELECTION AND MUTATION

The analysis of mutation is treated according to population size. First we consider small populations, then we look at large populations.

9.1 Mutation in small populations

The analysis of the mutation operator will first be done for an evolutionary algorithm with just one parent and one offspring. This algorithm can be called a random walk or *iterated hill climbing* if a hill-climbing strategy is used as well.

Iterated hill climbing (Random walk algorithm)

1. Generate a random initial string s.
2. If $hc \neq \emptyset$, do the hill-climbing strategy hc, giving a modified s'.
   If $f(s') \geqslant f(s)$ then $s := s'$.
3. Mutate each bit of s with probability m.
4. If not finished, return to Step 2.

Distinct from a *multistart* algorithm, iterated hill climbing is one of the simplest optimization methods and will locate the global optima with a probability greater than zero. The new configuration is generated by a random perturbation of the previously found local optimum. If the perturbation is too weak, the new configuration will be close to the local optimum. If the perturbation is too strong, the new configuration will be far away from the current search. In a multistart algorithm each initial configuration is randomly generated.

The importance of using a hill-climbing method to speed up the search has been shown by Mühlenbein [1991a, 1992b] by a Markov chain analysis. In order to compare mutation with crossover we just cite the following theorem from Mühlenbein [1992b].

Theorem 4 *Let the initial string have $n \cdot p_0$ bits correct. Then for* ONEMAX *of size $n \gg 0$ and a mutation rate of $m = j/n$ with $j \ll n$, the expected number of trials $\langle T(m) \rangle$ to reach the optimum is approximately given by*

$$\langle T(m) \rangle \approx e^j \frac{n}{j} \sum_{i=1}^{(1-p_0)n} \frac{1}{i}.$$

Remark For $0 \leqslant p_0 < 0.9$ the above equation can be approximated by

$$\langle T(j/n) \rangle \approx e^j \cdot (n/j) \cdot \ln((1 - p_0)n). \tag{11}$$

The number of trials T needed to reach the optimum increases exponentially in j compared to the optimal mutation rate of $1/n$. A small absolute change of the mutation rate may have a dramatic impact. For a fixed mutation rate we get the following result.

Corollary *The optimal mutation rate for unimodal discrete functions is inversely proportional to the size of the chromosome.*

This important result has been independently discovered several times. The implications of this result to biology and to evolutionary algorithms were first investigated by Bremermann, Roghson & Salaff [1966].

With a mutation rate of $m = 1/n$ the string will not be changed at all with probability $P = (1 - 1/n)^n \approx e^{-1}$. But if the population consists of one string only,

Table 6.3 $\langle T \rangle$ and speedup Sp. N offspring from one parent (ONEMAX (128))

N	2	4	8	16	32	64	128	256	512	1024
$\langle T(m) \rangle$	750	400	200	115	71	48	37	30	26	23
Sp	2.0	3.75	7.5	13.4	21	31	41	50	58	65

it makes no sense not to change the string. Therefore the mutation rule for small populations is as follows. *Change each bit with probability* $m = 1/n$. *If the chromosome is not mutated at all, randomly change one bit.*

We have confirmed equation (11) by intensive simulations [Mühlenbein, 1991a]. Recently Bäck [1993] has shown that $\langle T(m) \rangle$ can be only marginally reduced if a theoretically optimal *variable* mutation rate is used. This variable rate depends on the number of bits yet to be corrected, a result predicted by Mühlenbein [1992b]. Mutation spends most of the time in adjusting the very last bits. But in this region the optimal mutation rate is obviously $m = 1/n$.

Theorem 4 cannot easily be extended to a larger number of offspring. Therefore we will only discuss this problem qualitatively by simulations. Some results are displayed in Table 6.3. One clearly observes the law of diminishing returns. Increasing the population size N reduces $\langle T(m) \rangle$, the number of generations needed to find the optimum, less and less. Mutation is most efficient with a small number of offspring.

The speedup Sp shows how much faster the solution is obtained with a larger number of offspring N. It is defined as the ratio $\langle T(1) \rangle / \langle T(N) \rangle$. The speedup is almost linear for small N and seems to slow down to a logarithmic function. This indicates that mutation is not an efficient search in a large population. We will show in the next section that smaller selection intensities give still worse results.

9.2 Mutation in large populations

In order to analyze mutation in large populations, one has to extend the equation for the response to selection, and the concept of heritability is not of much use. The following theorem is one possibility to extend the concept of the response to selection.

Theorem 5 *Let* s_t *be the probability of a mutation success, imp the average improvement of a successful mutation. Let* f_t *be the probability that the offspring is worse than the parent, red the average reduction of the fitness. Then the response to selection for small mutations in large populations is given by*

$$R(t) = S(t) + s_t \cdot imp - f_t \cdot red.$$

Proof Let $\bar{f}_s(t)$ be the average of the selected parents. Then

$$\bar{f}(t+1) = \bar{f}_s(t) + s_t \cdot imp - f_t \cdot red.$$

Subtracting $\bar{f}(t)$ from both sides of the equation we obtain the theorem. □

The response-to-selection equation for mutation contains no *heritability*. Instead there is an offset, defined by the difference of the probabilities of getting better or worse. The importance of s_t and f_t has been independently discovered by Schaffer & Eshelman [1991]. They did not use the difference of the probabilities, but the ratio which they called the *safety factor*:

$$safety = \frac{s_t}{f_t}.$$

The following empirical law has been found by simulations [Mühlenbein & Schlierkamp-Voosen, 1994a].

Empirical Law 1 *For* ONEMAX *of size n, a truncation threshold of* $T = 50\%$, *a mutation rate of* $m = 1/n$, *and* $n \gg 1$ *the response to selection of a large population changing by mutation only is approximately*

$$R(t) = 1 + (1 - p(t))e^{-p(t)} - p(t)e^{-(1-p(t))}. \tag{12}$$

Equation (12) defines a difference equation for $p(t+1)$. We did not succeed in solving it analytically. We have found that the following linear approximation gives almost the same results.

Empirical Law 2 *Under the assumptions of Empirical Law 1 the response to selection can be approximated by*

$$R(t) = 2 - 2p(t). \tag{13}$$

The number of generations that $p(t) = 1 - \varepsilon$ *is reached is given by*

$$GEN_{1-\varepsilon} \approx \left(\frac{n}{2}\right) \ln \frac{1 - p_0}{\varepsilon}.$$

By comparing Theorem 2 with Empirical Law 2, one discovers how mutation with 50% truncation selection is twice as efficient as proportionate selection with crossover. This shows once more the ineffciency of proportionate selection.

Next we will compare the theoretical results with simulations. Figure 6.2 shows the development of the mean fitness. The simulations have been done with two pop-sizes ($N = 1024, 64$) and two mutation rates ($m = 1/n$, $4/n$). The agreement between the theory and the simulation is very good. The large population and the small population have almost the same evolution of mean fitness. This demonstrates that a large population is inefficient for mutation. Furthermore, Figure 6.2 shows that mutation spends most of its time in getting the last bits correct.

A large mutation rate has an interesting effect. The mean fitness increases faster at the beginning but it never finds the optimum. This might suggest the use of a variable mutation rate. We have already mentioned in the previous section there is a caveat: any increase in performance will be rather small.

The efficiency of mutation in large populations can be increased by very strong selection. It is obvious that for unimodal functions one should only select the best individual as parent.

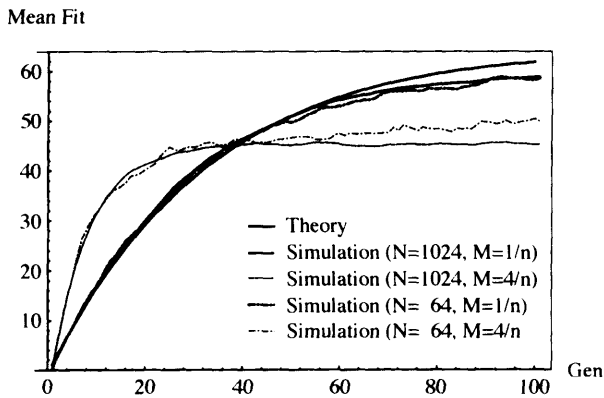

Figure 6.2 Theory (13) and simulation results for various population sizes N and mutation probabilities M

The major results of this section may be summarized as follows. The mutation operator in large populations is not effective. It needs very strong selection. Furthermore, the efficiency of the mutation operator critically depends on the mutation rate.

10 SUMMARY OF MAJOR RESULTS

In the previous sections we have presented the framework for analyzing evolutionary algorithms. We now give a summary along with additional results proved by Mühlenbein & Schlierkamp-Voosen [1993, 1994a, 1994b]. Most of the results are exact for gene pool recombination (GPR) and only approximate for two-parent mating (TPR).

A fascinating phenomenon of genetic populations is called *genetic drift*. Any finite genetic population of size N will converge to a single genotype, even if selection is not applied. The expected number of generations until convergence GEN_e is surprisingly low.

Let n denote the number of genes, N the size of the population. Then the expected number of generations until equilibrium is given [Asoh & Mühlenbein, 1994b] as

$$\mathbb{E}(GEN_e) \approx 1.4\, N(0.5 \ln n + 1).$$

This equation is valid for random mating with recombination but without selection and mutation. Note that the $\mathbb{E}(GEN_e)$ scales linearly in N and only logarithmically in n. Genetic drift is the reason for the surprising fact that small selection intensities *decrease* the probability to find the optimum.

We now turn to truncation selection. If the size N of the population is larger than the *critical pop-size* N^* – the minimum population size to converge to the optimum with high probability – we have for the expected number of generations

until convergence that

$$GEN_e \approx \left(\frac{\pi}{2} - \arcsin(2p_0 - 1)\right)\frac{\sqrt{n}}{I}.$$

Here p_0 is the proportion of 1's in the initial population. Note that GEN_e is independent of N. The estimation of the critical population size N^* is very difficult. Mühlenbein & Schlierkamp-Voosen [1994b] conjecture that

$$N^* \approx \sqrt{n}\cdot\ln(n)\cdot f_1(p_0)f_2(I).$$

Proportionate selection as used by the simple genetic algorithm (GA) [Goldberg, 1989] selects too weakly when the variance of the population becomes small. The expected number of generations $GEN_{1-1/n}$ until the favorable allele is distributed in the population with probability of $1 - 1/n$ is given by

$$GEN_{1-1/n} \approx n\ln[n(1 - p_0)].$$

$GEN_{1-1/n}$ is much larger than the corresponding value for truncation selection.

All the above equations are valid for a mutation rate of 0. We now turn to populations using only mutation. The most important result concerns the mutation rate.

Rule of thumb *The mutation rate $m = 1/n$, where n is the size of the chromosome, is almost optimal.*

For the above mutation rate the expected number of generations GEN_{opt} until the optimum is found has been computed for the (1 + 1)-strategy (one parent, one offspring; the better of the two survives):

$$GEN_{opt} \approx en\ln[n(1 - p_0)].$$

Mutation in a large population is inefficient. The asymptotic scaling of GEN_{opt} is independent of the pop-size N. It stays at $O(n\ln(n))$. For very large pop-sizes GEN_{opt} is given by

$$GEN_{1-1/n} \approx \left(\frac{n}{2}\right)\ln[n(1 - p_0)].$$

This equation is valid for a large population and a truncation selection threshold of $T = 0.5$. The above value is about half the value of proportionate selection.

These theorems show that, for binary representations, populations using either recombination or mutation are able to locate the optimum. If $p_0 = 0.5$, i.e., half of the bits are correct in the initial population, the asymptotic order of the number of trials needed (FE_{opt}), seems to be the same, namely $O(n\ln n)$. For recombination this number is obtained by multiplying GEN_e with the critical pop-size N^*, so it is difficult to determine which of the two operators is more efficient. The comparison needs an exact expression for N^*, which has not yet been obtained. But one can easily make a qualitative comparison. The major difference between mutation and recombination is their dependence on p_0, the percentage of the desired allele in the initial population.

Let us take $p_0 = 1 - 1/n$ as example; on average, only one bit is wrong. Mutation will need about $O(n)$ trials to change the incorrect bit. Uniform crossover of two strings, each with one bit wrong, will generate the optimum string with probability 1/4, independent of the size of the problem. Thus recombination is much more efficient than mutation. But the determination of the exact N^* is also difficult in this simple case. It will need on average 4 trials to generate the optimum. But the probability that a pop-size of 4 will not generate the optimum is $0.75^4 = 0.31$. It needs 16 trials in order to obtain the optimum with 99% probability.

If we take $p_0 = 1/n$ the situation is reversed. Now mutation is much more efficient than recombination, which needs a huge pop-size in order to locate the optimum. Recombination has too few building blocks to generate better offspring.

Most of our analytical results have been derived under the assumption of additive genetic effects. This theory explains the behavior of the most important evolutionary forces. It plays a similar role for the breeder genetic algorithm (BGA) as the 'ideal gas' theory for the thermodynamics. There exists no *ideal gas* in reality, but the ideal gas theory gives much insight into the overall behavior of gases. In order to understand evolutionary algorithms in more complex fitness landscapes, we have to extend the theory by using more advanced statistical methods. This is the topic of Section 12. But first we consider how to combine search by mutation and search by recombination.

11 MUTATION AND RECOMBINATION

Having analyzed recombination and mutation individually, we now compare them, starting with Figure 6.3. This compares recombination using truncation selection and recombination using proportionate selection with mutation using truncation selection. The fitness function is ONEMAX (64). The initial population was generated with $p_0 = 1/64$. As predicted by the theory, the mean fitness of

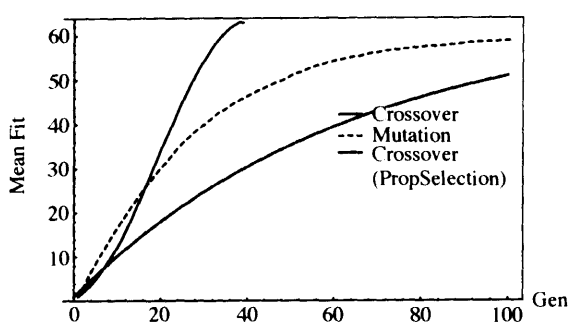

Figure 6.3 Comparison of crossover and mutation

the population with mutation is best at the beginning. Recombination with truncation selection is the best performer when $p(t) \geqslant 0.5$. The simulations confirm that recombination with proportionate selection is twice as inefficient as mutation with a truncation threshold of 50%.

What is the best way to combine mutation and recombination? This can be done by at least two different methods. One can try to use both operators in a single evolutionary algorithm with an almost optimal parameter setting. This means that a good mutation rate and a good population size have to be predicted. This method is used in the breeder genetic algorithm. A good mutation rate is constantly applied. The variance of the population remains high enough for recombination to be effective. A different approach is used by Eshelman [1991] in the CHC algorithm. CHC is based on recombination. If the variance of the population is below a certain threshold, the population is thoroughly changed by applying mutation vigorously. This event gives recombination the chance for further improvements. The success of this method depends on the right amount of change. If too much is changed, this will be just a new start of the algorithm. If the changes are too small, the population will stay in equilibrium.

We believe that difficult optimization problems lend themselves to a different method. If optimal parameter settings or strategies are not known beforehand, one can try a competition between subpopulations using different strategies. Such a competition can be performed on different levels: the level of the individuals, the level of subpopulations, or the level of populations. For evolution strategies, Bäck, Hoffmeister & Schwefel [1991] have implemented the adaptation of strategy parameters on the level of the individual. The strategy parameters of the best individuals are recombined, giving the new step-size for the mutation. Herdy [1992] uses competition on the population level; whole populations are evaluated at certain intervals. Successful populations proliferate and so do their strategies; poorly performing populations die out and so do their strategies.

The adaptation of the breeder genetic algorithm lies between these two extreme cases. The competition occurs between subpopulations. It uses a *quality criterion* to rate the groups, a *gain criterion* to reward or punish the groups, an *evaluation interval*, and a *migration interval*. The evaluation interval gives each strategy the chance to demonstrate its performance in a certain time window. The best individuals from successful groups occasionally migrate to groups that are performing badly; this gives the poorer groups a better chance to compete. The sizes of the subgroups have a lower limit, therefore no strategy is lost. The rationale behind this algorithm has been described by Schlierkamp-Voosen & Mühlenbein [1994].

12 STATISTICS AND GENETICS

In this section we present two methods for estimating *heritability*. The first uses the concept of *regression of offspring to parent* and the second uses the concept of decomposing the *genetic variance*. Both methods have been of great importance

in the development of statistics and population genetics. We will use a notation that is more common in statistics.

The first theorem connects the realized heritability $b_t = R(t)/S(t)$ with the regression coefficient between *midparent* and offspring. Let f_i, f_j be the phenotypic values of parents i and j. Then

$$\bar{f}_{i,j} = \frac{f_i + f_j}{2}$$

is called the midparent value. Let the stochastic variable $\bar{F}$ denote the midparent value.

Theorem 6 *Let $F(t) = (f_1, \ldots, f_N)$ be the fitness values of the population at generation t, where f_i denotes the fitness value of individual i. Assume that an offspring generation $O(t)$ is created by random mating, without selection. If the regression equation*

$$o_{ij}(t) = a(t) + b_{\bar{F}O}(t) \cdot \frac{f_i + f_j}{2} + \varepsilon_{ij}$$

with

$$\mathbb{E}(\varepsilon_{ij}) = 0$$

is valid, where o_{ij} is the fitness value of the offspring of i and j, then

$$b_{\bar{F}O}(t) \approx b_t.$$

Proof From the regression equation we obtain for the expected averages

$$\mathbb{E}(O(t)) = a(t) + b_{\bar{F}O}(t)\bar{f}(t).$$

Because the offspring generation is created by random mating without selection, the expected average fitness remains constant

$$\mathbb{E}(O(t)) = \bar{f}(t).$$

Let us now select a subset as parents. The parents will be randomly mated, producing the offspring generation. If the subset is large enough, we may use the regression equation and obtain for the averages

$$\bar{f}(t+1) = a(t) + b_{\bar{F}O}(t)\bar{f}_s(t).$$

Here $\bar{f}(t+1)$ is the average fitness of the offspring generation produced by the selected parents. Subtracting the above equations we obtain

$$\bar{f}(t+1) - \bar{f}(t) = b_{\bar{F}O}(t)[\bar{f}_s(t) - \bar{f}(t)].$$

This proves $b_{\bar{F}O}(t) = b_t$. □

The problem of computing a good regression coefficient is solved by the theorem of Gauss–Markov. We just cite the theorem. The proof can be found in Rao [1973].

Theorem 7 *A good estimate for the regression coefficient of midparent and offspring is given by*

$$b_{\bar{F}O}(t) = \frac{\operatorname{cov}(O(t),\ \bar{F}(t))}{\operatorname{var}\ (\bar{F}(t))}. \tag{14}$$

The covariance of O and $\bar{F}$ is defined by

$$\operatorname{cov}(O(t), \bar{F}(t)) = \frac{1}{N}\sum_{i,j}(o_{i,j} - \operatorname{av}(O(t)))\cdot(\bar{f}_{i,j} - \operatorname{av}(\bar{F}(t))),$$

where av denotes the average and var the variance. Closely related to the regression coefficient is the correlation coefficient $\operatorname{cor}(\bar{F}, O)$. It is given by

$$\operatorname{cor}(\bar{F}(t), O(t)) = b_{\bar{F}O}(t)\cdot\left(\frac{\operatorname{var}(\bar{F}(t))}{\operatorname{var}(O(t))}\right)^{1/2}.$$

Theorem 7 enables us to estimate the heritability by a second method. It works as follows. For a large sample population F, offspring are created by random mating. Then the regression coefficient $b_{\bar{F}O}$ is computed by equation (14). This estimate is called *regression heritability* [Falconer, 1981]. The relation between realized heritability and regression heritability is one of the most difficult aspects of population genetics. The next theorem shows the connection between *midparent* and *parent* regression.

Theorem 8 *Midparent and parent regression are connected by*

$$b_{FO}(t) = 0.5\, b_{\bar{F}O}(t), \quad \operatorname{cor}(F(t), O(t)) = \sqrt{\frac{1}{2}}\operatorname{cor}(\bar{F}(t), O(t)).$$

Proof We have

$$\operatorname{cov}(O(t), \bar{F}(t)) = \operatorname{cov}(O(t), F(t)),$$
$$\operatorname{var}(\bar{F}(t)) = 0.5\operatorname{var}(F(t)).$$

From (14) the theorem is obtained. □

We now describe a method for estimating the covariance. This method connects a microscopic genetic chance model and the macroscopic phenotypic covariance. It is restricted to discrete genes. We only give the necessary definitions and the fundamental theorem; the proof may be found in Asoh & Mühlenbein [1994a].

Let a haploid chromosome with n binary genes x_i be given. Let $\mathbf{x} = (x_1, x_2, \ldots x_n)$ be a genotype, $f(\mathbf{x})$ its fitness. Let the genetic chance model be defined by *uniform crossover*. This model can be considered as Mendel's chance model restricted to haploid chromosomes. We will decompose the fitness value $f(\mathbf{x})$ recursively into an additive part and interaction parts. Let $p(\mathbf{x})$ denote the probability of $\mathbf{x}$, $p(\mathbf{x}\,|\,x_i)$ the conditional probability of $\mathbf{x}$ given x_i. First we extract the average,

$$f(\mathbf{x}) = \operatorname{av}(f) + r_0(\mathbf{x}).$$

Then we extract the first-order (additive) part from the residual $r_0(\mathbf{x})$:

$$r_0(\mathbf{x}) = \sum_{i=1}^{n} f_{(i)}(x_i) + r_1(\mathbf{x}),$$

where $f_{(i)}(x_i)$ are given by

$$f_{(i)}(x_i) = \sum_{\mathbf{x}|x_i} p(\mathbf{x}|x_i) r_0(\mathbf{x}) = \sum_{\mathbf{x}|x_i} p(\mathbf{x}|x_i) f(\mathbf{x}) - \mathrm{av}(f).$$

Here $\sum_{\mathbf{x}|x_i}$ means that the ith locus is fixed to the value x_i. The $f_{(i)}(x_i)$ minimize the quadratic error $\sum_{\mathbf{x}} p(\mathbf{x}) r_1(\mathbf{x})^2$.

If $r_1(\mathbf{x}) \not\equiv 0$, we can proceed further to extract the second-order terms from $r_1(\mathbf{x})$:

$$r_1(\mathbf{x}) = \sum_{\substack{(i,j)\\ i<j}} f_{(i,j)}(x_i, x_j) + r_2(\mathbf{x}),$$

where

$$\begin{aligned} f_{(i,j)}(x_i, x_j) &= \sum_{\mathbf{x}|x_i,x_j} p(\mathbf{x}|x_i, x_j) r_1(\mathbf{x}) \\ &= \sum_{\mathbf{x}|x_i,x_j} p(\mathbf{x}|x_i, x_j) f(\mathbf{x}) - f_{(i)}(x_i) - f_{(j)}(x_j). \end{aligned}$$

If we have n loci, we can iterate this procedure $n-1$ times recursively and finally we get the decomposition of f as

$$\begin{aligned} f(\mathbf{x}) = \bar{f} &+ \sum_i f_{(i)}(x_i) + \sum_{(i,j)} f_{(i,j)}(x_i, x_j) + \cdots \\ &+ \sum_{\substack{(i_1,\ldots,i_{n-1})\\ i_1<\cdots<i_{n-1}}} f_{(i_1,\ldots,i_{n-1})}(x_{i_1}, \ldots, x_{i_{n-1}}) + r_{n-1}(\mathbf{x}). \end{aligned}$$

Let V_k for $k = 1$ to $n-1$ be defined as

$$V_k = \sum_{\substack{(i_1,\ldots,i_k)\\ i_1<\cdots<i_k}} \sum_{x_{i_1},\ldots,x_{i_k}} p(x_{i_1}, \ldots, x_{i_k}) f_{(i_1,\ldots,i_k)}(x_{i_1}, \ldots, x_{i_k})^2,$$

and

$$V_n = \sum_{\mathbf{x}} p(\mathbf{x}) r_{n-1}(\mathbf{x})^2,$$

where $p(x_{i_1}, \ldots, x_{i_k})$ are the marginal frequencies. Now the fundamental theorem of classical population genetics can be formulated.

Theorem 9 *Let the population be linkage equilibrium, i.e.,*

$$p(\mathbf{x}) = \prod_{i=1}^{n} p_i(x_i).$$

Then the variance of the population is given by

$$\mathrm{var}(F) = V_1 + V_2 + \cdots + V_{n-1} + V_n.$$

The covariance of midparent and offspring can be computed from

$$\operatorname{cov}(\bar{F}, O) = \frac{1}{2}V_1 + \frac{1}{4}V_2 + \cdots \frac{1}{2^n}V_n = \sum_{k=1}^{n} \frac{1}{2^k} V_k.$$

From Theorems 7 and 9 we obtain the following corollary.

Corollary 10 *If the fitness is additive, i.e.,* $f(\mathbf{x}) = \sum_i f_i(x_i)$, *then*

$$\operatorname{cor}(\bar{F}, O) = \sqrt{1/2} \text{ and } b_{\bar{F}O} = 1.$$

The above theorem plays an important role in the science of breeding. Breeders conjecture that the *additive genetic variance* V_1 is the most important factor of the regression heritability. The higher-order interactions contribute much less to the heritability; therefore they can be neglected. Unfortunately, our simulations have shown that the assumption of linkage equilibrium is seldom fulfilled for genetic algorithms, so the variance decomposition method gives no good predictions for the heritability. Section 13 shows how the regression technique can be used to control and guide the breeder genetic algorithm.

13 APPLICATIONS OF THE THEORY

It is known from statistics and population genetics that the regression coefficient should be a reliable estimate for heritability in the case of continuous fitness functions and large populations. Therefore the first example is the minimization of the hypersphere. The breeder genetic algorithm (BGA) for continuous functions has been described by Mühlenbein & Schlierkamp-Voosen [1994b]. It uses a floating point representation. Figure 6.4 shows scatter diagrams of midparent and offspring at generations 1 and 30, using only discrete recombination, no mutation. The whole population is moving towards the global minimum, zero in

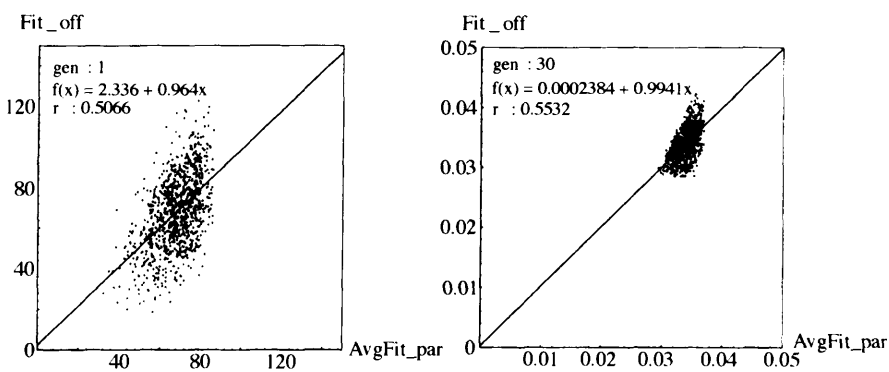

Figure 6.4 Scatter diagrams using discrete recombination only ($N = 1024$, $T = 0.5$): (a) generation 1, $f(x) = 2.336 + 0.964x$, $r = 0.5066$ and (b) generation 30, $f(x) = 0.000\,2384 + 0.9941x$, $r = 0.5532$

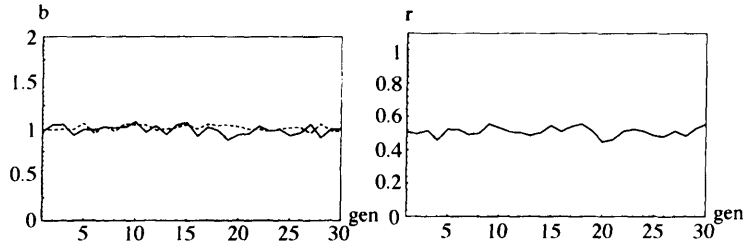

Figure 6.5 Hypersphere ($N = 1024$, $T = 0.5$): (a) heritability with (—) regression coefficient and (---) $R(t)/S(t)$; (b) correlation coefficient

this example. The regression coefficient in both diagrams is almost exactly one, as predicted by the theory.

Figure 6.5 shows the numerical values of the two different estimates for the heritability ($R(t)/S(t)$ and the regression coefficient). Both estimates oscillate around 1 as predicted. The correlation coefficient is about 0.5. This is less than the maximum possible value, which is $\sqrt{0.5}$. The reason for this difference is the selection. The selection reduces the variance of the parents and therefore the correlation coefficient.

A simulation was run without selection, and the results confirm the theory. In this case the $R(t)/S(t)$ estimator cannot be used because $S(t)$ is 0. The regression coefficient can be computed as usual and remains 1. Furthermore, the correlation coefficient is about $\sqrt{0.5}$, as predicted. These results are not restricted to simple unimodal functions. Our next example is Schwefel's function F_7, which is highly multimodal:

$$F_7 = \sum_{1}^{n} -x_i \sin(\sqrt{|x_i|}), \quad -500 \leqslant x_i \leqslant 500.$$

The theory predicts that the multimodality of this function can be considered more or less as noise for the BGA. It should have no major influence on the regression coefficient. Indeed, with random mating, the regression coefficient is 1 and the correlation coefficient between midparent and parent is about $\sqrt{0.5}$, just as for the hypersphere. Figure 6.6 shows a real BGA simulation run with selection, recombination, *and* mutation. One clearly observes that the search is first driven by recombination, then by mutation. From generation 17 on, the regression coefficient substantially differs from the ratio estimator $R(t)/S(t)$. Now the search is mainly driven by the random operator mutation.

Next we turn to binary functions. The following simple functions will be used:

- ONEMAX (n)
- PLATEAU (20, l)
- DECEP (10, 3)

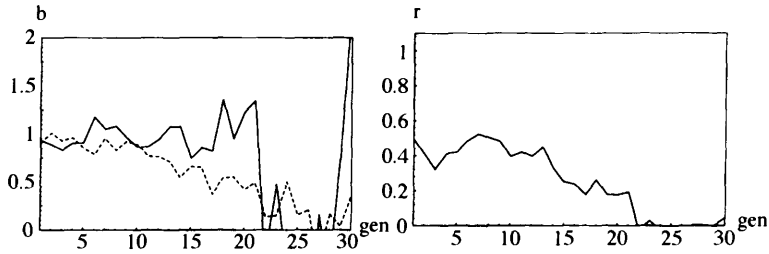

Figure 6.6 Mutation and recombination ($N = 256$): (a) heritability with (—) regression coefficient and (---) ratio estimator; (b) correlation coefficient; the regression coefficient and the ratio estimator are almost equal at the beginning, then the ratio estimator goes to zero whereas the regression coefficient remains high till generation 22

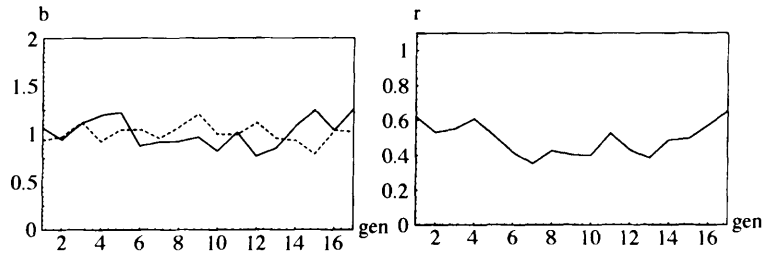

Figure 6.7 ONEMAX (64) with recombination only ($N = 128$, $T = 0.5$): (a) heritability with (—) regression coefficient and (---) $R(t)/S(t)$: (b) correlation coefficient

PLATEAU $(20, l)$ has a string length n of $20l$. An increase in fitness is allocated only if l consecutive bits starting at loci $1, l+1, 2l+1, \ldots$ are 1's. In each case the fitness is increased by 1. DECEP (10, 3) is the deceptive function defined by Goldberg [1989].

Figure 6.7 shows the results of a BGA run for ONEMAX (64) with a truncation threshold of $T = 0.5$ and uniform crossover, but without mutation. The two heritability estimates coincide fairly well. They are about 1, as predicted. The correlation coefficient is about 0.5 until generation 14. This is less than the correlation without selection, which is $\sqrt{0.5}$. The correlation coefficient increases at the end of the run. This behavior indicates that the genotypes of the selected parents are becoming very similar. Therefore the offspring are very similar to both parents.

The next examples are PLATEAU (20, 3) and PLATEAU (20, 5). PLATEAU (20, 5) has a plateau of size 5, so it is more difficult to optimize. Without selection, the regression coefficients for the two functions are about 0.7 and 0.4, the correlation coefficients are about 0.5 and 0.3. In Figure 6.8 we have used a truncation threshold of $T = 0.5$. For both functions the regression coefficients are substantially higher than without selection, indicating that selection is very

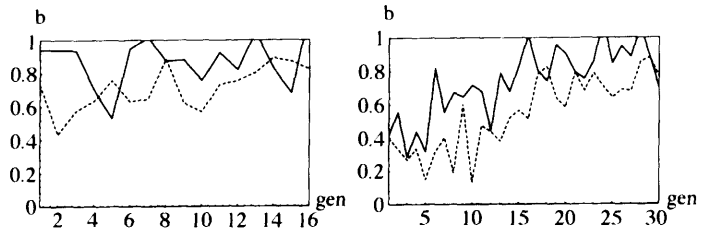

Figure 6.8 Heritability estimates for (a) PLATEAU (20, 3) and (b) PLATEAU (20, 5): ( ) regression coefficient and (---) $R(t)/S(t)$

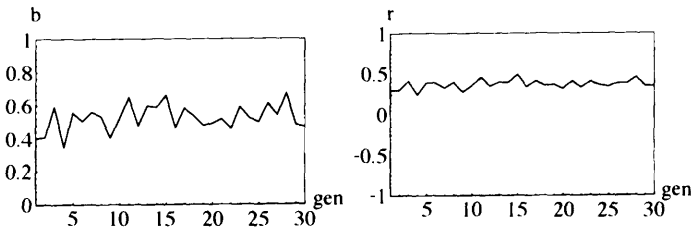

Figure 6.9 DECEP (10, 3) with recombination but no selection ($N = 256$): (a) heritability and (b) correlation coefficient

effective for this fitness function. But note that the realized heritability $R(t)/S(t)$ is considerably smaller than the regression coefficient. For PLATEAU (20, 5) it substantially increases during the run.

Finally, Figure 6.9 shows the deceptive function DECEP (10, 3). Without selection, the regression coefficient is about 0.5 and the correlation coefficient about 0.35. The behavior radically changes with selection. Both the regression coefficient and the ratio estimator become erratic, and half of the time they are negative. This shows that selection and recombination work against each other. It confirms our earlier statement that genetic algorithms are not global optimizers.

To summarize, many continuous fitness functions will have a regression coefficient of 1, the maximum possible. For binary functions, the regression coefficient and the realized heritability give useful information about the complexity of the fitness landscape and how to guide the search of the breeder genetic algorithm.

14 CONCLUSION

Evolutionary algorithms have been applied quite successfully to several difficult optimization problems. One of the their major advantages is the fact they can be intuitively understood. But this can become a disadvantage when researchers

start to argue intuitively, without mathematical rigor. Understanding why evolutionary algorithms work at all is almost as difficult as understanding natural evolution itself. The major difficulty lies in the fact that the algorithms combine two different search strategies, a random search by mutation and a biased search by recombination of the strings contained in the population.

We have presented a theoretical analysis based on methods developed in population genetics. It explains the behavior of selection, mutation, and recombination for problems that can be described by binary strings. The major part of the analysis deals with macroscopic variables. The genetic chance model is used only for estimating the heritability. The equation for the response to selection can be used for any representation. Only the results concerning the decomposition of the genetic variance are restricted to binary genes. The analysis has been used to design new genetic operators that do not have a counterpart in nature. Especially interesting is gene pool recombination.

Applying genetic algorithms to general combinatorial optimization problems leads to the *genetic representation problem*. This means that a representation of the problem has to be found in order for mutation and recombination to create feasible offspring with high probability. This leads to unique and problem-specific mutation and recombination operators, for which we have presented a general design principle: maximize the product of heritability and variance.

Another solution to this problem may be a universal genetic representation. This would make genetic algorithms even more similar to natural evolution. All natural systems use the same genetic mechanism, based on the interaction between the nucleic acids DNA and RNA.

Local Search in Combinatorial Optimization
Edited by E. Aarts and J. K. Lenstra
© 1997 John Wiley & Sons Ltd.

7

Artificial neural networks

Carsten Peterson, Bo Söderberg
University of Lund, Lund

1 INTRODUCTION 173
2 ARTIFICIAL NEURAL NETWORKS 174
2.1 Background 174
2.2 Basic ANN ingredients 175
2.3 Feedforward networks 177
2.4 Feedback networks 178
3 PURE ANN APPROACH TO OPTIMIZATION: BINARY CASE 181
3.1 Basic encoding 181
3.2 Minimizing E 181
3.3 The mean field equations 182
3.4 The mean field neural approach 183
3.5 Analysis of the dynamics 184
3.6 The graph bisection problem 185
4 OPTIMIZATION WITH POTTS NEURAL NETWORKS 186
4.1 Potts spins 187
4.2 Potts mean field equations 188
4.3 Mean field dynamics 189
4.4 A black-box procedure 191
4.5 The graph partitioning problem 191
4.6 The traveling salesman problem 193
4.7 Scheduling problems 194
5 OPTIMIZATION WITH INEQUALITY CONSTRAINTS 198
5.1 Dealing with inequality constraints 199
5.2 The knapsack problem 199
6 DEFORMABLE TEMPLATES 204
6.1 The traveling salesman problem 204
6.2 Track finding 207
7 ROTOR NEURONS 210
7.1 Mean field rotor neurons 210
7.2 Applications 211
8 SUMMARY AND OUTLOOK 211

1 INTRODUCTION

The use of artificial neural networks (ANNs) to find good solutions to combinatorial optimization problems [Hopfield & Tank, 1985; Peterson & Söderberg, 1989; Durbin & Willshaw, 1987; Peterson, 1990a] has recently caught some

attention. Whereas the use of ANNs for pattern recognition and prediction problems constitutes a nonlinear extension of conventional linear interpolation/extrapolation methods, ANNs in the optimization domain really bring something new to the table. In contrast to existing search and heuristics methods, the ANN approach does not fully or partly explore the different possible configurations. Rather it 'feels' its way in a fuzzy manner towards good solutions. This is done in a way that allows for a statistical interpretation of the results. The ANN approach is therefore conceptionally and technically very different from conventional approaches. Two basic steps are involved when using ANNs to find good solutions to combinatorial optimization problems:

- Formulate the problem as the minimization of a feedback ANN energy function $E(s_1, \ldots, s_N)$, where the neurons s_i encode possible solutions.
- Find an approximate solution by iteratively solving the corresponding mean field (MF) equations.

This procedure often produces high-quality solutions, as will be demonstrated below. The neural approach has the additional advantage that the MF equations are isomorphic to VLSI (very large scale integration) RC equations, which makes hardware implementations straightforward. It is rare to have such tight bonds between algorithms and hardware.

This chapter aims to give a self-contained introduction to the use of ANNs for finding good approximate solutions to combinatorial optimization problems. Each theoretical description is illustrated by one or more examples. The chapter is organized as follows. Section 2 gives a general introduction to both feedforward and feedback networks. Section 3 discusses how to map an optimization problems onto a system of binary (Ising) spins, derives the corresponding mean field (MF) equations, and analyzes the resulting neural dynamics. The approach is illustrated with the graph bisection problem. An analogous path is followed in Section 4 for the more general multistate (Potts) neurons. Here the examples are graph partitioning, the traveling salesman, and scheduling problems. Optimization problems with inequality constraints require special care, dealt with in Section 5. An alternative procedure for low-dimensional geometrical problems, the deformable templates approach, is discussed in Section 6. This method is illustrated with the traveling salesman and track finding problems. In Section 7 we discuss the generalization to rotor neurons in order to deal with the optimization of systems with angular variables. Section 8 contains a brief summary and outlook.

2 ARTIFICIAL NEURAL NETWORKS

2.1 Background

The introduction of artificial neural networks was inspired by the structure of biological neural networks and their way of encoding and solving problems. The human brain contains approximately 10^{12} *neurons*. They can be of many different types, but most of them have the same general structure. The *cell body* or *soma*

receives electric input signals to the *dendrites*, signals carried by ions. The interior of the cell body is negatively charged against a surrounding *extracellular fluid*. Signals arriving at the dendrites depolarize the resting potential, enabling Na^+ ions to enter the cell through the membrane, producing an electric discharge from the neuron – the neuron fires. The accumulated effect of several simultaneous signals arriving at the dendrites is usually almost linearly additive, whereas the resulting output is a strongly nonlinear, all-or-none type process. The discharge propagates along the *axon* to a *synaptic junction*, where *neurotransmitters* travel across a *synaptic cleft* and reach the dendrites of the postsynaptic neuron. A synapse that repeatedly triggers the activation of a postsynaptic neuron will grow in strength; others will gradually weaken. This *plasticity*, which is known as the *Hebb rule*, plays a key part in *learning*.

The *connectivity* (number of neurons connected to a neuron) varies from ~ 1 to $\sim 10^5$. For the *cerebral cortex* $\sim 10^3$ is an average. This corresponds to $\sim 10^{15}$ synapses per brain. Synapses can be either *excitatory* or *inhibitory* of varying strength. In the simplified binary case of just two states per synapse the brain thus has $\sim 2^{10^{15}}$ possible configurations! The neural network consequently stands in sharp contrast to a von Neumann computer both with respect to architecture and functionality.

The von Neumann computer was originally developed for 'heavy duty' numerical computing, but has later also turned out to be profitable for data handling, word processing and the like. However, when it comes to matching the vertebrate brain in terms of performing 'human' tasks, it has very strong limitations. There are therefore strong reasons to design an architecture and an algorithm that show more resemblance to the vertebrate brain.

The philosophy of the ANN approach is to abstract some key biological ingredients from which to construct simple mathematical models that exhibit most of the appealing features we have just considered; for a general textbook on ANN see Hertz, Krogh & Palmer [1991]. In physics one has good experience of model building out of major abstractions. For example, details of individual atoms in a solid can be lumped into effective 'spin' variables in such a way that a good description of *collective* phenomena (phase transitions, etc.) is obtained. It is indeed the collective behavior of the neurons that is interesting from the point of view of intelligent data processing.

2.2 Basic ANN ingredients

The basic computational entities of an ANN are the neurons v_i, which normally can take real values within the interval $[0, 1]$ (or $[-1, 1]$); $i = 1, 2, \ldots, N$ is an index labeling the N individual neurons. Sometimes even simpler, discrete neurons s_i are used, with $s_i \in \{0, 1\}$ (or $\{-1, 1\}$), simplifications of the biological neurons described above. Common to most neural models is a *local updating rule*

$$v_i = g\left(\sum_{j=1}^{N} \omega_{ij} v_j - \theta_i\right), \tag{1}$$

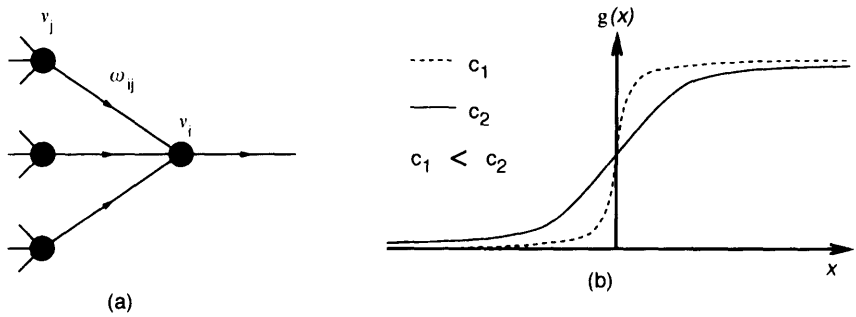

Figure 7.1 (a) Neuron updating and (b) sigmoid response functions of equations (1) and (2) for different temperatures c

where the weights (synapses) $\omega_{ij} \in \mathbb{R}$ are nonzero only for neurons v_j that feed to the neuron v_i. These weights can have both positive (excitatory) and negative (inhibitory) values. The θ_i term is a threshold, corresponding to the membrane potential in a biological neuron. The nonlinear *transfer function* g: $\mathbb{R} \to [0, 1]$ is typically a sigmoid-shaped function like

$$g(x) = \tfrac{1}{2}(1 + \tanh(x/c)), \tag{2}$$

where the 'temperature' $c > 0$ sets the inverse gain: a lower temperature gives a steeper transfer function (Figure 7.1(b)). The limit $c \to 0$ produces a step function corresponding to discrete neurons. This simple artificial neuron mimics the main features of real biological neurons in terms of linear additivity for the inputs and strong nonlinearity for the resulting output. If the integrated input signal is larger than a certain threshold θ_i, the neuron will fire. There are two different kinds of architecture in neural network modeling: feedforward (Figure 7.2(a)) and and feedback (Figure 7.2(b)); we will describe them below.

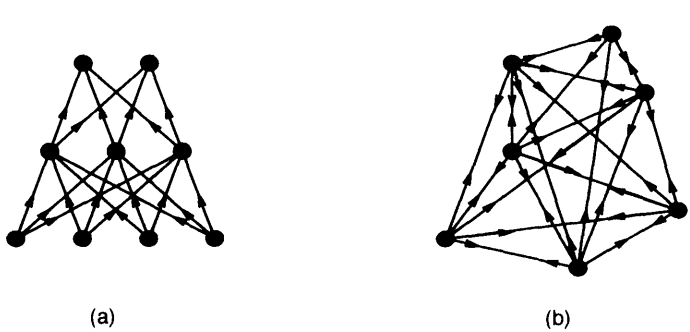

Figure 7.2 (a) Feedforward and (b) feedback architectures

2.3 Feedforward networks

Feedforward networks process signals in a one-way manner, from a set of input units in the bottom to output units in the top, layer by layer using the local updating rule (1). Feedback networks allow information to go both ways – the synapses are bidirectional. How can the power of these networks be exploited? Feedforward networks have two major areas of application, somewhat overlapping; they are feature recognition and function approximation.

Feature recognition

Feature recognition (or pattern classification) is about categorizing input patterns $\mathbf{x} \in \mathbb{R}^{N_i}$ in terms of different features $o_1, \ldots, o_{N_f}$. An input pattern $\mathbf{x} = (x_1, x_2, \ldots, x_{N_i})$ is fed into the input layer (receptors), and the output nodes represent the features. For the architecture in Figure 7.2(a), with one intermediate (or hidden) layer of N_h neurons, o_i depends on $\mathbf{x}$ (cf. (1)) as

$$o_i(\mathbf{x}) = g\left(\sum_{j=1}^{N_h} \omega_{ij}^{(2)} g\left(\sum_{k=1}^{N_i} \omega_{jk}^{(1)} x_k\right)\right), i = 1, \ldots, N_f, \tag{3}$$

where $\omega_{ij}^{(l)}$ are the weight parameters between layers l and $l+1$. Equation (3) may be generalized to any number of layers. (The thresholds, or bias terms, θ_i appearing in (1) have in (3) been transformed into weights by adding to each layer an extra dummy unit, which is constantly equal to 1.)

Fitting $\omega_{ij}^{(l)}$ to a given set of input patterns (learning) takes place with gradient descent on a suitable *error function*. In this process the *training patterns* are presented over and over again with successive adjustments of the weights – *back-propagation* of the error [Rumelhart & McClelland, 1986]. Once this iterative learning has reached an acceptable level, in terms of a low error rate, the weights are frozen and the ANN is ready to be used on patterns it has never seen before. The capability of the network to correctly characterize these *test patterns* is called *generalization* performance. This procedure is somewhat analogous to 'normal' curve fitting, where a smooth parameterization from a training is used to interpolate between the data points (generalization).

Function approximation

Rather than having a 'logical' output unit (o_i) with a threshold behavior described by (1) and (2), one could imagine having an output representing an unrestricted real number, obtained e.g. by replacing the sigmoid in (1) and (2) with a linear transfer function for the output units. In this case one adjusts the weight parameters to parameterize an unknown real-valued function. Such an approach can be useful in *time-series predictions*, where one aims at predicting future values of a series given previous values [Lapedes & Farber, 1987; Weigend, Huberman & Rumelhart, 1990], e.g.

$$x_t = \mathscr{F}(x_{t-1}, x_{t-2}, \ldots),$$

where x_t is the real-valued output node and $x_{t-1}, x_{t-2}, \ldots$ are the values of the series at previous times.

2.4 Feedback networks

In contrast to the feedforward networks, the activation in feedback networks continues until a steady state has been reached. Feedback networks appear in the context of associative memories using the Hopfield model [Hopfield, 1982] and difficult optimization problems [Hopfield & Tank, 1985; Peterson & Söderberg, 1989], which is the focus of this chapter, but also in feature recognition applications using the Boltzmann machine [Ackley, Hinton & Sejnowski, 1985] and its mean field approximation [Peterson & Hartman, 1989]. Simple models for magnetic systems have a lot in common with feedback networks, so they have been the source of much inspiration. We therefore start this section by familiarizing the reader with *Ising models* for magnetic systems.

Magnetic systems

The *Ising model* describes a magnetic system in terms of a set of binary spins $s_i \in \{-1, 1\}$, $i = 1, \ldots, N$, which are effective variables for the individual atoms, assumed to be positioned on a lattice. The two spin states at each site i represent the possible magnetization directions. The model is governed by an energy function $E(s)$ (from here on we use the convention that a symbol stripped of its indices represents the whole collection of variables denoted by the indexed symbol), given by

$$E(s) = -\frac{J}{2} \sum_{\langle i,j \rangle} s_i s_j, \tag{4}$$

where the sum runs over pairs i, j of neighboring sites. Thus the neighboring spins interact via a constant attractive coupling of strength $J > 0$. The lowest energy state is reached by iterative updating of the spins according to

$$s_i = \operatorname{sgn}\left(J \sum_{\langle j \rangle_i} s_j \right), \tag{5}$$

where the sum runs over the neighbors j of i, and sgn is the algebraic sign function. Eventually, a state is reached where all spins point in one of the two possible directions (Figure 7.3(a)). If the system is embedded in a thermal environment (Figure 7.3(a) characterizes $c = 0$), fluctuations will appear subject to the Boltzmann distribution

$$\mathbb{P}(s) \propto \exp(-E(s)/c),$$

which can be simulated by replacing the dynamics of (5) by some stochastic procedure leading to fluctuating configurations (Figure 7.3(b)).

For large N the behavior of the *average magnetization* $M = \langle \bar{s} \rangle$, $\bar{s} = \sum_{i=1}^{N} s_i/N$, depends on c. Above a phase transition point, $\hat{c}$, there is no global alignment, and

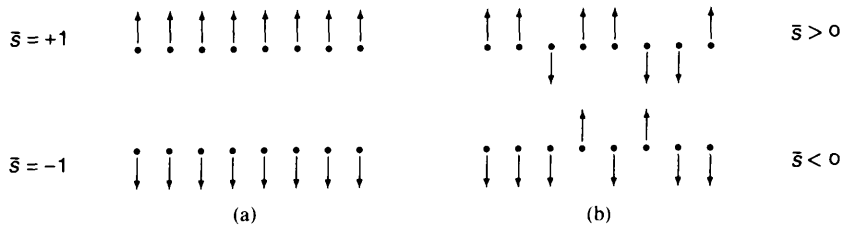

Figure 7.3 (a) The two possible $c = 0$ states for the Ising model and (b) two typical $c > 0$ configurations

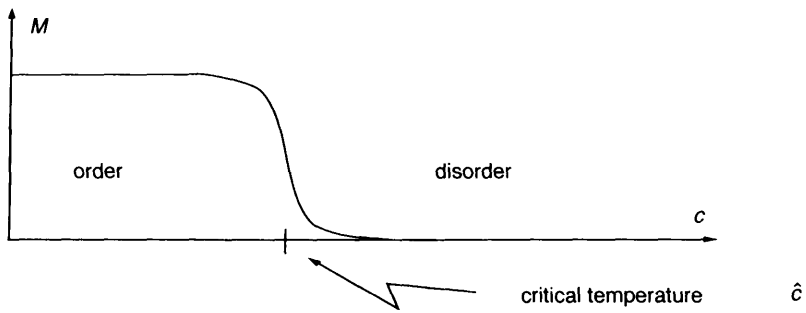

Figure 7.4 Average magnetization M as a function of c, illustrating a phase transition

$M = 0$. At very high temperatures there is no alignment whatsoever – all the spins are completely random. A phase transition between an *ordered* phase and a *disordered* phase is depicted in Figure 7.4. Indicators of phase transitions such as M (order parameters) represent *global* properties of the system. Following the evolution of individual spins does not tell us very much. The transition into an ordered phase will play an important role in the applications of feedback networks described below; this is because information is related to order.

Similar behavior can be found in a more realistic modeling of magnets. It is interesting that the effective description given by the spin models suffices for capturing the global properties of such magnetic systems; all atomic physics details are lumped into *effective* spin variables and their energy function E.

An interesting generalization of the Ising model is a *spin glass* system (which can model certain alloys like AuFe), obtained by allowing for:

- nonlocal interactions, i.e. terms $\propto s_i s_j$ in E for all $j \neq i$;
- pair-dependent (symmetric) couplings $\omega_{ij} = \omega_{ji}$ between s_i and s_j.

Thus (4) is replaced by

$$E(s) = -\frac{1}{2} \sum_{i \neq j = 1}^{N} \omega_{ij} s_i s_j. \tag{6}$$

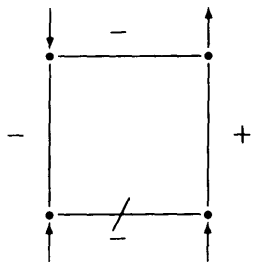

Figure 7.5 Frustration: four spins connected by differently signed couplings

Allowing also for negative couplings produces *frustration*, a situation with conflicting interests (Figure 7.5). Not all the energy terms can be minimized simultaneously, which leads to many almost degenerate 'ground states'. For a Gaussian random distribution of couplings ω_{ij} there exist $\approx e^{0.2N}$ such low energy states for an N spin system. This suggests the possibility of exploiting such a system for information technology – a feedback network.

The Hopfield model

The Hopfield model [Hopfield, 1982] is based on the energy function of (6) with binary neurons $s_i = \pm 1$. By appropriate choice of ω_{ij} the idea is to let the system function as an associative memory. A dynamics that locally minimizes (6) (cf. (5)) is given by

$$s_i = \operatorname{sgn}\left(\sum_{j \neq i} \omega_{ij} s_j\right). \tag{7}$$

Given a set of N_p patterns $\mathbf{x}^{(p)} = (x_1^{(p)}, x_2^{(p)}, \ldots, x_N^{(p)})$, $p = 1, \ldots, N_p$, with $x_i^{(p)} \in \{-1, 1\}$, the *Hebb rule* [Hebb, 1949]

$$\omega_{ij} = \frac{1}{N_p} \sum_{p=1}^{N_p} x_i^{(p)} x_j^{(p)}$$

is used for learning. With this choice of ω_{ij} one can show [Hopfield, 1982] that, under certain conditions and when initiated at some starting value $s_i^{(0)}$, the updating rule of (7) brings the system to the closest stored pattern $x_i^{(p)}$, which is a local minimum of E. One has an associative memory.

Combinatorial optimization

Many combinatorial optimization problems are known to be NP-complete. Exact solutions to such problems require state-space exploration of one kind or another, leading to a growth of the computational effort with the size of the system that is exponential or even factorial. Different kinds of heuristic method are therefore often used to find reasonably good approximate solutions. The ANN approach, which is based on feedback networks, falls within this category.

It has advantages in terms of solution quality and a 'fuzzy' interpretation of the answers through the MF variables. Furthermore, it is inherently parallel, facilitating implementations on concurrent processors, and custom-made hardware is straightforward to design for MF equations. The ANN approach differs from optimization and most other approximation methods in the sense that it has no trial-and-error mechanism; it 'feels' its way to a good solution.

There are two families of ANN algorithms for optimization problems:

- The '*pure*' *neural approach* is based on a system of either binary [Hopfield & Tank, 1985] or multistate [Peterson & Söderberg, 1989] neurons with mean field equations for the dynamics; it is a very general approach.
- *Hybrid approaches*, such as 'deformable template' algorithms [Durbin & Willshaw, 1987], supplement the neural variables with problem-specific variables, e.g. geometric variables.

Hybrid approaches are appropriate for low-dimensional geometrical problems like the traveling salesman problem (TSP), whereas the more general approach of pure neural networks is suitable for generic multiple-choice problems, like assignment or scheduling problems. The following sections describe both approaches illustrated by relevant applications.

3 PURE ANN APPROACH TO OPTIMIZATION: BINARY CASE

3.1 Basic encoding

In the simplest neural approach to an optimization problem we assume that the problem at hand can be formulated in terms of a system of Ising (binary) spins $s_i = \pm 1$, $i = 1, \ldots, N$, as the minimization of an objective (energy) function $E(s)$. This may take the form of (6),

$$E(s) = -\frac{1}{2} \sum_{i \neq j = 1}^{N} \omega_{ij} s_i s_j, \tag{8}$$

with a suitable choice of couplings ω_{ij} so as to represent the particular problem. Note that in this approach the couplings (or, more generally, the parameters of $E(s)$) are fixed once and for all for each problem instance; they are not adaptive as in some other domains of ANN application.

No approximations have yet been made; everything so far amounts to a particular mathematical encoding of the problem.

3.2 Minimizing E

The next step will be to devise an efficient procedure for minimizing $E(s)$, such that spurious local minima are avoided as much as possible. In the neural approach, this is done by employing the *mean field* (MF) approximation, to be defined below. It will turn out to be very powerful in this respect. Before arriving at the MF equations, we will touch upon some related approaches.

A straightforward method for minimizing E is to update s_i according to a local optimization rule,

$$s_i = \operatorname{sgn}\left(\sum_j \omega_{ij} s_j\right). \tag{9}$$

With this procedure the system typically ends up in a local minimum close to the starting point, which is not desired here.

Instead, a stochastic algorithm might be employed, which allows for uphill moves. One such possibility is *simulated annealing* (SA) [Kirkpatrick, Gelatt & Vecchi, 1983]; see also Chapter 4. This method consists of generating configurations via neighborhood search methods according to the *Boltzmann distribution*

$$\mathbb{P}(s) = \frac{1}{Z} e^{-E(s)/c},$$

where Z is the *partition function*

$$Z = \sum_s e^{-E(s)/c}. \tag{10}$$

Here, $c > 0$ is the temperature of the system, and the sum runs over the allowed values ± 1 of all the spins s_i. When c is small, the distribution should be concentrated around the global energy minimum. By starting at a finite c and generating configurations at successively lower c (annealing), the final configuration is less likely to get stuck in a bad local minimum. But this procedure can be very time-consuming.

3.3 The mean field equations

The MF approach aims at approximating the stochastic SA method with a set of deterministic equations. The derivation has two steps. First, the partition function of (10) is rewritten in terms of an integral over new continuous variables u_i and v_i. Second, Z is approximated by the maximum value of its integrand.

To this end, embed the spins s_i in a linear space $\mathbb{R}$, introduce a new set of variables v_i living in this space, one for each spin, and set them equal to the spins with Dirac delta functions. Then we can express the energy in terms of the v_i, and Z takes the form

$$Z = \sum_s \int dv\, e^{-E(v)/c} \prod_i \delta(s_i - v_i).$$

Next Fourier expand the delta functions, introducing a set of conjugate variables u_i to produce

$$Z \propto \sum_s \int dv \int du\, e^{-E(v)/c} \prod_i e^{u_i(s_i - v_i)}.$$

Finally, carry out the original sum over s:

$$Z \propto \int dv \int du\, e^{-E(v)/c - \sum_i u_i v_i + /\sum_i \log\cosh u_i} \equiv \int dv \int du\, e^{-E_{\text{eff}}(u,v)/c}. \tag{11}$$

The original partition function is now rewritten entirely in terms of the new variables u_i, v_i, with an effective energy $E_{\text{eff}}(u, v)$ in the exponent. So far no approximation has been made. We next assume that Z in (11) can be approximated by the contribution from the maximal value of the integrand, the saddle-point approximation. For the position of the saddle point we obtain

$$u_i = -\frac{\partial E(v)}{\partial v_i} \Big/ c, \tag{12}$$

$$v_i = \tanh u_i. \tag{13}$$

Combining (12) and (13) we obtain the *mean field equations*

$$v_i = \tanh\left(-\frac{\partial E(v)}{\partial v_i} \Big/ c \right). \tag{14}$$

For the energy of (8) this gives the familiar

$$v_i = \tanh\left(\sum_j \omega_{ij} v_j / c \right).$$

The *mean field variables* v_i can be interpreted as (approximate) thermal averages $\langle s_i \rangle_c$ of the original binary spins. We thus recover the local updating equations (1) and (2). What we have obtained is a set of deterministic equations emulating the stochastic behavior. They correspond to an approximation where each spin feels the others only via their averages.

An alternative derivation of the MF equations can be done in a variational approach, where the free energy is minimized with respect to the coefficients of a linear Ansatz for an approximate energy.

3.4 The mean field neural approach

The MF equations define a feedback neural network, if v_i is interpreted as a neuron with u_i as its input, acted upon by the transfer function $\tanh(\cdot)$. The neural approach for binary systems consists in solving the MF equations (14) *iteratively*, either synchronously (in parallel) or serially (one v_i at a time), starting at a finite c and slowly letting $c \to 0$ (annealing).

For this approach to work well in the low c limit, it is desirable that $E(s)$ contain no self-couplings of the form $s_i s_i$, or more generally, that $\partial E/\partial s_i$ not depend on s_i. This can always be arranged (e.g., $s_i s_i$ can always be replaced by 1). Then in the $c \to 0$ limit, $\tanh(\cdot/c)$, will turn into a step function, and (14) will reduce to local optimization of the original spin system (cf. (9)).

The basic advantage of the MF approach is that, at finite c, the mean field variables can evolve through a continuous space not accessible to the original, discrete spin variables.

3.5 Analysis of the dynamics

The dynamics of this kind of feedback ANN typically exhibits a behavior with two phases (similar to Figure 7.4): at large temperatures the system relaxes into a trivial fixpoint, which is zero for the quadratic objective function (8). As the temperature is lowered, a 'phase transition' occurs at a critical temperature $c = \hat{c}$, where the trivial fixpoint turns unstable, and as $c \to 0$ nontrivial fixpoints $v_i^{(*)} = \pm 1$ emerge, representing a suggestive solution to the optimization problem in question (Figure 7.6).

The position of $\hat{c}$ depends on ω_{ij}, and can be calculated by expanding the sigmoid function $\tanh(\cdot/c)$ in a power series around the origin (Figure 7.6). In this approximation the dynamics is linear, and the v_i evolve according to

$$v_i = \frac{1}{c} \sum_j \omega_{ij} v_j. \tag{15}$$

For *synchronous updating* it is clear that, if the modulus of one of the eigenvalues of the matrix ω/c in (15) is greater than 1, the trivial fixpoint becomes unstable and the system can begin exploring the nonlinear region. This happens when c decreases past the larger of (a) the largest positive eigenvalue of ω, or (b) minus the largest negative eigenvalue of ω. Case (a) is desirable, since case (b) leads to oscillating behavior due to a negative destabilizing eigenvalue. To enforce case (a), a positive self-coupling is sometimes needed to enable synchronous updating; this might overstabilize the dynamics at low c, leading to lower-quality solutions [Peterson & Söderberg, 1989].

In the case of *serial updating* the philosophy is the same but the analysis slightly more complicated; we refer the reader to Peterson & Söderberg [1989] for a more detailed discussion. The result is simple though (and encouraging): $\hat{c}$ is always

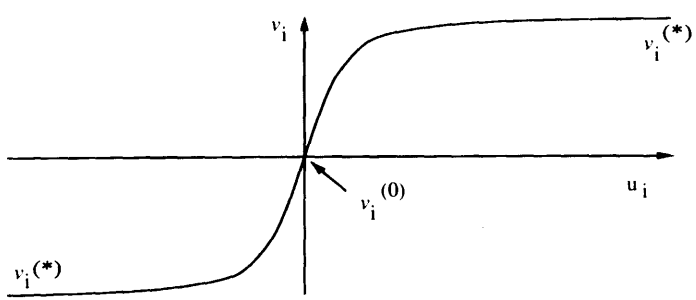

Figure 7.6 Fixpoints and $\tanh u_i$

given by the largest positive eigenvalue of ω, which corresponds to case (a). No self-coupling is needed to stabilize the dynamics.

Finding the largest eigenvalue presents no computational difficulty. For problems where the coupling matrix contains random parts, knowledge of the average coupling and the corresponding standard deviation often suffices to produce a good estimate of $\hat{c}$. Such an analysis is also important in foreseeing and avoiding oscillatory behavior with synchronous updating.

Given a method of estimating $\hat{c}$ in advance, one can devise a reliable, parallelizable 'black-box' algorithm for solving problems of this kind (cf. Section 4.4).

3.6 The graph bisection problem

An application of the approach described above is the graph bisection (GB) problem. The neural approach is very transparent in this case, since the problem has an intrinsic binary structure. The GB problem is defined as follows (Figure 7.7(a)). Partition the N nodes of a given graph into two subsets of equal size ($N/2$) such that the number of connections between the two halves, the cutsize, is minimized. The graph is defined by the connection matrix, J_{ij}, $i,j = 1,\ldots,N$, where J_{ij} is 1 if nodes i and j are connected, and 0 otherwise.

The problem is mapped onto an Ising spin system by the following representation. For each node i, assign a binary neuron $s_i = \pm 1$, signifying whether the node belongs to one half or the other. Then $s_i s_j = 1$ whenever i and j are in the same partition, and -1 otherwise. Neglecting an unimportant additive constant, the cutsize is then proportional to

$$E_{\text{conn}} = -\frac{1}{2}\sum_{i,j=1}^{N} J_{ij} s_i s_j.$$

However, minimization of E_{conn} alone will lead to the result that all nodes are forced into one partition. To enforce the *global constraint* of even partition, we need a *constraint term* that penalizes situations where the nodes are not evenly partitioned. Since $\sum_i s_i = 0$ precisely when the partition is balanced, a term

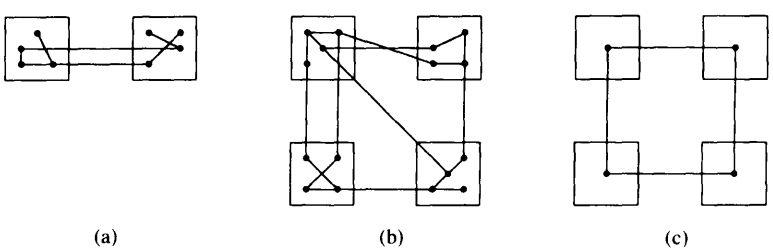

Figure 7.7 (a) A graph bisection problem; (b) a graph partitioning problem with $K = 4$; and (c) a TSP with $N = 4$

proportional to $(\sum_i s_i)^2$ will obviously increase the energy for an unbalanced partition. After a final removal of diagonal terms, a possible energy function for graph bisection takes the form

$$E(s) = -\frac{1}{2}\sum_{i,j=1}^{N} J_{ij} s_i s_j + \frac{\alpha}{2}\left(\left(\sum_{i=1}^{N} s_i\right)^2 - \sum_{i=1}^{N} s_i^2\right), \tag{16}$$

where the *constraint coefficient* $\alpha > 0$ sets the relative strength between the cutsize and constraint terms. Note that the energy in (16) has precisely the form of (6) with $\omega_{ij} = J_{ij} + \alpha(\delta_{ij} - 1)$. The corresponding MF equations are

$$v_i = \tanh\left(\frac{1}{c}\sum_{j=1(j\neq i)}^{N} (J_{ij} - \alpha) v_j\right),$$

which are to be iterated, serially or in parallel, under annealing in c, until convergence.

Note the generic form of (16),

$$E = \text{'cost'} + \text{'global constraint'},$$

which is typical when casting combinatorial optimization problems onto neural networks. The difficulty inherent in this kind of problem is very transparent here: the system is frustrated in the sense that the two terms ('cost' and 'global constraint') are competing with each other. Just as for spin glasses, such frustrations often lead to many local minima.

Our treatment of constraints resembles penalty function methods such as Lagrangian relaxation, but differs from straightforward heuristic approaches. In the case of graph bisection one often starts in a configuration where the nodes are equally partitioned and then proceeds by swapping pairs subject to some acceptance criteria; the constraint of equal partition is respected throughout the updating process. This is in contrast to the neural network technique, where the constraints are implemented in a 'soft' manner by a penalty term. The final MF solutions could therefore sometimes suffer from a minor imbalance. This is easily remedied, either by applying a *greedy heuristic* to the solutions, or by reheating the system and letting it reanneal.

Peterson & Anderson [1988] used serial updating to achieve good numerical results for the graph bisection problem on random graphs of a fixed connectivity and graph sizes N ranging from 20 to 2000. Their quality is comparable to solutions obtained by the time-demanding simulated annealing method. This technique has a time consumption lower than other methods, and it becomes more competitive if the intrinsic parallelism is exploited on dedicated hardware.

4 OPTIMIZATION WITH POTTS NEURAL NETWORKS

For a large group of combinatorial optimization problems, it is natural to choose an encoding in terms of binary elementary variables. However, there are many problems where this is not the case. One kind of complication arises when the

natural elementary decisions are of the type one-of-K with $K > 2$, rather than one-of-two.

Early attempts to approach such problems by neural network methods were in terms of *neuron multiplexing*, where for each elementary K-fold decision, K binary 0/1 neurons were used, with the additional constraint that precisely one of them be on (equal to 1). These *syntax* constraints were implemented in a soft way as penalty terms. As it turned out in the original work on the traveling salesman problem [Hopfield & Tank, 1985], as well as in subsequent investigations for the graph partitioning problem [Peterson & Söderberg, 1989], this approach does not produce high-quality solutions in a parameter-robust way.

As was demonstrated by Peterson & Söderberg [1989], an alternative encoding using *Potts neurons* (see Wu [1983]) is the way to go because the syntax constraint is built-in.

4.1 Potts spins

A K-state Potts spin is a variable that has K possible values (states). Such a variable can be represented in many ways; for our purposes, the best way is as a vector in the Euclidean space $\mathscr{E}_K$. Thus, denoting a spin variable by $\mathbf{s} = (s_1, s_2, \ldots, s_K)$, the ath possible state is given by the ath principal unit vector, defined by $s_a = 1$, $s_b = 0$ for $b \neq a$. These vectors point to the corners of a regular K-simplex (Figure 7.8 shows the case of $K = 3$). They are all normalized and mutually orthogonal, and also fulfill $\sum_a s_a = 1$.

We assume that an optimization problem has been given, and encoded in terms of a set of N distinct K-state Potts spins $\mathbf{s}_i = (s_{i1}, \ldots, s_{iK})$, $i = 1, \ldots, N$, together with an objective function $E(\mathbf{s})$ to be minimized. Often $E(\mathbf{s})$ can be written in the form

$$E(\mathbf{s}) = -\frac{1}{2} \sum_{i,j=1}^{N} \omega_{ij} \mathbf{s}_i \cdot \mathbf{s}_j. \tag{17}$$

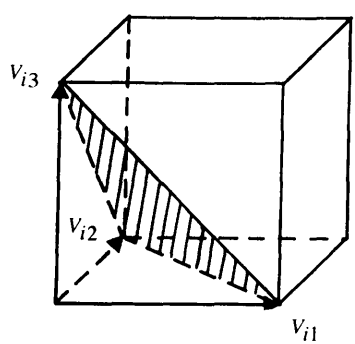

Figure 7.8 The volume of solutions corresponding to the neuron-multiplexing encoding for $K = 3$; the shaded plane corresponds to the solution space of the corresponding Potts encoding

As in the case of Ising spins, it is desirable (and always possible) to choose $E(\mathbf{s})$ not to contain any self-couplings (i.e., $\omega_{ii} = 0$, $i = 1, \ldots, N$).

4.2 Potts mean field equations

We next derive the MF equations corresponding to K-state Potts spins. Again we start off with the partition function

$$Z = \sum_{\mathbf{s}} \mathrm{e}^{-E(\mathbf{s})/c},$$

where the sum runs over the allowed states of the whole set of Potts spins. Proceeding along the same lines as the binary case, the partition function is transformed into

$$Z \propto \int d\mathbf{u} d\mathbf{v} \exp\left(-E(\mathbf{v})/c - \sum_i \mathbf{u}_i \cdot \mathbf{v}_i \right) \sum_{\mathbf{s}} \exp\left(\sum_i \mathbf{u}_i \cdot \mathbf{s}_i \right).$$

Performing the $\mathbf{s}_i$ sums, we are left with

$$Z \propto \int d\mathbf{u} d\mathbf{v} \exp\left(-E(\mathbf{v})/c - \sum_i \mathbf{u}_i \cdot \mathbf{v}_i + \sum_i \log \sum_a \mathrm{e}^{u_{ia}} \right). \tag{18}$$

The MF approximation to $\langle \mathbf{s}_i \rangle$ is again given by the value of $\mathbf{v}_i$ at a saddle point of the effective energy of $(\mathbf{u}, \mathbf{v})$ in the exponent of (18). Differentiating, we obtain the Potts MF equations (cf. (12), (13)):

$$u_{ia} = -\frac{\partial E(\mathbf{v})}{\partial v_{ia}} \Big/ c, \tag{19}$$

$$v_{ia} = \frac{\mathrm{e}^{u_{ia}}}{\sum_b \mathrm{e}^{u_{ib}}}, \tag{20}$$

where indices a and b denote vector components.

The set of *Potts neurons* $\mathbf{v}_i$ defines a Potts neural network, with the dynamics given by iteration of (19) and (20), Equation (20) defines a kind of vector sigmoid function, acting on the input $\mathbf{u}_i$, and it follows trivially that

$$v_{ia} > 0, \sum_a v_{ia} = 1.$$

One can thus think of the neuron component v_{ia} as the probability for the ith Potts spin to be in state a (fuzzy logic). The state space available to $\mathbf{v}_i$ is the interior of the K-simplex spanned by the allowed states for the Potts spin $\mathbf{s}_i$. For $K = 3$ this is a triangular area (Figure 7.8).

In particular, for $K = 2$ we recover the formalism of the Ising case, provided that we make the following identification for the Ising mean field variable v_i:

$$v_i = v_{i1} - v_{i2},$$

equivalent to

$$v_{i1} = (1 + v_i)/2,$$
$$v_{i2} = (1 - v_i/2).$$

The Potts MF formalism thus provides a natural generalization of the MF approach from binary to general multistate systems.

4.3 Mean field dynamics

High temperature

Again one can analyze the linearized dynamics as in Section 3.5 in order to estimate the critical temperature $\hat{c}$. The value of the critical temperature depends on the couplings and on precisely how the updating is done. In *serial mode*, the Potts neurons are updated one by one, using fresh values of previously updated neurons. In *synchronous mode*, all neurons are updated in parallel, using only old values as input.

At large enough temperatures the system relaxes into a trivial fixpoint $v_{ia}^{(0)}$, which is a completely symmetric state, where all v_{ia} are equal:

$$v_{ia}^{(0)} = \frac{1}{K}. \tag{21}$$

As the temperature is lowered, a phase transition is passed at $c = \hat{c}$, and as $c \to 0$ nontrivial fixpoints $v_{ia}^{(*)}$ emerge which are characterized by $\Sigma \to 1$, where

$$\Sigma \equiv \frac{1}{N} \sum_{ia} v_{ia}^2$$

is the *saturation*, which measures the degree to which the neurons are forced into the corners of the accessible state space. Figure 7.9 illustrates this for a graph partitioning problem with $K = 4$, $N = 100$ (see Section 4.5). The trivial fixpoint corresponds to the symmetry point of the nonlinear transfer functions in (20), where the dynamics is almost linear (Figure 7.6). Let us consider fluctuations around the trivial fixpoint

$$v_{ia} = v_{ia}^{(0)} + \varepsilon_{ia}.$$

First, from (20) and (21) it follows that

$$\sum_a \varepsilon_{ia} = 0,$$

so the fluctuations will always be perpendicular to $(1, 1, 1, \ldots)$. In a linear approximation (suppressing the index a) the perpendicular components of ε_{ia} evolve according to

$$\varepsilon_i = \frac{1}{Kc} \sum_j \omega_{ij} \varepsilon_j$$

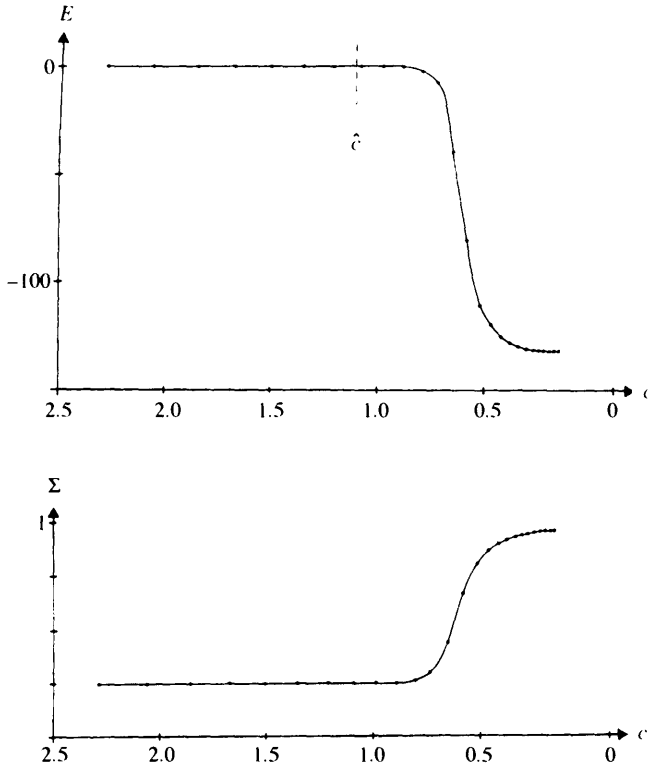

Figure 7.9 (a) Internal energy $E(v_{ia})$ as a function of c for a graph partitioning problem with $K = 4$ and $N = 100$; also shown is an approximation to $\hat{c}$ based on a statistical estimate; (b) the saturation $\sum$ as a function of c for the same problem.

(cf. (15)). This is almost identical to the situation for binary neurons. Thus, provided Kc is substituted for c, the discussion of $\hat{c}$ following (15) in Section 3 applies in every detail. The result of the analysis follows.

For *synchronous updating*, $\hat{c}$ is given by the larger of (a) $1/K$ (largest positive eigenvalue of ω) and (b) $-1/K$ (largest negative eigenvalue of ω). In (a) a positive eigenvalue is responsible for the destabilization of the trivial fixpoint; this is to be preferred. In (b), which pertains for some problems, a negative eigenvalue is involved, leading to oscillatory behavior. Case (b) can be avoided by adding an artificial self-coupling term to the objective function, but this can have the side effect of overstabilizing the dynamics at low c, leading to a lower expected solution quality.

For *serial updating*, we always have case (a), so this mode of updating is safe to use for all kinds of problems, and no self-coupling is needed.

Low temperature

At low temperature, any difference between the components u_{ia} of a neuron's input will be strongly magnified. As a result, we have a *winner-take-all* situation, where one neuron component will be almost one, and the others close to zero. The dynamics turns discrete, and a kind of local optimization results as $c \to 0$. Eventually the network should settle at a fixpoint close to an allowed state of the Potts spin system, representing a suggested solution to the optimization problem.

A self-coupling is sometimes needed with synchronous updating, to stabilize the high-c dynamics. Adding a self-coupling term, $(-\beta/2)\sum_{ia} v_{ia}^2$, has a clear impact at low c; a positive β value tends also to stabilize bad decisions (positive feedback), whereas a negative value destabilizes even a good solution, and can lead to cyclic or even chaotic behavior.

For serial mode, there is never a need for self-coupling, so serial mode will be understood where not otherwise stated. For some problems, implicit self-couplings are difficult to avoid; a general method to deal with such complications will be discussed in Section 5.

4.4 A black-box procedure

For most problems, the proper value of $\hat{c}$ can be calculated or estimated in advance. The complete neural network algorithm for a generic K-state Potts system (the Ising system is just a special case) will then look as follows.

A generic Potts neural network algorithm

1. Choose a problem instance $\Rightarrow \{\omega_{ij}\}$.
2. Calculate (or estimate) the phase transition temperature $\hat{c}$ by linearizing (20). (For synchronous updating: add a self-coupling $\beta\delta_{ij}$ to ω_{ij}, if necessary, and modify $\hat{c}$ accordingly.)
3. Initialize the neurons v_{ia} with $1/K \pm$ random values, and set $c = \hat{c}$.
4. Until $\Sigma \geqslant 0.99$, do:

   update all v_{ia}: $v_{ia} = e^{u_{ia}}/\sum_b e^{u_{ib}}$, with u_{ia} given by $(-\partial E(\mathbf{v})/\partial v_{ia})/c$;
   anneal: $c = 0.9 \times c$.

5. Finally, if needed to correct for violations of softly implemented constraints, perform a greedy heuristic on the obtained solution.

4.5 The graph partitioning problem

A generalization of the graph bisection problem is *graph partitioning* (GP): an N-node graph, defined by a symmetric connection matrix $J_{ij} = 0,1$, $i \neq j = 1, \ldots, N$, is to be partitioned into K subsets of N/K nodes each, while minimizing the cutsize, i.e., the number of connected node pairs winding up in different subsets (see Figure 7b).

Potts representation

This problem is naturally encoded in terms of K-state Potts spins as follows: For each node $i = 1, \ldots, N$, a Potts spin variable $\mathbf{s}_i = (s_{i1}, \ldots, s_{iK})$ is assigned, such that the spin component s_{ia} takes the value 1 or 0 depending on whether node i belongs to subset a or not. A suitable energy function (cf. (16)) is given by

$$E = -\frac{1}{2}\sum_{i,j=1}^{N} J_{ij}\mathbf{s}_i \cdot \mathbf{s}_j + \frac{\alpha}{2}\left(\left(\sum_{i=1}^{N}\mathbf{s}_i\right)^2 - \sum_{i=1}^{N}\mathbf{s}_i^2\right),$$

where the first term is a cost term (cutsize) and the second is a penalty term with a minimum when the nodes are equally partitioned into the K subsets.

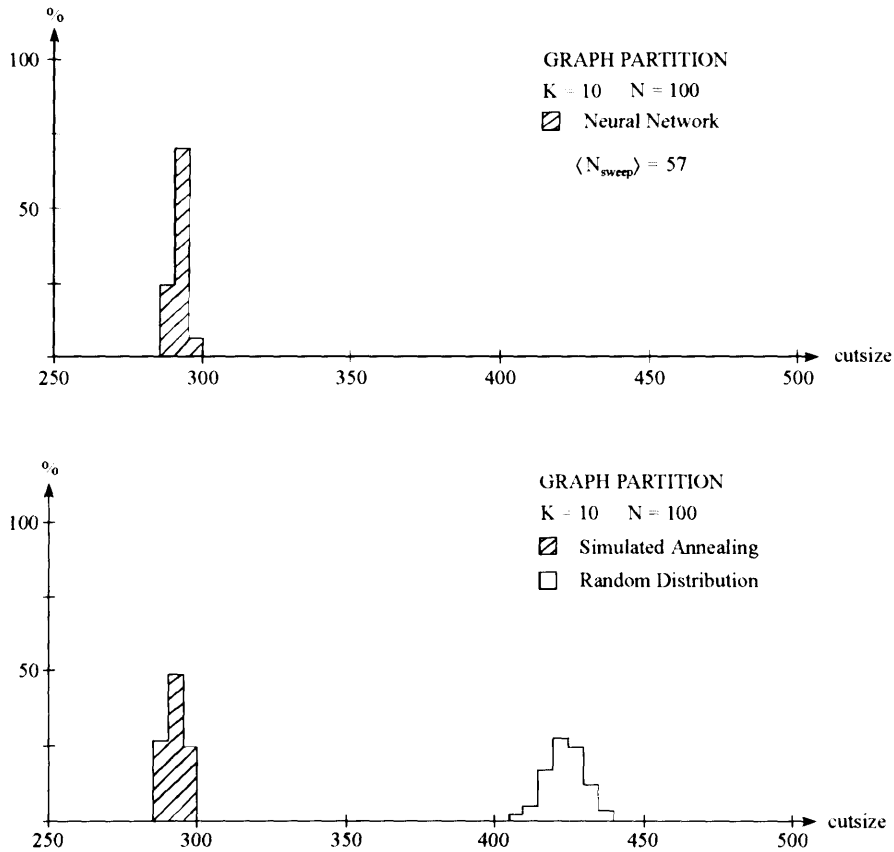

Figure 7.10 Comparison of Potts neural network solutions versus simulated annealing and random partitions for a graph partitioning problem with $K = 10$ and $N = 100$. The histograms are based on 50 problem instances for the neural network and simulated annealing algorithms and 1000 for the random partition.

Results from numerical experiments

Figure 7.10 compares the performance of the Potts neural network with simulated annealing. The graphs were generated by randomly connecting the node pairs with probability $P = 10/N$, and the critical temperature $\hat{c}$ could be estimated by statistical methods. The results are impressive. The neural network algorithm performs as well as, sometimes better than the simulated annealing method with its excessive annealing and sweep schedule [Peterson & Söderberg, 1989]. This is accomplished with between 50 and 100 iterations. In fact, the number of iterations needed was found empirically to be independent of problem size [Peterson & Söderberg, 1989].

4.6 The traveling salesman problem

In the traveling salesman problem (TSP) the coordinates $\mathbf{x}_i \in \mathbb{R}^2$ of a set of N cities are given. A closed tour of minimal total length is to be chosen such that each city is visited exactly once. To encode it we define an N-state Potts neuron $\mathbf{s}_i$ for each city $i = 1, \ldots, N$, such that the component s_{ia} $(a = 1, \ldots, N)$ is 1 if city i has the tour number a and 0 otherwise. Let d_{ij} be the distance between city i and j. Then a suitable energy function is given by

$$E = \sum_{i,j=1}^{N} d_{ij} \sum_{a=1}^{N} s_{ia} s_{j(a+1)} + \frac{\alpha}{2}\left(\left(\sum_{i=1}^{N} \mathbf{s}_i\right)^2 - \sum_{i=1}^{N} \mathbf{s}_i^2\right),$$

where the first term is a cost term, and the second a soft constraint term penalizing configurations where two cities are assigned the same tour number. In the first term, $a+1$ is to be taken mod N. Note that E does not have the form of (17). (An alternative encoding results from interchanging the roles of the labels i and the components a of the Potts spins.)

Problem instances are generated by positioning the cities at random in a square. As in the graph partitioning case, the energy is minimized by iteratively solving the Potts MF equations, (19) and (20), using the generic prescription of Section 4.4. Again the linearized dynamics can be analyzed to estimate $\hat{c}$. Peterson & Söderberg [1989] explore the Potts approach for the TSP numerically for problem sizes up to $N = 200$, again with encouraging results. The average neural network solution has a quality comparable to simulated annealing, and as in the graph partitioning case, there are no really bad solutions.

As for convergence time, we have observed no dependence on N for the problem sizes probed. The comparisons with simulated annealing concern quality only. When discussing the total time consumption one has to distinguish between serial and parallel implementations. With serial execution, the time consumption of the Potts neural algorithm is proportional to N^3 for the TSP. This should be compared with $N^2\sigma(N)$ for simulated annealing, where $\sigma(N) > O(N)$ is the total number of sweeps needed. A further speed increase may be obtained by exploiting the parallelism inherent in the neural approach, which would give constant time on a general-purpose parallel machine or custom-made

hardware. The TSP may also be tackled by *deformable templates*, a hybrid approach discussed in Section 6. See also Chapter 8, Section 7.

4.7 Scheduling problems

A Potts neural network formulation is almost ideal for scheduling problems because they lend themselves to a natural formulation. In its purest form, a scheduling problem consists entirely of fulfilling a set of basic constraints, each of which can be encoded as a *penalty term* that will vanish when the constraint is obeyed. Thus, the minimum energy is known (0), and one can recognize exact (legal) solutions a posteriori by inspection. But in many applications there exist additional preferences within the set of legal schedules, preferences which lead to the appearance of *cost terms*.

First we will discuss a synthetic problem, where the principles of the neural mapping are very transparent. Then we will briefly discuss how to deal with the additional complication in a real-world problem [Gislén, Söderberg & Peterson, 1992b], in this case a Swedish high school.

A synthetic example

Gislén, Söderberg & Peterson [1989] map a simplified scheduling problem, where N_p teachers lecture N_q classes in N_x classrooms at N_t time slots, onto a Potts neural network. In this problem one wants solutions where every teacher p gives a lecture to each of the classes q, using the available rooms x and the available time-slots t, with no conflicts in space (classrooms) or time. This defines the *basic constraints* that have to be satisfied; various *preferences* regarding continuity in classrooms, etc., were also considered.

There is a very transparent way to describe this problem that naturally lends itself to the Potts neural encoding, where *events*, defined by teacher–class pairs (p, q), are mapped onto space–time slots (x, t) (Figure 7.11). The basic constraints in this picture are as follows:

1. An event (p, q) should occupy precisely one space–time slot (x, t).
2. Different events (p_1, q_1) and (p_2, q_2) cannot occupy the same space–time slot (x, t).
3. A teacher p cannot take part in more than one event at a time.
4. A class q cannot take part in more than one event at a time.

A schedule fulfilling all the basic constraints is said to be *legal*.

The obvious encoding is in terms of Potts spins $\mathbf{s}_{pq}$; the component $s_{pq;xt}$ is defined to be 1 if the event (p, q) takes place in the space–time slot (x, t) and 0 otherwise. Thus we need N_pN_q distinct K-state Potts spins, with $K = N_xN_t$. Then the first constraint is trivially satisfied through the usual Potts condition

$$\sum_{x,t} s_{pq;xt} = 1 \tag{22}$$

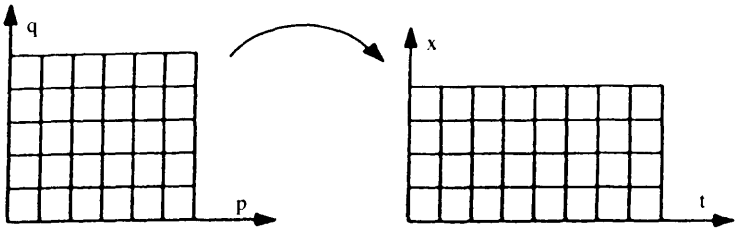

Figure 7.11 Mapping of events (p, q) onto space–time slots (x, t)

for each event (p, q). The other three constraints are implemented using energy penalty terms as follows:

$$E_{XT} = \frac{1}{2} \sum_{x,t} \left(\sum_{p,q} s_{pq;xt} \right)^2,$$

$$E_{PT} = \frac{1}{2} \sum_{p,t} \left(\sum_{q,x} s_{pq;xt} \right)^2,$$

$$E_{QT} = \frac{1}{2} \sum_{q,t} \left(\sum_{p,x} s_{pq;xt} \right)^2.$$

The energy E to be minimized is the sum of these terms, with all diagonal terms subtracted.

Again, mean field variables $v_{pq;xt} \sim \langle s_{pq;xt} \rangle_c$ are introduced, and the corresponding MF equations (cf. (19) and (20)) read

$$u_{pq;xt} = -\frac{1}{c} \cdot \frac{\partial E}{\partial v_{pq;xt}},$$

$$v_{pq;xt} = \frac{e^{v_{pq;xt}}}{\sum_{x't'} e^{u_{pq;x't'}}}.$$

With the usual annealing, a very efficient algorithm results.

However, an important simplification can be made. With the above encoding, it turns out that the MF equations produce two separate phase transitions, one in x and one in t. In other words, the system spontaneously factorizes into two parts. It is therefore economical to implement this factorization at the encoding level. This can be done by replacing each spin $\mathbf{s}_{pq}$ by the direct product of two spins: an N_x-state Potts spin $\mathbf{s}_{pq}^{(X)}$ for assigning classrooms, and an N_t-state Potts spin $\mathbf{s}_{pq}^{(T)}$ for assigning time-slots,

$$s_{pq;xt} \to s_{pq;x}^{(X)} s_{pq;t}^{(T)},$$

with separate Potts conditions replacing (22):

$$\sum_x s_{pq;x}^{(X)} = 1$$

and

$$\sum_t s^{(T)}_{pq;t} = 1$$

respectively. This reduces the dimensionality from $N_p N_q N_x N_t$ to $N_p N_q (N_x + N_t)$, so the sequential execution time goes down; the solution quality is not affected. Note that the factorization brings a situation where the Potts neurons will have different numbers of components. Also this situation can be dealt with when analyzing the MF dynamics in terms of fixpoints and $\hat{c}$.

The performance of these algorithms was investigated by Gislén, Söderberg & Peterson [1989] for a variety of problem sizes and for various levels of difficulty, as measured by the ratio between the number of events and the number of available space–time slots. Legal solutions were consistently found with very modest convergence times for problem sizes $(N_p, N_q) = (5, 5), \ldots, (12, 12)$. By convergence time we mean the total number of sweeps needed to obtain a legal solution, no matter how many trials it takes. And very good solutions were obtained when preferences were introduced, e.g., for having subsequent lessons in the same room.

High-school scheduling

The synthetic scheduling problem contains several simplifications as compared to realistic problems. Gislén, Söderberg & Peterson [1992b] explored real-world problems from the Swedish high-school system, which required an extended formalism. Here is a list of complications that must be handled by such an extended formalism:

1. A week–day partition of the time range, with a one-week periodicity, occasionally extended to two- or four-week periodicity.
2. In the toy example each teacher had each class exactly once. In the real world the teacher has to give lessons in certain subjects to a subset of the classes a few hours a week.
3. In the earlier paper it was assumed that all classrooms were available for every event. This is not true in reality; many subjects require purpose-built spaces.
4. Many subjects are taught for two hours in a row (double hours).
5. For some optional subjects the classes are broken up into option groups, temporarily forming new classes.
6. Lunch events have to appear within three hours around noon.
7. Various preferences have to be considered.

Item 1 presents no problem: the time index t is simply subdivided into weekdays (d) and daily hours (h). For item 2 we cannot use p and q as independent spin labels. Instead we define an independent label i (event index), to which p and q are attributes, $p(i)$ and $q(i)$. Other event attributes are the subject, and whether the hours are double or single; they are stored in a table containing all the a priori information relevant for each event i.

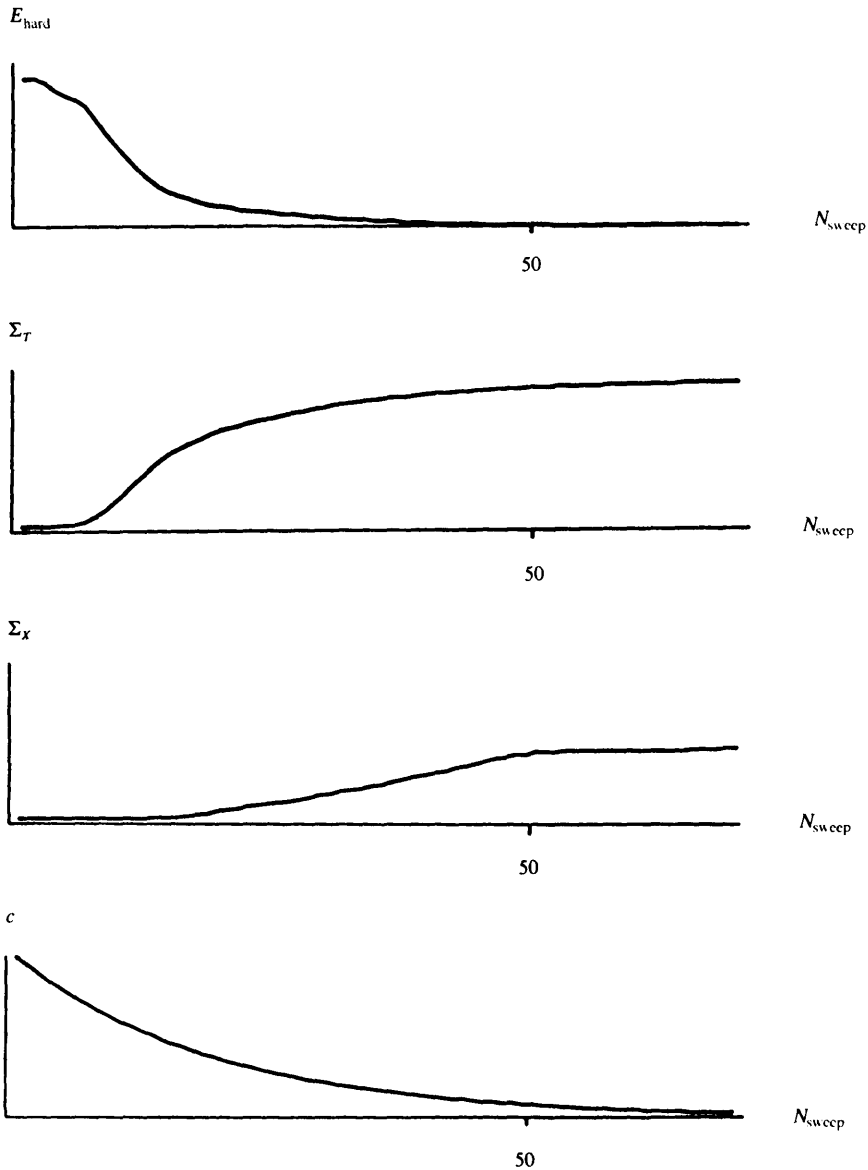

Figure 7.12 Energy (E_{hard}), saturations ($\sum_T$, $\sum_X$) and temperature (c) as functions of N_{sweep} for one run with a Swedish high-school problem; the estimated phase transition temperature $\hat{c}$ is indicated

To facilitate the handling of double-hour events, their t-neurons must be substituted in the energy with effective ones, $\tilde{s}_{i;t}^{(T)}$, defined as

$$\tilde{s}_{i;t}^{(T)} \equiv s_{i;t}^{(T)} + s_{i;t-1}^{(T)}.$$

Items 5 and 6 are syntactic constraints, which can be taken into account by restricting the Potts index ranges for the relevant events.

We encounter three kinds of *preferences* when scheduling Swedish high schools:

- The different lessons for a class in a particular subject should be *spread* as much as possible over the weekdays.
- The class schedules should have as few 'holes' as possible; lessons should be *glued* together.
- Teachers may have various individual preferences.

These preferences have to be accommodated by appropriate penalty terms; see Gislén, Söderberg & Peterson [1992b].

High-school scheduling problems typically have ~ 90 teachers, ~ 50 weakly hours, ~ 45 classes, and ~ 60 classrooms, which corresponds to $\sim 10^{4600}$ possible choices. In the factorized Potts formulation these are handled by $\sim 10^5$ neural variables.

In spite of the above complications, an automated implementation of the MF algorithm can be made along the lines of the generic prescription defined in Section 4.4, and high-quality solutions emerge [Gislén, Söderberg & Peterson, 1992b]. Figure 7.12 shows a typical evolution of the basic constraint part of the energy, the separate neuron saturations, and the temperature c.

A revision capability is inherent in the neural formalism, which is useful when encountering unplanned events once a schedule exists. Such rescheduling is performed by *clamping* those events not subject to change and heating up and cooling the remaining dynamical neurons.

One should keep in mind that problems of this kind and this size are so complex that even several staff months of human planning will in general not yield solutions that will meet all the requirements in an optimal way. We have not been able to find any algorithm in the literature that solves a real-world problem of this complexity. Existing commercial software packages do not solve the entire problem. The problem is solved with an interactive user taking stepwise decisions.

5 OPTIMIZATION WITH INEQUALITY CONSTRAINTS

The application areas dealt with earlier (traveling salesman, graph partitioning, and scheduling) are characterized by having low-order polynomial *equality* constraints. Hence they can be implemented by polynomial penalty terms. However, in many other optimization problems, especially resource allocation, one has to deal with *inequalities*. The objective of this section is to develop a method to deal with this kind of problem in the MF approach. We illustrate resource allocation by the knapsack problem, a typical example.

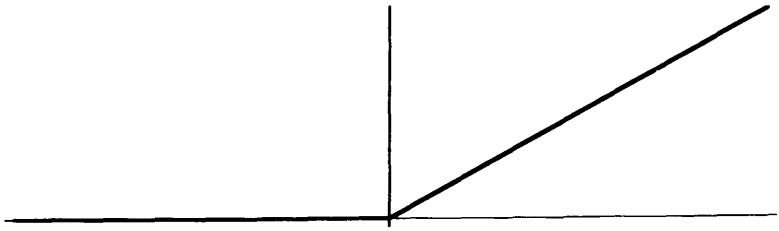

Figure 7.13 The penalty function $C\,\Phi(C)$ of equation (23)

5.1 Dealing with inequality constraints

In a neural formalism, an equality constraint $C=0$ with C some function of the spins, can be taken into account by adding an appropriate penalty term to the objective function. This can be chosen as a suitable function $\Phi(C)$ of the constraint variable, the obvious choice being $\Phi(C)=C^2$.

With an inequality constraint, $C \leqslant 0$, we need a $\Phi(C)$ that only penalizes configurations for which $C>0$. A possible choice is

$$\Phi(C)=C\Theta(C), \tag{23}$$

where Θ is the standard step function. This penalty function (Figure 7.13), multiplied by a suitable coefficient, gives a penalty in proportion to the degree of violation of the constraint.

Even if C happens to be a polynomial function of the spins, the nature of Φ ensures that the constraint term is not a polynomial function, and special care is needed when implementing the MF approximation. As shown by Ohlsson, Peterson & Söderberg [1993], this can be done by replacing the derivative $\partial E/\partial v_i$ in the MF equation (we assume binary ± 1 neurons):

$$v_i = g\left(-\frac{\partial E}{\partial v_i}; c\right)$$

by a difference:

$$\frac{\partial E}{\partial v_i} \to \frac{1}{2}\left(E(v)|_{v_i=1} - E(v)|_{v_i=-1}\right). \tag{24}$$

This is equivalent to assuming that E is a linear function of v_i between the extreme values ± 1. This trick also has the desirable side effect of killing all self-couplings, and can be used for any energy function, not necessarily with inequality constraints. It can easily be generalized to Potts systems, too.

5.2 The knapsack problem

In the knapsack problem one has a set of N *items* i, each with an associated *utility* $u_i>0$ and a set of *loads* $a_{ki}>0$, $k=1,\ldots,M$. The goal is to fill a knapsack with

a subset of the items such that their total utility,

$$U = \sum_{i=1}^{N} u_i s_i, \tag{25}$$

is maximized, subject to a set of M load constraints,

$$\sum_{i=1}^{N} a_{ki} s_i \leqslant b_k, \quad k = 1, \ldots, M, \tag{26}$$

defined by load *capacities* $b_k > 0$. The encoding is in terms of binary decision variables (spins) $s_i \in \{1, 0\}$, representing whether or not item i goes into the knapsack.

We will consider a class of problems, where a_{ki} and u_i are independent uniform random numbers on the unit interval $[0, 1]$, and b_k are fixed to a common value b. With $b > N/2$ the problem becomes trivial; the solution will have almost all $s_i = 1$. Conversely, with $b \ll N/4$ the number of allowed configurations will be small and an optimal solution can easily be found. We pick the most difficult case, defined by $b = N/4$. The expected number of items used in an optimal solution will then be about $N/2$, and an optimal solution becomes inaccessible for large N.

In the optimal solution to such a problem, there will be a strong correlation between the value of u_i and the probability for s_i to be 1. With a simple heuristic based on this observation, one can often obtain near-optimal solutions very fast. We will therefore also consider a class of harder problems with narrower u_i distributions, *homogeneous* problems. The extreme case is when u_i is independent of i, and the utility proportional to the number of items used.

We note in passing that the *set covering problem* is a special case of the general problem, with random $a_{ki} \in \{0, 1\}$, and $b_k = 1$. This defines a comparatively simple problem class, according to the above discussion, and we will stick to the knapsack problem in what follows.

Neural approach

A suitable energy function for the problem defined in (25) and (26) is

$$E(s) = -\sum_{i=1}^{N} u_i s_i + \alpha \sum_{k=1}^{M} \Phi\left(\sum_{i=1}^{N} a_{ki} s_i - b_k\right), \tag{27}$$

where the first term measures the utility, and the rest are constraint terms, with Φ the penalty function of (23) ensuring that the constraints in (26) are fulfilled. The coefficient α governs the relative strengths of the utility and constraint terms.

Minimizing (27) is achieved with the mean field equations,

$$v_i = g\left(-\frac{\partial E}{\partial v_i}; c\right),$$

though with differences substituted for derivatives, as in (24), and modified for

[0, 1] neurons. This leads to

$$\frac{\partial E}{\partial v_i} \to -u_i + \alpha \sum_{k=1}^{M} \left(\Phi\left(\sum_{j=1}^{N} a_{kj} v_j - b_k \right) \Bigg|_{v_i = 1} - \Phi\left(\sum_{j=1}^{N} a_{kj} v_j - b_k \right) \Bigg|_{v_i = 0} \right).$$

The modified MF equations are solved iteratively by annealing in c. Again there exists a more or less automated scheme for solving them (cf. Section 4.4). The high-c fixpoint analysis is somewhat more difficult in this case; we refer the reader to Ohlsson, Peterson & Söderberg [1993] for a discussion of this point.

Performance comparisons

Ohlsson, Peterson & Söderberg [1993] compare the neural network (NN) approach with several other methods. The *branch-and-bound* (BB) method is an optimization algorithm and consists of going down a search tree, checking bounds on constraints or utility for entire subtrees, thereby avoiding exhaustive search. In particular, for nonhomogeneous problems, this method is accelerated by ordering the utilities according to magnitude:

$$u_1 > u_2 > \cdots > u_N. \tag{28}$$

In a naive implementation, this implementation approach is only feasible for small problem instances. The *greedy heuristic* (GH) is a fast and dirty approximation method applicable to nonhomogeneous problems. Proceeding from larger to smaller u_i (cf. (28)), collect every item that does not violate any constraint. *Simulated annealing* (SA) [Kirkpatrick, Gelatt & Vecchi, 1983] is a stochastic method, implemented in terms of attempted single-spin flips, subject to the constraints. *Linear programming* (LP) based on the simplex method [Press et al., 1986] applies only to a modified problem with $s_i \in [0, 1]$. For the ordered (cf. (28)) nonhomogeneous knapsack problem it gives solutions with a set of leading 1's and a set of trailing 0's, with a window in between containing fractional numbers. Fairly good solutions emerge when augmented by greedy heuristics for the elements in this window.

Table 7.1 Comparison of performance and time consumption for the different algorithms on a knapsack problem with $N = M = 30$; the time consumption refers to a DEC3100 workstation

Algorithm	$u_i = \text{rand}[0, 1]$		$u_i = \text{rand}[0.45, 0.55]$		$u_i = 0.5$	
	Perf.	CPU time	Perf.	CPU time	Perf.	CPU time
BB	1	16	1	1500	1	1500
NN	0.98	0.80	0.95	0.70	0.97	0.75
SA	0.98	0.80	0.95	0.80	0.96	0.80
LP	0.98	0.10	0.93	0.25	0.93	0.30
GH	0.97	0.02	0.88	0.02	0.85	0.02

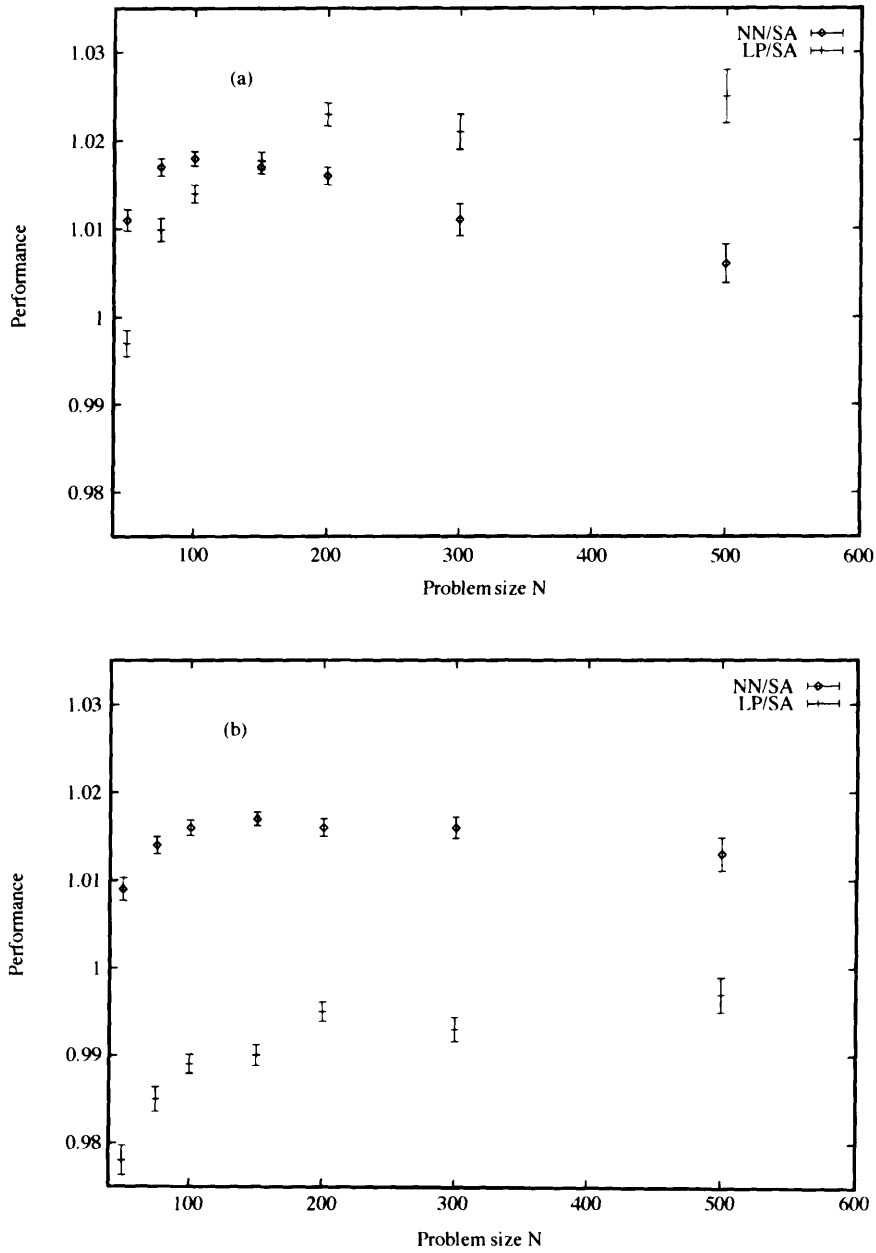

Figure 7.14 Performance of the neural network (NN) and linear programming (LP) approaches normalized to simulated annealing (SA) for problem sizes ranging from 50 to 500 with $M = N$: (a) $u_i = \text{rand}[0.45, 0.55]$ and (b) $u_i = 0.5$

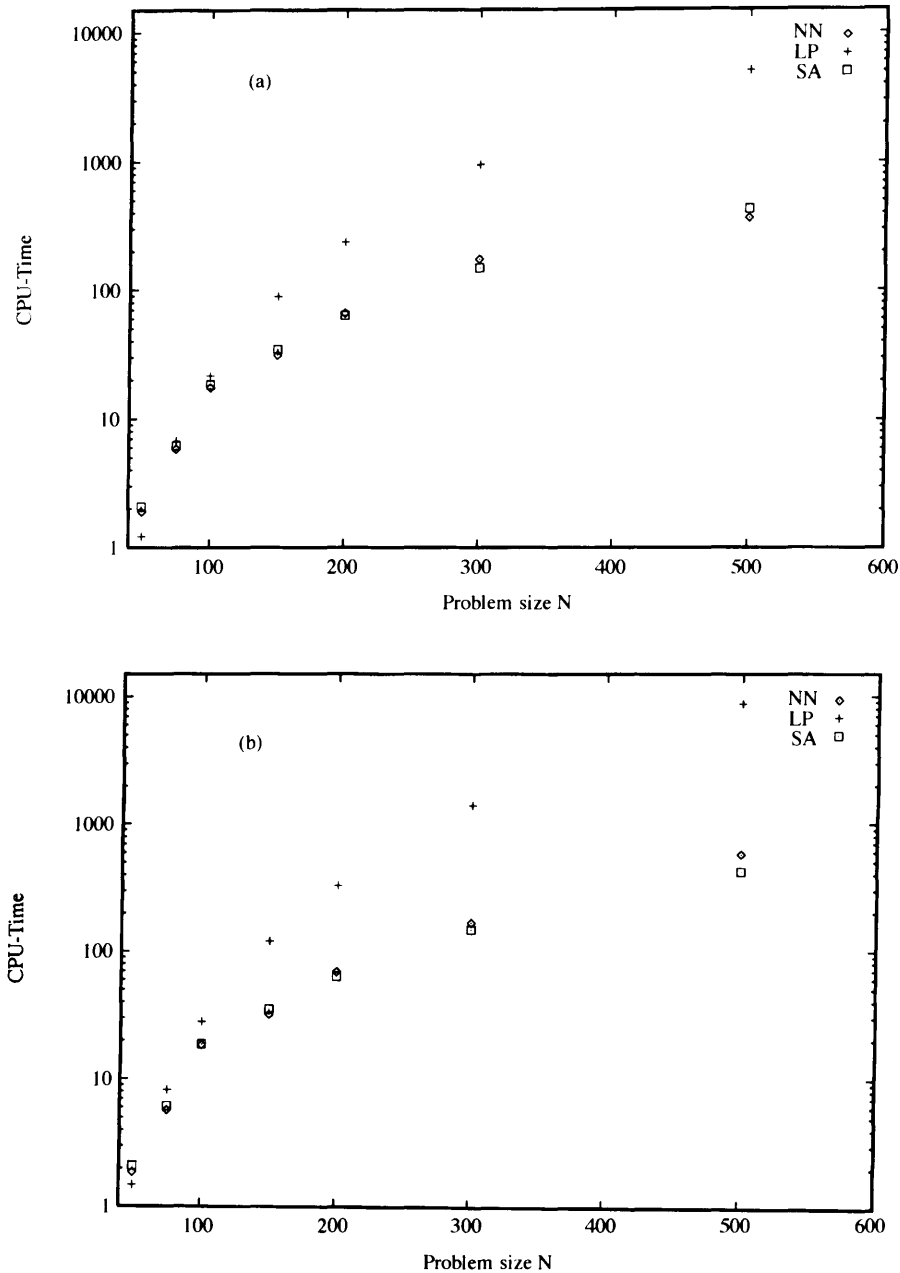

Figure 7.15 Time consumption of the neural network (NN) and linear programming (LP) approaches normalized to simulated annealing (SA) for problem sizes ranging from 50 to 500 with $M = N$: (a) $u_i = \text{rand}[0.45, 0.55]$ and (b) $u_i = 0.5$; the times are for a DEC3100 workstation

First of all, the NN, SA, and LP approaches are compared with the exact BB for $N = M = 30$, on homogeneous and nonhomogeneous problems; the results are shown in Table 7.1. Performance is measured as average utility normalized to BB (for large problems, normalized to SA).

LP and GH obviously benefit from nonhomogeneity in terms of quality and time (BB benefits on time), whereas the NN algorithm wins on homogeneous problems [Ohlsson, Peterson & Söderberg, 1993]. We cannot use BB for larger problem sizes, so only the approximative approaches are compared. The conclusions from problem sizes $N = M = 50$ to 500 are the same as above. The real strength of NN is best exploited for more homogeneous problems. Figures 7.14 and 7.15 show the performance and time consumption for $N \in [50, 500]$ with $M = N$.

In summary, the MF approach is competitive compared to other approximative methods for the hard homogeneous problems, both with respect to solution quality and time consumption. It also compares well with exact solutions, and the accessibility of an exact solution depends on the problem size. The ability to find good approximate solutions to difficult knapsack problems opens up several application areas within the field of resource allocation.

6 DEFORMABLE TEMPLATES

The above optimization problems were all treated as *pure assignment* problems – all variables were logical (neurons). In some areas it is advantageous to take a *parametric assignment* approach, which produces a hybrid picture where MF Potts decision variables are supplemented by geometric *template* variables. The TSP [Durbin & Willshaw, 1987] and particle physics track finding [Yuille, Honda & Peterson, 1991; Ohlsson, Peterson & Yuille, 1992; Gyulassy & Harlander, 1991] are examples where it pays to use the deformable templates approach. In the context of the TSP, this approach is often called an 'elastic net' [Durbin & Willshaw, 1987].

The deformable template formulation is rather application specific. Hence we will illustrate derivations, etc., using the TSP and track finding examples. Our presentation follows a general probabilistic path rather than the more intuitive approach in the original elastic network [Durbin & Willshaw, 1987].

6.1 The traveling salesman problem

Denote the N city positions in a TSP instance by $x_i \in \mathbb{R}^2$, $i = 1, \ldots, N$. The idea is to use a template trajectory, consisting of a closed chain of $M > N$ ordered points $\mathbf{y}_a \in \mathbb{R}^2$, $a = 1, \ldots, M$, in addition to a set of N distinct M-state Potts spins $\mathbf{s}_i$. The role of the Potts spins is to assign a point a in the chain to each city i, by $s_{ia} = 1$. The template coordinates $\mathbf{y}_a$ and the Potts spins $\mathbf{s}_i$ are to be chosen so as to minimize the chain length, while for each city i forcing the assigned $\mathbf{y}_a$ to match $\mathbf{x}_i$. In order to accomplish this the following energy expression is used:

$$E(\mathbf{s}, \mathbf{y}) = \sum_{ia} s_{ia} |\mathbf{x}_i - \mathbf{y}_a|^2 + \gamma \sum_a |\mathbf{y}_a - \mathbf{y}_{a+1}|^2. \tag{29}$$

The first term in (29) enforces matching: it is minimized when each $\mathbf{x}_i$ coincides with that $\mathbf{y}_a$ for which $s_{ia} = 1$. The second term minimizes the tour length while preserving an even spacing between the template points. The parameter γ governs the relative strength between the matching and tour length terms, and should be suitably chosen depending on M, N and the typical distances in the problem. In order to avoid getting trapped in local minima when minimizing E, noise is added by using the Boltzmann distribution for the system

$$\mathbb{P}(\mathbf{s}, \mathbf{y}; c) = \frac{e^{-E(\mathbf{s},\mathbf{y})/c}}{Z},$$

with the partition function Z given by

$$Z = \sum_{\mathbf{s}} \int d\mathbf{y} e^{-E(\mathbf{s},\mathbf{y})/c}.$$

Performing the trivial sums over the allowed states of the Potts spins [Yuille, 1990; Yuille, Honda & Peterson, 1991], we can rewrite Z as

$$Z = \int d\mathbf{y} e^{-E_{\text{eff}}(\mathbf{y})/c},$$

where the *effective energy* E_{eff} of the template is given by

$$E_{\text{eff}}(\mathbf{y}) = -c \sum_i \log\left(\sum_a e^{-|\mathbf{x}_i - \mathbf{y}_a|^2/c}\right) + \gamma \sum_a |\mathbf{y}_a - \mathbf{y}_{a+1}|^2. \tag{30}$$

Next we minimize E_{eff} with respect to $\mathbf{y}_a$ using gradient descent

$$\Delta \mathbf{y}_a = \eta\left(\sum_{ia} v_{ia}(\mathbf{x}_i - \mathbf{y}_a) + \gamma(\mathbf{y}_{a+1} - 2\mathbf{y}_a + \mathbf{y}_{a-1})\right), \tag{31}$$

where $\eta > 0$ is a suitable step size, and the Potts factor (cf. (20)) v_{ia} is given by

$$v_{ia} = \frac{e^{-|\mathbf{x}_i - \mathbf{y}_a|^2/c}}{\sum_b e^{-|\mathbf{x}_i - \mathbf{y}_b|^2/c}}. \tag{32}$$

This is all done under slow annealing in the temperature c, which enters the dynamics only via the Potts factors. A typical evolution of (31) and (32) is schematically depicted in Figure 7.16. In contrast to the Potts description of Section 4, the logical units are only implicit in the dynamics; the nontrivial dynamics lies with the analog variables $\mathbf{y}_a$. Here is an example of a complete algorithm.

An elastic net algorithm for the TSP

1. Choose problem instance $-\{\mathbf{x}_i, i = 1, N\}$.
2. Choose multiplicity M of template ($M \gg N$).
3. Choose a suitable γ and initial temperature.

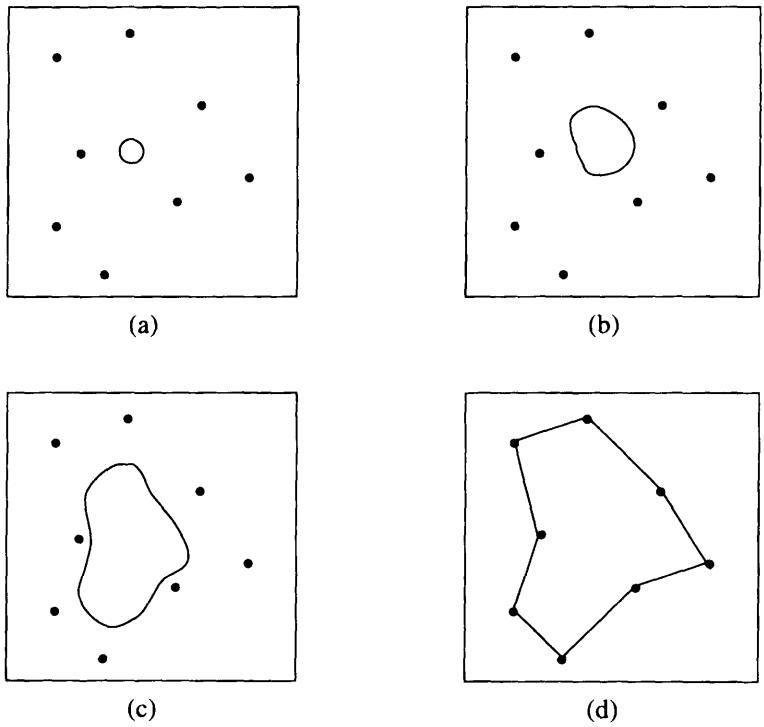

Figure 7.16 Evolution of the template chain from high to low temperatures; the dots denote city coordinates $\mathbf{x}_i$ and the template coordinate $\mathbf{y}_a$ reside on the closed curves

4. Place template points $\mathbf{y}_a$ equally spaced on a small circle with a slight random displacement from the center of gravity of the cities.
5. Until the system has settled, do:

   update the template points: $\mathbf{y}_a = \mathbf{y}_a + \Delta\mathbf{y}_a$, with $\Delta\mathbf{y}_a$ given by (31);
   $c = 0.9 \times c$.

How does this algorithm work? The first term in (30) contains a sum of Gaussians of width $\sqrt{c}$ around the templates. At high c this makes all the $\mathbf{y}_a$ compete to match all the cities; in effect, they will be attracted towards the center of mass. As c decreases, the Potts factors become more selective, and the range of competition for each $\mathbf{x}_i$ is focused on a smaller neighborhood. Finally one $\mathbf{y}_a$ is singled out to match each $\mathbf{x}_i$, and the remaining $\mathbf{y}$'s become arranged equidistantly along straight line segments connecting the cities.

This algorithm produces high-quality solutions [Peterson, 1990a]. And, very important, an N-city problem requires only $O(N)$ variables. It is a good example of a low-dimensional geometric problem where a template method is to be preferred over the pure neural method. See also Chapter 8, Section 7.

6.2 Track finding

Track finding is the problem of fitting smooth tracks to a given set of signal points. In its most general form – as in the problem of reconstructing the trajectories of airborne objects from radar signals – the parametric form of the tracks is unknown. A pure neural approach is appropriate in this case: assign a decision element (neuron) s_{ij} between two signals i and j, which is equal to 1 if i and j are connected and 0 otherwise. An energy function is then constructed in terms of s_{ij} such that smooth tracks will correspond to minima [Denby, 1988; Peterson, 1989]. Another possibility is to have a rotor neuron [Gislén, Söderberg & Peterson, 1992a] (see Section 7) associated with each signal, with an interaction such that smooth tracks emerge [Peterson, 1990b].

In particle physics experiments, a known magnetic field makes charged particles bend in tracks of known parametric form. The values of the track parameters contain information about the momenta of the particles. Even though the pure neural approach [Denby, 1988; Peterson, 1989; Stimpfl-Abele & Garrido, 1991] seems to work reasonably well, it is natural to use a deformable templates approach [Yuille, Honda & Peterson, 1991; Ohlsson, Peterson & Yuille, 1992; Gyulassy & Harlander, 1991] for the following reasons:

1. Not only does it solve the combinatorial optimization part of the problem, assigning signals to tracks, it also delivers the track parameters, which contain the physical quantities of interest.
2. The pure neural approach is more general than this problem requires. One should benefit from the fact that the parametric form of the tracks is known in advance, straight lines or helices.
3. The number of variables needed to solve an N-signal problem is large even with the connectivity restrictions imposed by Peterson [1989] and Stimpfl-Abele & Garrido [1991]. For a problem with N signals and M tracks one should only need $O(M)$ variables.
4. As demonstrated by Gyulassy & Harlander [1991], the neural approach is somewhat sensitive to noise. Again, with prior knowledge of the parametric form, the method should be more robust with respect to noise.

The strategy of the deformable templates approach for tracking [Yuille, Honda & Peterson, 1991; Gyulassy & Harlander, 1991] is to match the observed events to simple parameterized models, *templates*, the form of which reflects the a priori knowledge about the possible track shapes, e.g., helices passing through the origin (the collision point). In addition, the formalism allows for some data points (sensor noise) to be unmatched. The mechanism involved is closely related to redescending M-estimators used in robust statistics [Huber, 1981].

We assume we are given a set of N signal coordinates $\mathbf{x}_i \in \mathbb{R}^3$, $i = 1, \ldots, N$. For the case of a constant magnetic field, we define the templates, labeled by $a = 1, \ldots, M$, to be helices passing through the origin, parameterized each by an elevation angle θ_a, a transverse curvature κ_a, and a longitudinal velocity parameter γ_a. The assignment of templates a to the signals i is handled by an M-state

Potts spin $\mathbf{s}_i$ for each signal i. The algorithm works in two steps. First, for example a Hough transform [Duda & Hart, 1973] is used to determine the number M of templates required, and to initialize the templates (Hough transforms are essentially variants of 'histogramming' or 'binning' techniques, which are commonly applied to particle tracking). The elastic arms method then takes over, resolves ambiguities, and makes detailed fits to the signals, based on the following energy function (cf. (29)):

$$E(\mathbf{s}, \theta, \kappa, \gamma) = \sum_{i=1}^{N} \sum_{a=1}^{M} (M_{ia} - \lambda) s_{ia}, \tag{33}$$

where M_{ia}, short for $M(\theta_a, \kappa_a, \gamma_a, \mathbf{x}_i)$, is a measure of the squared distance between signal i and the helix a, and the Potts spin component s_{ia} is 1 if the ith point is assigned to the ath arm, and zero otherwise, subject to the *modified Potts constraint*,

$$\sum_a s_{ia} = 0 \text{ or } 1. \tag{34}$$

Thus, for each i there should be *at most* one a such that $s_{ia} = 1$. This allows for noise signals not being assigned to any track. It is equivalent to having an extra null component

$$s_{i0} = 1 - \sum_{a=1}^{M} s_{ia}$$

in the Potts spin, signifying no assignment. The parameter λ is a penalty for the nonassignment of a signal point. In effect a distance cutoff is introduced such that signals with no templates within the distance $\sqrt{\lambda}$ tend not to be assigned.

We want to minimize $E(\mathbf{s}, \theta, \kappa, \gamma)$ with respect to the Potts spins and the template parameters, subject to the constraint of (34). As in the TSP the problem is encoded in two kinds of variables, assignment variables s_{ia} and template parameters θ_a, κ_a, and γ_a. Following the steps of the TSP application, we define the Boltzmann distribution as

$$P(\mathbf{s}, \theta, \kappa, \gamma) = \frac{e^{-E(\mathbf{s},\theta,\kappa,\gamma)/c}}{Z}.$$

Summing over the Potts spins [Ohlsson, Peterson & Yuille, 1992], one obtains

$$P_M(\theta, \kappa, \gamma) = \frac{1}{Z} e^{-E_{\text{eff}}(\theta,\kappa,\gamma)/c},$$

where the effective energy E_{eff} amounts to

$$E_{\text{eff}}(\theta, \kappa, \gamma) = -c \sum_{i=1}^{N} \log\left(e^{-\lambda/c} + \sum_{a=1}^{M} e^{-M_{ia}/c} \right).$$

Straightforward gradient descent on E_{eff} with a step size η gives

$$\Delta\theta_a = -\eta \frac{\partial E_{\text{eff}}}{\partial \theta_a} = -\eta \sum_{i=1}^{N} v_{ia} \frac{\partial M_{ia}}{\partial \theta_a}$$

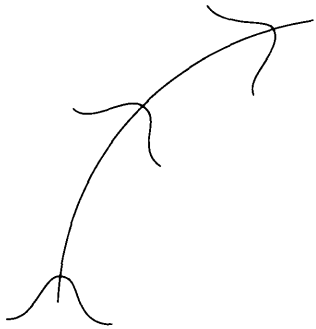

Figure 7.17 An elastic arm at temperature c

with the Potts factor

$$v_{ia} = \frac{e^{-M_{ia}/c}}{e^{-\lambda/c} + \sum_{b=1}^{M} e^{-M_{ib}/c}} \tag{35}$$

and analogous equations for $\Delta\kappa_a$ and $\Delta\gamma_a$. These are iterated under annealing in c. The structure of these equations is reminiscent of the TSP case, with decision elements v_{ia} intermixed with geometric fitting terms proportional to $\partial M_{ia}/\partial\theta_a$, $\partial M_{ia}/\partial\kappa_a$, and $\partial M_{ia}/\partial\gamma_a$. In a more sophisticated treatment, the step size η is replaced by a matrix, allowing for an interaction between the different parameters, so as to take into account an inhomogeneous metric in the space of parameters.

How does this algorithm work? At a high starting temperature a set of initial template arms are placed according to the Hough transform values for the parameters θ_a, κ_a, and γ_a. The templates compete for the signals by means of Gaussian distributions of width $\sqrt{c}$ centered around the arm positions (Figure 7.17). Initially each arm can attract many signals. The relative importance of the different signals is measured by the Potts factor (35). As the temperature is lowered, the different arms are attracted more exclusively to nearby signals only.

As discussed in connection with (33), λ governs the amount of noise points or *outliers* the algorithm allows for. It enters the Potts factor (35) via an extra component contributing to the denominator. For $\lambda \to \infty$ no signals are ignored and the extra component vanishes. For finite values of λ the domain of attraction of the arms is cut off (for small c) at a distance $\sqrt{\lambda}$.

The elastic arms approach is very similar to human processing for this kind of recognition problem. A human looks for helices in a global way and then makes fine-tuning adjustments. Indeed, when confronted with high-multiplicity data, the algorithm performs very well [Ohlsson, Peterson & Yuille, 1992].

7 ROTOR NEURONS

A binary (Ising) neuron can be considered as a vector living on a 'sphere' in one dimension. The MF approach can be generalized to variables defined on spheres in higher dimensions. Such *rotor* neurons may be used in geometrical optimization problems with angular variables.

7.1 Mean field rotor neurons

We consider the general problem of minimizing an energy function $E(\mathbf{s}_1, \ldots, \mathbf{s}_N)$ with respect to a set of N D-dimensional unit vectors (rotors), $\mathbf{s}_i \in \mathbb{R}^D$, $|\mathbf{s}_i| = 1, i = 1, \ldots, N$. To derive the MF equations, consider the partition function

$$Z = \int \mathrm{e}^{-E(\mathbf{s})/c} d\mathbf{s}_1 \cdots d\mathbf{s}_N,$$

where the integrations over $d\mathbf{s}_i$ are performed over the directions only. Along the same lines as in the binary (Ising) and Potts cases previously discussed [Gislén, Söderberg & Peterson, 1992a], rewrite Z as

$$Z \propto \int \exp\left(-E(\mathbf{v})/c - \sum_i \mathbf{v}_i \cdot \mathbf{u}_i + \sum_i F(u_i) \right) d\mathbf{v}_1 d\mathbf{u}_1 \cdots d\mathbf{v}_N d\mathbf{u}_N, \tag{36}$$

where $u_i = |\mathbf{u}_i|$ and

$$F(u) = \log I_{(D-2)/2}(u) - \frac{D-2}{2} \log u + \text{constant}. \tag{37}$$

In (37) I_n is the modified Bessel function of order n. Next we seek a saddle point of the effective potential in the exponent of (36) as in the previous cases, which leads to the *mean field* equations

$$\mathbf{u}_i = -\frac{1}{c} \nabla_i E(\mathbf{v}), \tag{38}$$

$$\mathbf{v}_i = \mathbf{g}(\mathbf{u}_i) \equiv \hat{u}_i g(u_i) \equiv \hat{u}_i F'(u_i), \tag{39}$$

where $\hat{u}_i = \mathbf{u}_i / u_i$. They give $\mathbf{v}_i$ as the average of $\mathbf{s}_i$ in the *local field* $-\nabla_i E(\mathbf{v})$. For $D = 1$ the standard sigmoid (cf. (2))

$$v_i = g(u_i) = \tanh u_i$$

is of course recovered.

The obvious dynamics consists of iterating (38) and (39) by annealing in the temperature c. At high temperature the system is in a symmetric phase, characterized by a stable trivial fixpoint $\mathbf{v}_i \approx 0$. At lower c this becomes unstable, and the mean fields $\mathbf{v}_i$ will be repelled by the origin. For low enough temperature they will stabilize close to the sphere $\mathbf{v}_i^2 = 1$ (Figure 7.18).

The dynamics is thus very different from conventional methods, where moves typically take place on the surface. Although the ability of exploring an 'off-shell'

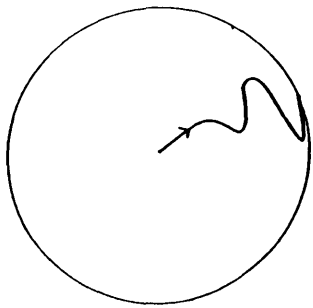

Figure 7.18 Schematic evolution of a $D = 2$ rotor initialized close to the center for $c < \hat{c}$

interpolating state space probably does not give as large an advantage as in the discrete Ising and Potts cases, it does provide an additional dimension through which to escape spurious local minima.

7.2 Applications

Gislén, Söderberg & Peterson [1992a] applied the rotor approach to the specific problem of the equilibrium configuration of N equal charges on a sphere ($D = 3$). Configurations were computed for 3, 5, 10, 20, 30, and 100 charges. The MF rotor model gave the correct solutions where they were explicitly known ($N = 2, 3, 4, 6$). For larger problems the solutions were compared to those from a local optimization (LO) and a simulated annealing (SA) algorithm.

The quality of the MF rotor algorithm turns out to be better or equal to algorithms produced by LO and SA. As for time consumption, the rotor algorithm empirically scales like N^2, whereas LO and SA scale like N^3 or worse.

Even though the rotor formalism applies to optimization problems that are not of a strictly combinatorial type, it is a natural extension of the binary decision neuron formalism that seems to be working well on the test beds explored. Possible realistic applications are molecular foldings and related problems.

8 SUMMARY AND OUTLOOK

Using neural networks to find good solutions to combinatorial optimization problems is an approach whose very nature differs from other approximative methods. The problems are mapped onto energy functions very similar to those of magnetic systems. Hence tools from statistical mechanics can be exploited. In addition to the simulated annealing method, one borrows the mean field approximation; and continuous variables feel their ways in a fuzzy manner towards good solutions. This is in contrast to most other methods, where moves are performed within the discrete solution space.

There are two main alternatives within the ANN paradigm, the pure ANN approach and deformable templates. For generic combinatorial optimization problems, such as scheduling and assignment problems, one uses the pure neural approach in which the basic steps are as follows:

- Map the problem onto a neural network by a suitable encoding of the solution space and an appropriate choice of energy function.
- Utilize prior knowledge about phase transition properties from analyzing the linearized dynamics. Special care is needed when defining the MF equations in cases when the penalty terms are nonpolynomial, as in the case of inequality constraints.
- While annealing solve the corresponding mean field equations iteratively.
- When the MF equations have converged, check the solutions for their legality, whether or not they satisfy the basic constraints. If not, supplement the algorithm with a greedy heuristic, or reanneal the system (possibly with modified constraint coefficients).

The MF equations of this method show an appealing similarity to the Kirchoff equations for analog VLSI circuitry.

For low-dimensional geometrical problems, like the traveling salesman problem and track finding problems, it is often advantageous to use a hybrid procedure, the deformable templates method. In this case, one has the following basic steps:

- Map the problem onto an energy function in terms of *both* template (geometric) and assignment (neural) variables. The dependence on the assignment variables is usually trivial.
- Form the corresponding partition function, and integrate (or sum) out the assignment variables to produce an effective energy in terms of template variables only.
- Minimize the effective energy with a gradient descent method. The templates will then be updated in such a way that the contributions from the data points are weighted with their relevance through Potts factors.

This template method has a lot in common with clustering and robust statistics methodologies as well as with Bayesian method [Ohlsson, Peterson & Yuille, 1992]. Not only is the pure neural approach closely related to analog VLSI, both the pure and hybrid approaches easily lend themselves to parallel execution.

With respect to quality of the solutions, the ANN methods produce very competitive results compared to other approximative schemes. See Peterson [1990a] and Ohlsson, Peterson & Söderberg [1993] for comparisons between methods of solving the traveling salesman and knapsack problems, respectively. No comparisons have been possible in full-size high-school scheduling applications, since no results exist from other approaches.

The neural language is very natural for encoding many combinatorial optimization problems. The approach has the advantage of being easily used for

revision of solutions due to new situations. One simply reanneals the network, clamping any nonrevisable units.

An Ising neuron can be considered as a one-dimensional rotor. The MF approach can also be extended to rotors in high dimensions. This can be fruitful when dealing with configurational problems with angular minimization.

By considering the MF approximation as resulting from a variational principle, the approach might be extended to systems with any type of discrete or continuous variables. Work in that direction has been pursued in molecular conformation problems [Bouchaud et al., 1991].

Local Search in Combinatorial Optimization
Edited by E. Aarts and J. K. Lenstra
© 1997 John Wiley & Sons Ltd.

8

The traveling salesman problem: a case study

David S. Johnson
AT & T Labs-Research, Florham Park

Lyle A. McGeoch
Amherst College, Amherst

1 INTRODUCTION 216
2 TOUR CONSTRUCTION HEURISTICS 219
2.1 Theoretical results 220
2.2 Four important tour construction algorithms 221
2.3 Experimental methodology 223
2.4 Implementation details 226
2.5 Experimental results for tour construction heuristics 228
3 2-OPT, 3-OPT, AND THEIR VARIANTS 230
3.1 Theoretical bounds on local search algorithms 231
3.2 Experimental results for 2-Opt and 3-Opt 234
3.3 How to make 2-Opt and 3-Opt run quickly 238
3.4 Parallel 2-Opt and 3-Opt 241
3.5 Other simple local optimization algorithms 246
4 TABU SEARCH AND THE LIN–KERNIGHAN ALGORITHM 249
4.1 Simple tabu search algorithms 251
4.2 The Lin–Kernighan algorithm 254
5 SIMULATED ANNEALING AND ITS VARIANTS 261
5.1 A baseline implementation of simulated annealing 264
5.2 Key speedup techniques 266
5.3 Other potential improvements 273
5.4 Threshold-based variants on annealing 281
6 GENETIC ALGORITHMS AND ITERATED LIN–KERNIGHAN 286
6.1 Filling in the schema 288
6.2 Production-mode Iterated Lin–Kernighan: experimental results 293
6.3 Further variants 298
7 NEURAL NETWORK ALGORITHMS 302
7.1 Neural networks based on integer programs 302
7.2 Geometric neural networks 305
8 CONCLUSION 309
ACKNOWLEDGMENTS 310

1 INTRODUCTION

In the traveling salesman problem (TSP) we are given a set $\{c_1, c_2, \ldots, c_N\}$ of *cities* and for each pair $\{c_i, c_j\}$ of distinct cities a *distance* $d(c_i, c_j)$. Our goal is to find an ordering π of the cities that minimizes the quantity

$$\sum_{i=1}^{N-1} d(c_{\pi(i)}, c_{\pi(i+1)}) + d(c_{\pi(N)}, c_{\pi(1)}).$$

This quantity is called the *tour length*, since it is the length of the tour a salesman would make when visiting the cities in the order specified by the permutation, returning at the end to the initial city. We shall concentrate in this chapter on the *symmetric* TSP, in which the distances satisfy $d(c_i, c_j) = d(c_j, c_i)$ for $1 \leqslant i, j \leqslant N$.

The symmetric traveling salesman problem has many applications, from very large scale integration (VLSI) chip fabrication [Korte, 1989] to X-ray crystallography [Bland & Shallcross, 1989], and a long history, for which see Lawler et al. [1985]. It is NP-hard [Garey & Johnson, 1979] so any algorithm for finding optimal tours must have a worst-case running time that grows faster than any polynomial (assuming the widely believed conjecture that P $\neq$ NP). This leaves researchers with two alternatives: either look for heuristics that merely find *near*-optimal tours, but do so quickly, or attempt to develop optimization algorithms that work well on 'real-world' rather than worst-case instances.

Because of its simplicity and applicability (or perhaps because of its intriguing name), the TSP has for decades served as an initial proving ground for new ideas related to both these alternatives. These new ideas include most of the local search variants covered in this book, which makes the TSP an ideal subject for a case study. In addition, the new ideas include many of the important advances in the related area of optimization, and to keep our discussions of local search in perspective, let us begin by noting the impressive recent progress in optimization algorithms.

The TSP is one of the major success stories for optimization. Decades of research into optimization techniques, combined with the continuing rapid growth in computer speeds and memory capacities, have led to one new record after another. Over the past 15 years, the record for the largest nontrivial TSP instance solved to optimality has increased from 318 cities [Crowder & Padberg, 1980] to 2392 cities [Padberg & Rinaldi, 1987] to 7397 cities [Applegate et al., 1995]. Admittedly, this last result took roughly 3–4 years of CPU time on a network of machines of the caliber of a SPARCstation 2. (SPARC is a trademark of SPARC International, Inc. and is licensed exclusively to Sun Microsystems, Inc.). However, the branch-and-cut technology developed for these record-setting performances has also had a major impact on the low end of the scale. Problems with 100 or fewer cities are now routinely solved within a few minutes on a workstation (although there are isolated instances in this range that take much longer) and instances in the 1000-city range typically take only a few hours (or days); see Padberg & Rinaldi [1991], Grötschel & Holland [1991], and Applegate et al. [1995].

These optimization results yield a twofold perspective. First, they weaken the appeal of the more costly heuristics, at least when the number of cities is 1000 or less, so where possible, we shall concentrate on results for instances with significantly more than 1000 cities. Second, they suggest that the TSP is not a *typical* combinatorial optimization problem, since most such problems seem significantly harder to solve to optimality.

Another way in which the TSP may be atypical lies in the high quality of results that can be obtained by traditional heuristics. The world of heuristic approaches to the TSP can be roughly divided into two classes. In addition to the *local search* approaches that are the topic of this book, there are many different *successive augmentation* heuristics for the TSP. Such heuristics build a solution (tour) from scratch by a growth process (usually a greedy one) that terminates as soon as a feasible solution has been constructed. In the context of the TSP, we call such a heuristic a *tour construction* heuristic. Whereas the successive augmentation approach performs poorly for many combinatorial optimization problems, in the case of the TSP many tour construction heuristics do surprisingly well in practice. The best typically get within roughly 10–15% of optimal in relatively little time. Furthermore, 'classical' local optimization techniques for the TSP yield even better results, with the simple 3-Opt heuristic getting with 3–4% of optimal and the 'variable-opt' algorithm of Lin & Kernighan [1973] typically getting with 1–2%. Moreover, for geometric data the aforementioned algorithms all appear to have running time growth rates that are $o(N^2)$, i.e., subquadratic, at least in the range from 100 to 1 000 000 cities. These successes for traditional approaches leave less room for new approaches like tabu search, simulated annealing, etc., to make contributions. Nevertheless, at least one of the new approaches, genetic algorithms, does have something to contribute if one is willing to pay a large, although still $o(N^2)$, price in running time.

In reaching these conclusions, we must take a hybrid approach. Where possible, we will report the results of performing experiments on common sets of instances on a fixed computer. The common test instances are the main ones used in a forthcoming study by Johnson et al. [1997b], and they will be described in more detail in the next section. The computer in question is an SGI Challenge machine containing sixteen 150 MHz MIPS R4400 processors, although our running times are for sequential implementations that use only a single one of these processors. (MIPS is a trademark of MIPS, Inc. and Challenge is a trademark of Silicon Graphics, Inc.). As a point of comparison, the MIPS processor can be 10–15 times faster than a SPARCstation 1 and perhaps 3–4 times faster than a SPARCstation 2. Our baseline experiments cover implementations of tour generation heuristics and classic local search algorithms written by Johnson et al. [1997b], implementations of simulated annealing written by Johnson et al. [1997a], and implementations of the GENI and GENIUS local search heuristics written by Gendreau, Hertz & Laporte [1992], who graciously provided us with their source code.

For many of the algorithms we cover, however, the only information we have is from published papers that provide only high-level descriptions of the algorithms

and summaries of experimental results. These papers usually do not provide enough information for us to compare the algorithms directly to our baseline implementations, and we are reduced to a process of deduction if we are to make comparisons. For tour quality comparisons, we are often fortunate in that other authors have generally used test instances that were similar in nature (if not identical) to the ones in our own test set. Running time comparisons are a bit more difficult, as rules of thumb for relating times between machines are far from exact and are highly dependent on the actual code, compilers, and operating systems used. (We used the standard IRIX operating system provided by SGI.) Moreover, many papers do not provide enough implementation details for us to know what sorts of optimization are present in (or missing from) their codes, and some papers are unclear as to whether preprocessing is included in their running times (as it is in ours). Comparisons involving results from such papers must often be based on unverified assumptions about the missing details. We shall generally try to be explicit about any such assumptions we make.

In comparing results, we shall also be interested in the relationship between what is known theoretically about the algorithms we study and what can be observed empirically. We shall see that each type of analysis has useful things to say about the other, although the worst-case nature of many theoretical results makes them far too pessimistic to tell us much about typical algorithmic behavior.

The remainder of this chapter is organized as follows. In Section 2 we discuss four important tour construction heuristics, along with key theoretical results that help characterize their behavior and that identify general complexity-theoretic limitations on what any heuristic approach can achieve. We also introduce our experimental methodology and summarize experimental results for the four heuristics from the extensive study by Johnson et al. [1997b]. Note that tour construction heuristics are important in the context of this book not only for the perspective they provide but also because they can be used to generate the starting points (initial tours) needed by local search algorithms and their variants. Section 3 describes *2-Opt* and *3-Opt*, the simplest and most famous of the classical local optimization algorithms for the TSP; it discusses their empirical behavior and what is known about them from a theoretical point of view. In addition to providing good results in their own right, these algorithms provide the essential building blocks used by many researchers in adapting tabu search, simulated annealing, etc., to the TSP. Section 4 is devoted to adaptations of tabu search to the TSP and to the Lin–Kernighan algorithm. Although the Lin–Kernighan algorithm was invented some 15 years before the introduction of tabu search, it embodies many of the same ideas, combining them with search-space truncation to yield what was for many years the 'champion' TSP heuristic.

A key question in each of the remaining sections is whether the approach under consideration can beat Lin–Kernighan. Given how well Lin–Kernighan performs, this means we will occasionally find ourselves forced to make distinctions between algorithms based on tour quality differences as small as 0.1%. Even though such differences may be statistically significant, we must admit that they

are unlikely to be of practical importance. Nevertheless, the intellectual appeal of discovering 'which is best' is hard to resist, and observations made here may suggest what can happen for other problems, where the ranges in solution quality may well be much wider. Section 5 surveys the results for various adaptations of simulated annealing and its variants to the TSP with special emphasis on the results and conclusions of Johnson et al. [1997a]. Section 6 discusses genetic algorithms and the *iterated local optimization* algorithms that have been derived from them, in particular the *Iterated Lin–Kernighan* algorithm. Section 7 surveys the wide variety of neural net algorithms that have been applied to the TSP. We conclude in Section 8 with a summary of how the best current adaptations of the various approaches compare for the TSP, and what if anything this means for other problem domains.

A wealth of algorithmic creativity has been applied to the TSP over the years, and by covering an extensive slice of it, as we do here, we hope at the very least to provide the reader with a useful source of more widely applicable ideas. As to the TSP itself, we can by no means claim to have been encyclopedic in our coverage. We have no doubt missed many interesting papers relevant to the topics we consider, and more papers have continued to appear since the main work on this chapter was completed. It is our hope, however, that the results and insights presented here will prove useful in putting such papers in perspective.

2 TOUR CONSTRUCTION HEURISTICS

Each TSP heuristic can be evaluated in terms of two key parameters: its running time and the quality of the tours it produces. Because of space limitations, we shall concentrate here on the most important *undominated* heuristics, where a heuristic is undominated if there is no competing heuristic which finds better tours and runs more quickly. Furthermore, we shall limit our main attention to just four heuristics, omitting those that only work for particular classes of instances, such as two-dimensional cases. The tour construction heuristics we cover in detail are Nearest Neighbor, Greedy, Clarke–Wright, and Christofides; each has a particular significance in the context of local search. The first three provide plausible mechanisms for generating starting tours in a local search procedure, and interesting lessons can be learned by evaluating them in this context. The fourth, in a sense, represents the best that tour construction heuristics can currently provide, and so it is a valuable benchmark. We shall allude briefly to several other tour construction heuristics, but readers interested in the full picture are referred to more extensive studies such as those of Bentley [1990a, 1992], Reinelt [1994], Jünger, Reinelt & Rinaldi [1995], and Johnson et al. [1997b].

We begin in Section 2.1 with a discussion of the general complexity-theoretic limitations imposed on all heuristics. Section 2.2 then presents brief descriptions of our key tour construction heuristics and summarizes what is known about their performance from a theoretical point of view. Section 2.3 discusses the key methodological issues involved in evaluating heuristics from an experimental

point of view. In particular, we discuss the method we use for measuring tour quality and the particular instance classes we use for our primary experimental test beds. Section 2.4 describes some of the key implementation details needed to make the heuristics run quickly on instances such as those in our test beds. Section 2.5 then summarizes the specific results obtained by Johnson et al. [1997b] for the heuristics of Section 2.2.

2.1 Theoretical results

Two fundamental complexity-theoretic results constrain the behavior of *any* heuristic for the TSP. For a given heuristic A and TSP instance I, let $A(I)$ denote the length of the tour produced by A and let OPT (I) denote the length of an optimal tour. The first result concerns the best performance guarantee that is possible when there are no restrictions on the types of instances considered.

Theorem A [Sahni & Gonzalez, 1976] *Assuming* $\mathrm{P} \neq \mathrm{NP}$, *no polynomial-time TSP heuristic can guarantee* $A(I)/\mathrm{OPT}\,(I) \leqslant 2^{p(N)}$ *for any fixed polynomial p and all instances I.*

Fortunately, most applications impose substantial restrictions on the types of instances allowed. In particular, distances must usually obey what is called the *triangle inequality*. This says that for all i, j, k, $1 \leqslant i, j, k \leqslant N$, $d(c_i, c_j) \leqslant d(c_i, c_k) + d(c_k, c_j)$, i.e., the direct path between two cities is always the shortest route. (Even in real-world situations where the shortest physical route from city c_i to city c_j must pass through city c_k, we typically ignore such details in formulating the corresponding TSP instance and simply take $d(c_i, c_j)$ to be the length of the shortest path between c_i and c_j, not the length of the shortest physical route that avoids all other cities.) Thus much of the theoretical work on TSP heuristics is predicated on the assumption that the triangle inequality holds. In this case the result of Sahni & Gonzalez [1976] no longer applies, and polynomial-time algorithms are known that guarantee $A(I)/\mathrm{OPT}(I) \leqslant 1.5$ [Christofides, 1976]. The only known constraint is the following much more limited (and recent) one, derived as a consequence of the deep connection between approximation and the characterization of NP in terms of probabilistically checkable proof systems.

Theorem B [Arora et al., 1992] *Assuming* $\mathrm{P} \neq \mathrm{NP}$, *there exists an* $\varepsilon > 0$ *such that no polynomial-time TSP heuristic can guarantee* $A(I)/\mathrm{OPT}\,(I) \leqslant 1 + \varepsilon$ *for all instances I satisfying the triangle inequality.*

Even this lower bound evaporates if one is willing to consider the further restriction to instances consisting of points in the plane under a geometric norm such as the Euclidean metric, a condition holding in many applications. For such instances, Arora [1996] has proved that *polynomial-time approximation schemes* exist. Stated in Euclidean terms, Arora's result is as follows.

Theorem C [Arora, 1996] *There is an algorithm A that, given a Euclidean TSP instance and a constant* $\varepsilon > 0$, *runs in time* $n^{O(1/\varepsilon)}$ *and guarantees* $A(I)/\mathrm{OPT}\,(I) < 1 + \varepsilon$.

Unfortunately, Arora's result appears to be of only theoretical significance, both because of the constants built into the $O(\ )$ notation and because the algorithm also requires space $n^{O(1/\varepsilon)}$. Thus in practice, we may have to settle for algorithms with worse guarantees but better time and space requirements. In this chapter, we shall concentrate on four such tour construction heuristics: *Nearest Neighbor*, *Greedy*, *Clarke–Wright*, and *Christofides*. They provide a wide range of guarantees for instances obeying the triangle inequality and exhibit interesting trade-offs between running time and average tour quality for Euclidean instances. For a more comprehensive study of tour construction heuristics, see Johnson et al. [1997b].

2.2 Four important tour construction algorithms

Nearest Neighbor

Perhaps the most natural heuristic for the TSP is the famous *Nearest Neighbor* algorithm (NN). In this algorithm one mimics the traveler whose rule of thumb is always to go next to the nearest as-yet-unvisited location. We construct an ordering $c_{\pi(1)}, \ldots, c_{\pi(N)}$ of the cities, with the initial city $c_{\pi(1)}$ chosen arbitrarily and in general $c_{\pi(i+1)}$ chosen to be the city c_k that minimizes $\{d(c_{\pi(i)}, c_k)\colon k \neq \pi(j), 1 \leqslant j \leqslant i\}$. The corresponding tour traverses the cities in the constructed order, returning to $c_{\pi(1)}$ after visiting city $c_{\pi(N)}$.

The running time for NN as described is $\Theta(N^2)$. If we restrict attention to instances satisfying the triangle inequality, NN does substantially better than the general upper bound of Theorem A, although it is still far worse than the limit provided by Theorem B. In particular, we are guaranteed that $\mathrm{NN}(I)/\mathrm{OPT}(I) \leqslant 0.5(\lfloor \log_2 N \rfloor + 1)$. No substantially better guarantee is possible, however, as there are instances for which the ratio grows as $\Theta(\log N)$ [Rosenkrantz, Stearns & Lewis, 1977].

Greedy

Some authors use the name *Greedy* for Nearest Neighbor, but it is more appropriately reserved for the following variant on the 'greedy algorithm' of matroid theory. In this heuristic, we view an instance as a complete graph with the cities as vertices and with an edge of length $d(c_i, c_j)$ between each pair $\{c_i, c_j\}$ of cities. A tour is then simply a Hamiltonian cycle in this graph, i.e., a connected collection of edges in which every city has degree 2. We build up this cycle one edge at a time, starting with the shortest edge, and repeatedly adding the shortest remaining available edge, where an edge is *available* if it is not yet in the tour and if adding it would not create a degree-3 vertex or a cycle of length less than N. In view of the intermediate partial tours typically constructed by this heuristic, it is called the *multifragment* heuristic by Bentley [1990a, 1992].

The Greedy heuristic can be implemented to run in time $\Theta(N^2 \log N)$ and is thus somewhat slower than NN. On the other hand, its worst-case tour quality

may be somewhat better. As with NN, it can be shown that greedy(I)/OPT(I) $\leqslant (0.5)(\lceil \log_2 N \rceil + 1)$ for all instances I obeying the triangle inequality [Ong & Moore, 1984], but the worst examples known for Greedy only make the ratio grow as $(\log N)/(3 \log\log N)$ [Frieze, 1979].

Clarke–Wright

The *Clarke–Wright savings heuristic* (Clarke–Wright or simply CW for short) is derived from a more general vehicle routing algorithm due to Clarke & Wright [1964]. In terms of the TSP, we start with a pseudotour in which an arbitrarily chosen city is the *hub* and the salesman returns to the hub after each visit to another city. (In other words, we start with a multigraph in which every nonhub vertex is connected by two edges to the hub.) For each pair of nonhub cities, let the *savings* be the amount by which the tour would be shortened if the salesman went directly from one city to the other, bypassing the hub. We now proceed analogously to the Greedy algorithm. We go through the nonhub city pairs in nonincreasing order of savings, performing the bypass so long as it does not create a cycle of nonhub vertices or cause a nonhub vertex to become adjacent to more than two other nonhub vertices. The construction process terminates when only two nonhub cities remain connected to the hub, in which case we have a true tour.

As with Greedy, this algorithm can be implemented to run in time $\Theta(N^2 \log N)$. The best performance guarantee currently known (assuming the triangle inequality) is CW(I)/OPT(I) $\leqslant \lceil \log_2 N \rceil + 1$ (a factor of 2 higher than that for Greedy) [Ong & Moore, 1984], but the worst examples known yield the same $(\log N)/(3 \log\log N)$ ratio as obtained for Greedy [Frieze, 1979].

Christofides

The previous three algorithms all have worst-case ratios that grow with N even when the triangle inequality holds. Theorem B does not rule out much better performance, however, and in fact a large class of algorithms do perform much better. As observed by Rosenkrantz, Stearns & Lewis [1977], there are at least three simple polynomial-time tour generation heuristics, *Double Minimum Spanning Tree*, *Nearest Insertion*, and *Nearest Addition*, that have worst-case ratio 2 under the triangle inequality. That is, they guarantee $A(I)/\mathrm{OPT}(I) \leqslant 2$ under that restriction, and there exist instances with arbitrarily large values of N that show that this upper bound cannot be improved. We do not discuss these heuristics in detail since they are all dominated in practice by NN, Greedy, and CW, despite the fact that their worst-case performance is so much better.

One tour construction heuristic with a constant worst-case performance ratio is not so dominated. This is the algorithm of Christofides [1976], the current champion as far as performance guarantee is concerned, having a worst-case ratio of just 3/2 assuming the triangle inequality, a tight bound even for Euclidean instances [Cornuéjols & Nemhauser, 1978]. The Christofides heuristic proceeds

as follows. First, we construct a minimum spanning tree T for the set of cities. Note that the length of such a tree can be no longer than OPT(I), since deleting an edge from an optimal tour yields a spanning tree. Next, we compute a minimum-length matching M on the vertices of odd degree in T. It can be shown by a simple argument that, assuming the triangle inequality, this matching will be no longer than OPT(I)/2. Combining M with T we obtain a connected graph in which every vertex has even degree. This graph must contain an Euler tour, i.e., a cycle that passes through each edge exactly once, and such a cycle can be easily found. A traveling salesman tour of no greater length can then be constructed by traversing this cycle while taking shortcuts to avoid multiply visited vertices. A *shortcut* replaces a path between two cities by a direct edge between the two. By the triangle inequality the direct route cannot be longer than the path it replaces.

Not only does the Christofides algorithm provide a better worst-case guarantee than any other currently known tour construction heuristic, it also tends to find better tours in practice, assuming care is taken in the choice of shortcuts. Its running time cost is substantial, however, compared to those for Nearest Neighbor, Greedy, and Clarke–Wright. This is primarily because the best algorithms currently available for its matching step take time $\Theta(N^3)$ [Edmonds, 1965; Gabow, 1973; Lawler, 1976], whereas none of the other three algorithms takes more than $O(N^2 \log N)$ time. In theory this running time gap can be reduced somewhat; a modification of the Christofides algorithm with the same worst-case guarantee and an $O(N^{2.5})$ running time can be obtained by using a scaling-based matching algorithm and halting once the matching is guaranteed to be no longer than $1 + (1/N)$ times optimal [Gabow & Tarjan, 1991]. As far as we know, however, this approach has never been implemented, and as we shall see, the competition from local search algorithms is sufficiently strong that the programming effort needed to do so would not be justified.

2.3 Experimental methodology

The Held–Karp lower bound

When evaluating the empirical performance of heuristics, we are often not allowed the luxury of comparing to the precise optimal tour length, as we did in the above theoretical results, since for large instances we typically do not *know* the optimal tour length. Consequently when studying large instances, it has become the practice to compare heuristic results to something we can compute: the lower bound on the optimal tour length due to Held & Karp [1970, 1971].

This bound is the solution to the standard linear programming relaxation of the TSP. For instances of moderate size it can be computed exactly using linear programming, although if one goes about this directly one is confronted with a nontrivial computation: the number of constraints in the linear program is exponential in N. A more practical approach is to solve a sequence of restricted linear programs (LPs), each involving only a subset of the constraints, and to use a separation subroutine to identify violated constraints that need to be included

in the next LP. This approach has been implemented both by Reinelt [1994] and by Applegate et al. [1994] using the simplex method to solve the linear programs. Exact values for the bound have been computed in this way for instances as large as 33 810 cities [Johnson, McGeoch & Rothberg, 1996], including all instances in our test beds up to this size. For larger instances, we settle for an approximation to the Held–Karp bound (a lower bound on the lower bound) computed by an iterative technique proposed in the original Held–Karp papers and sped up by a variety of algorithmic tricks [Helbig-Hansen & Krarup, 1974; Held, Wolfe & Crowder, 1974; Johnson, McGeoch & Rothberg, 1996]. Based on results for instances where the true Held-Karp (HK) bound is known, we expect that for those instances in our test beds where we must rely on this approximation, it is within 0.01% or less of the true bound. What is more important, the Held–Karp bound itself appears to provide a consistently good approximation to the optimal tour length. From a worst-case point of view, the Held–Karp bound can never be smaller than (2/3)OPT(I), assuming the triangle inequality [Wolsey, 1980; Shmoys & Williamson, 1990]. In practice, it is typically far better than this, even when the triangle inequality does not hold. We shall see just how much better in the next section, where we discuss our main test beds.

Standard test instances

When we talk about experimental results in this chapter, we shall for the most part be talking about behavior on instances that not only obey the triangle inequality but are also geometric in nature; in typical instances the cities correspond to points in the plane and distances are computed under a standard metric such as the Euclidean or rectilinear norm. Many of the applications of the symmetric TSP are of this sort, and most recent published studies have concentrated on them, using two main sources of such instances. The first source consists simply of randomly generated instances, where the cities have their locations chosen uniformly in the unit square, with distances computed under the Euclidean metric. The second source is a database of instances called `TSPLIB` collected by Reinelt [1991] and available via anonymous ftp from `softlib.rice.edu`. `TSPLIB` contains instances with as many as 85 900 cities, including many from printed circuit board and VLSI applications, as well as geographical instances based on real cities. Results are surprisingly consistent between the two sources. For example, NN averages less than 24% above the Held–Karp lower bound on random Euclidean instances with N ranging from 10 000 to 1 000 000; for a selection of 15 of the largest two-dimensional instances from Version 1.2 of `TSPLIB` (including all 11 with $N > 3\,000$), NN averaged roughly 26% above.

Many papers covering geometric instances of the above two types do not use the Held–Karp bound as their standard of comparison. Researchers dealing with `TSPLIB` instances often restrict attention to those for which optimal solutions are known, comparing their results to the optimal tour lengths. Fortunately, the exact Held–Karp bounds are known for all of these instances, so it is easy to translate from one sort of comparison to the other. The average gap over all 89

instances in TSPLIB (solved or unsolved) is 0.82% or less, with all but two of the gaps being less than 1.76% [Johnson, McGeoch & Rothberg, 1996].

More difficult to deal with are those papers that study random Euclidean instances and compare their results only to the expected optimal tour length or, more precisely, to estimates thereof. Even if the estimated expected values were correct, the natural variation between instances of a given size means that the expected value would not be a totally reliable estimate for the optimal tour length of any specific instance. Adding about 0.7% to the Held–Karp lower bound for that instance would yield a much more reliable estimate [Johnson, McGeoch & Rothberg, 1996]. Furthermore, the estimates used are typically far from correct. It is known from work of Beardwood, Halton & Hammersley [1959] that for these instances the ratio of the optimal tour length to $\sqrt{N}$ approaches a constant C_{OPT} as $N \to \infty$, but the most frequently quoted estimates for this constant, 0.749 by Beardwood et al. and 0.765 by Stein [1977], are both significant overestimates. Recent experiments of Percus & Martin [1996] and Johnson, McGeoch & Rothberg [1996] suggest that the actual limit is more like 0.7124. Thus many claims of closeness to optimality are too optimistic by 5% or more, and we will have to reinterpret such claims in this light. In doing so, we shall rely on the following formula for the Held–Karp bound, derived empirically by Johnson, McGeoch & Rothberg [1996]. Let $C_{\mathrm{HK}}(N)$ be the expected ratio of the Held–Karp bound to $\sqrt{N}$ for N-city random Euclidean instances. Then for $N \geqslant 100$,

$$C_{\mathrm{HK}}(N) \sim 0.708\,05 + \frac{0.522\,29}{N^{0.5}} + \frac{1.315\,72}{N} - \frac{3.074\,74}{N^{1.5}}.$$

As this formula suggests, a result analogous to that of Beardwood et al. holds for the Held–Karp lower bound, with the ratio of the expected Held–Karp bound to $\sqrt{N}$ approaching a constant $C_{\mathrm{HK}} < C_{\mathrm{OPT}}$ as $N \to \infty$ [Goemans & Bertsimas, 1991]. Similar results (with constants bigger than C_{OPT}) may also hold for many TSP heuristics, although the only heuristic for which this question has been successfully analyzed does not quite live up to this expectation. For the *Spacefilling Curve* heuristic of Platzman & Bartholdi [1989], $\limsup_{N\to\infty} \mathbb{E}[A(L_N)]/N^{1/2}$ and $\liminf_{N\to\infty} \mathbb{E}[A(L_N)]/N^{1/2}$ are both constants, but they are *different* constants, and so there is no true limit. (The difference between the constants is less than 0.02%, however).

In addition to the geometric instances mentioned above, we shall also consider another test bed that is frequently encountered in the literature. These are instances in which the distances between cities are all independent, chosen randomly from the uniform distribution on [0, 1]. Although these *random distance matrix* instances do not have any apparent relevance, they are interesting from a theoretical point of view. One can prove that the expected length of an optimal tour is bounded, independently of N, and statistical mechanical arguments suggest that it approaches a limiting value of 2.0415... [Krauth & Mézard, 1989]. This value agrees closely with the experimental results of Johnson, McGeoch & Rothberg [1996]. Since these instances do not typically

obey the triangle inequality, they offer a strikingly different challenge to our heuristics. The Held–Karp bound remains a good estimate of the optimal tour length, better even than it was for the geometric case since the difference between it and the optimal tour length appears to be approaching 0 as $N \to \infty$. The performance of heuristics, however, appears to degrade markedly. For example, the expected percentage excess over the Held–Karp bound for NN can be proved to grow as $\Theta(\log N)$ for such instances, which is clearly much worse than the 24–26% (independent of N) observed for random Euclidean instances.

For the above two classes of random instances, our test beds consist of instances for which N is a power of $\sqrt{10}$, rounded to the nearest integer. They contain several instances of each size, and the results we report are typically averages over all instances of a given size and type. The numbers of random Euclidean instances range from 21 instances with $N = 100$ to one each for $N = 316\,228$ and $1\,000\,000$. The use of fewer instances for larger values of N is made possible by the fact that the normalized variation in behavior between random instances of the same size declines with N [Steele, 1981]. More specifically, the probability that $\mathrm{OPT}(I)/\sqrt{N}$ differs from its expected value by more than t is at most $Ke^{-t^2N/K}$ for some fixed constant K [Rhee & Talagrand, 1988]. The number of instances in the test bed is not enough to guarantee high-precision estimates of average-case behavior, but is enough for the distinctions we wish to make. For random distance matrices there is even less need for precision, and we typically use just two instances each for all the sizes considered, from 100 to 31 623 cities.

Many of the observations we shall make based on the above test beds carry over to other classes of instances. For a more complete study that covers a variety of these other classes, see Johnson et al. [1997b].

2.4 Implementation details

As observed in Section 2.2, all our tour construction heuristics would seem to require at least $\Theta(N^2)$ time for an arbitrary instance, since in general every edge is a potential candidate for membership in the tour and there are $N(N-1)/2$ edges to be considered. This is not necessarily the case, however, for geometric instances like those in the first two classes of our test bed, where cities correspond to points in the plane or some higher-dimensional space and where the distance between two cities is a function of the coordinates of the cities. For such instances it may be possible to rule out large classes of edges quickly, by exploiting the geometry of the situation with appropriate data structures.

In particular, for points in the plane it is possible in $O(N \log N)$ time to construct a data structure that allows us to answer all subsequent nearest neighbor queries in average time much less than $\Theta(N)$ per query. One such data structure is the k–d tree of Bentley [1975, 1990b], exploited in the implementations of NN, Greedy, and Clarke–Wright described by Bentley [1990a], Bentley [1992] and Johnson et al. [1997b]. Another is the Delaunay triangulation, exploited in implementations of approximate versions of these algorithms by

Reinelt [1992, 1994] and Jünger, Reinelt & Rinaldi [1995]. In the case of the implementations using k–d trees, on which we shall concentrate here, the number of operations required for finding nearest neighbors typically averages around $\Theta(\log N)$. (For large instances, the actual *time* to do the computation grows faster on our machine than does the operation count, due to increasing numbers of cache misses and other memory hierarchy effects.)

Using such a data structure, all four of our heuristics can be sped up substantially. It is relatively easy to see how this is done for Nearest Neighbor. Greedy and Clarke–Wright are sped up by using nearest neighbor (and related) queries to stock and update a priority queue that maintains an entry for each city that does not yet have degree 2 in the current tour. The entry for city c_i is a triple $\langle c_i, c_j, d(c_i, c_j)\rangle$, where c_j is c_i's nearest neighbor (in the case of Clarke–Wright, its neighbor yielding maximum savings).

For Christofides, a similar priority queue based approach can be used to compute the minimum spanning tree, although computing the minimum-length matching on the odd-degree vertices remains the bottleneck. For this we use the fast geometric matching code of Applegate & Cook [1993], although even with this fast code the largest instances we can feasibly handle involve only 100 000 cities. In addition, our implementation of Christofides uses a slightly more sophisticated short-cutting mechanism in its final stage: instead of simply traversing the Euler tour and skipping past previously visited cities; we choose the shortcut for each multiply visited city that yields the biggest reduction in tour length. This innovation yields an improvement of roughly 5% in tour length for random geometric instances, and it is crucial if Christofides is to maintain its undominated status. Ideally, we would have liked to find the *best* possible way of doing the shortcuts, but that task in NP-hard [Papadimitriou & Vazirani, 1984].

Different implementation issues arise with our random distance matrix instances. In particular, there is the problem of storing them. We would like to get an idea of algorithmic behavior on large instances of this type, but a 10 000-city instance would take roughly 200 megabytes, at one 32-bit word per edge. We have thus chosen not to store instances directly, but instead to generate them on the fly using a subroutine that, given i, j, and a seed s unique to the instance, computes $d(c_i, c_j)$ in a reproducible fashion. For details, see Johnson et al. [1997b]. By allowing us to consider instances whose edge sets are too large to store in main memory, this scheme influences the algorithm implementations we use.

For NN we can still use the straightforward $\Theta(N^2)$ implementation. For greedy and CW, however, we cannot use the straightforward $\Theta(N^2 \log N)$ implementations, as these would require us to sort all $\Theta(N^2)$ edges, and paging costs become prohibitive if the edges cannot all fit in main memory. Thus for these algorithms we mimic the corresponding geometric implementations, replacing the k–d data structure by one consisting of sorted lists of the 20 nearest neighbors for each city, computed in overall time $\Theta(N^2)$. To find the nearest legal neighbor of city c_i, we then can simply search down the list for c_i. Only if no c_j on the list is such that the edge $\{c_i, c_j\}$ can be added to the current tour do we resort to a full

$\Theta(N)$ search of all possibilities. We did not implement Christofides for this last class of instances, for reasons we explain below.

2.5 Experimental results for tour construction heuristics

In this section we summarize the results obtained by Johnson et al. [1997b] for our four heuristics on random Euclidean instances and random distance matrices, as described above. Table 8.1 covers tour quality and Table 8.2 covers running times in seconds on our SGI Challenge machine. Running times do not include the time needed to read in the instance or write out the resulting tour, which were only a small part of the overall time and could have been reduced almost to negligibility by clever coding of the input and output routines. Results for instances from TSPLIB were similar to those for the random Euclidean instances; running times for instances of similar size were comparable, with the tour quality for NN and Greedy being slightly worse on average than for the random instances, and the tour quality for CW and Christofides being slightly better.

The results presented are averages taken over all the instances in our test bed of each given size. For NN and CW we also average over 10 or more random choices of the starting city for each instance. Nevertheless, the accuracy of the least significant digits in the numbers reported should be viewed with some skepticism, and it is probably safer to draw conclusions from the overall picture rather than from any particular value in the tables. This is especially so for the random distance matrix results, which are based on fewer instances of each size.

For random Euclidean instances, each of the four algorithms seems to approach a relatively small limiting value for the average percentage by which its tours exceed the Held–Karp lower bound, with the best of these limits (that for Christofides) being slightly under 10%. (The result for Christofides would be more like 15% if we had not used the greedy short-cutting method mentioned in

Table 8.1 Tour quality for tour generation heuristics: average percent excess over the Held–Karp lower bound

$N =$	10^2	$10^{2.5}$	10^3	$10^{3.5}$	10^4	$10^{4.5}$	10^5	$10^{5.5}$	10^6
				Random Euclidean instances					
CHR	9.5	9.9	9.7	9.8	9.9	9.8	9.9		
CW	9.2	10.7	11.3	11.8	11.9	12.0	12.1	12.1	12.2
GR	19.5	18.8	17.0	16.8	16.6	14.7	14.9	14.5	14.2
NN	25.6	26.2	26.0	25.5	24.3	24.0	23.6	23.4	23.3
				Random distance matrices					
GR	100	160	170	200	250	280			
NN	130	180	240	300	360	410			
CW	270	520	980	1800	3200	5620			

Table 8.2 Running times for tour generation heuristics: seconds on a 150 MHz SGI Challenge

$N =$	10^2	$10^{2.5}$	10^3	$10^{3.5}$	10^4	$10^{4.5}$	10^5	$10^{5.5}$	10^6
			Random Euclidean instances						
CHR	0.03	0.12	0.53	3.57	41.9	801.9	23 009	-	
CW	0.00	0.03	0.11	0.35	1.4	6.5	31	173	670
GR	0.00	0.02	0.08	0.29	1.1	5.5	23	90	380
NN	0.00	0.01	0.03	0.09	0.3	1.2	6	20	120
			Random distance matrices						
GR	0.02	0.12	0.98	9.3	107	1 400	-	-	
NN	0.01	0.07	0.69	7.2	73	730			
CW	0.03	0.24	2.23	22.0	236	2 740	-	-	

the previous section.) For all the heuristics except Christofides, results for $N = 100$ are relatively poor predictors of how good the solutions will be when N is large. This is especially true for Greedy, whose average excess drops from 19.5% to 14.2% as one goes from $N = 100$ to $N = 1\,000\,000$. As to running times, the time for Christofides is growing faster than N^2 as expected, but the rest have running times that grow subquadratically. Quadratic growth requires the running time to increase by a factor of 10 each time N goes up by a factor of $\sqrt{10}$; this does not occur in Table 8.2. Indeed, based on actual operation counts, the running times should be more like $\Theta(N \log N)$, although memory hierarchy effects cause the running times to grow significantly faster than this once $N > 10\,000$. Even so, the slowest of NN, Greedy, and CW takes only about 10 minutes on the million-city instance.

The story for random distance matrices is markedly different. Running time growth rates are now all at least quadratic (somewhat worse for Greedy and CW). More importantly, the percentage excesses are no longer bounded by small constants independent of N, but grow with N for all the algorithms, starting at figures of 100% or more for $N = 100$. For NN and Greedy, the growth rate appears to be proportional to $\log N$ (the theoretically correct growth rate for NN). For Clarke–Wright, the best of these three algorithms on Euclidean instances, the growth rate is substantially worse, more like $\sqrt{N}$ (presumably because of the failure of the triangle inequality). Christofides, had it been implemented, would likely have been even worse; recall that the expected length of an optimal tour is bounded, independent of N. On the other hand, each time Christofides has to make a shortcut in the final stage of its operation, the edge added in making the shortcut will be more or less random; whatever conditioning there is might be expected to make it *longer* than average. Given this, its expected length would be roughly 0.5 (or perhaps 0.25, if one is picking the best of two possible shortcuts). The number of shortcuts that need to be made is typically

proportional to N, so this suggests the percentage excess for Christofides would itself grow linearly with N.

3 2-OPT, 3-OPT, AND THEIR VARIANTS

In this section, we consider local improvement algorithms for the TSP based on simple tour modifications, *exchange heuristics* in the terminology of Chapter 1. Such an algorithm is specified in terms of a class of operations (exchanges or moves) that can be used to convert one tour into another. Given a feasible tour, the algorithm then repeatedly performs operations from the given class, so long as each reduces the length of the current tour, until a tour is reached for which no operation yields an improvement, *a locally optimal* tour. Alternatively, we can view this as a *neighborhood search* process, where each tour has an associated neighborhood of *adjacent* tours, i.e., those that can be reached in a single move, and one continually moves to a better neighbor until no better neighbors exist.

Among simple local search algorithms, the most famous are 2-Opt and 3-Opt. The 2-Opt algorithm was first proposed by Croes [1958], although the basic move had already been suggested by Flood [1956]. This move deletes two edges, thus breaking the tour into two paths, then reconnects those paths in the other possible way (Figure 8.1). Note that this picture is a schematic; if distances were as shown in the figure, the particular *2-change* depicted here would be counterproductive and so would not be performed. In 3-Opt [Bock, 1958b; Lin, 1965] the exchange replaces up to three edges of the current tour (Figure 8.2).

In Section 3.1 we shall describe what is known theoretically about these algorithms in the worst and average cases. Section 3.2 then presents experimental results from Johnson et al. [1997b] showing that the algorithms perform much better in practice than the theoretical bounds might indicate (at least for our geometric instances). Section 3.3 sketches some of the key implementation details that make these results possible. Section 3.4 then considers the question of how

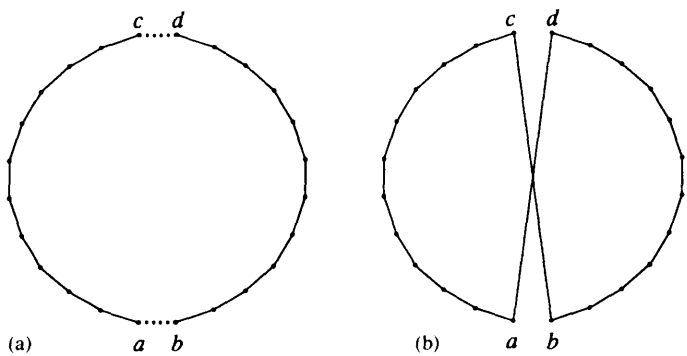

Figure 8.1 A 2-Opt move: (a) original tour, (b) resulting tour

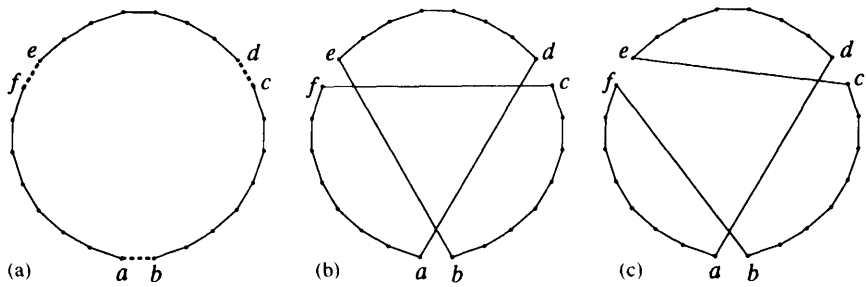

Figure 8.2 Two possible 3-Opt moves: (a) original tour, (b, c) resulting tours

local optimization algorithms like 2-Opt and 3-Opt might be sped up by the use of parallelism. Finally, Section 3.5 briefly surveys other 'simple' local optimization algorithms that have been considered in the literature.

3.1 Theoretical bounds on local search algorithms

Worst-case results

The first question to address concerns the quality of tours obtainable via an arbitrary local optimization algorithm for the TSP. For arbitrary instances such algorithms are constrained by Theorems A and B of Section 2.1, but something stronger can be said: if P $\neq$ NP no local search algorithm that takes polynomial time per move can guarantee $A(I)/\mathrm{OPT}(I) \leqslant C$ for any constant C, even if an exponential number of moves is allowed [Papadimitriou & Steiglitz, 1977]. In the unlikely event that P = NP one can still say that no local search algorithm with polynomial-size neighborhoods that do not depend on the values of the intercity distances can be guaranteed to find *optimal* solutions in polynomial time [Weiner, Savage & Bagchi, 1976]. This restriction on algorithms is stronger than simply requiring polynomial time per move, but it still allows 2-Opt, 3-Opt, and any k-Opt algorithm for fixed k, where k-Opt is the natural generalization of 2-Opt and 3-Opt to allow exchanges of as many as k edges. For the specific cases of 2-Opt, 3-Opt, and k-Opt with $k < 3N/8$, the situation is much worse. Papadimitriou & Steiglitz [1978] have shown that there exist instances which have a single optimal tour but exponentially many locally optimal tours, each of them longer than optimal by an exponential factor.

The Papadimitriou – Steiglitz results do not apply when the triangle inequality holds. Even if it does hold, however, 2-Opt and 3-Opt can fare poorly, as indicated by the following results of Chandra, Karloff & Tovey [1997]. Assuming an adversary is allowed to choose the starting tour, the best performance guarantee possible for 2-Opt is a ratio of at least $(1/4)\sqrt{N}$, and for 3-Opt it is at least $(1/4)N^{1/6}$. More generally, the best performance guarantee for k-Opt, assuming the triangle inequality, is at least $(1/4)N^{1/2k}$. On the 'positive' side, none

of these algorithms can yield a ratio *worse* than $4\sqrt{N}$. The situation is somewhat better if we restrict attention to instances where cities are points in $\mathbb{R}^d$ for some fixed d and distances are computed according to some norm for that space. In this case, no k-Opt algorithm has worst-case performance ratio worse than $O(\log N)$, although the ratio can grow as quickly as $\Theta(\log N/\log\log N)$ even for $\mathbb{R}^2$ under the Euclidean norm.

All these lower bound results hold only if we let an adversary choose our starting tours for us (in which case, the adversary simply picks a bad solution that is already locally optimal). In practice, we can obtain significantly better worst-case behavior simply by using a good heuristic to generate our starting tours. For example, if we use Christofides to generate our starting tours, then 2-Opt will never be worse than 3/2 times optimal, assuming the triangle inequality, much better than the above bounds would allow even for two-dimensional Euclidean instances. It is not clear, however, that either 2-Opt or 3-Opt can guarantee any consistent improvement over its starting tour. For example, the worst-case instances of Cornuéjols & Nemhauser [1978] for Christofides are such that the resulting starting tours are 2-optimal and the worst-case examples of Rosenkrantz, Stearns & Lewis [1977] for nearest insertion are such that the resulting starting tours are k-optimal for all $k \leqslant N/4$.

Another important question about local optimization algorithms concerns how long a sequence of moves they can make before reaching a locally optimal solution. For 2-Opt and 3-Opt this number can be quite large (even assuming the triangle inequality holds). Lueker [1976] has shown that there exist instances and starting tours such that 2-Opt will make $\Theta(2^{N/2})$ moves before halting. A similar result has been proved for 3-Opt by Chandra, Karloff & Tovey [1997] and extended to k-Opt for any fixed k. These results again assume we allow an adversary to pick the starting tour as well as the instance. However, for sufficiently large k, even finding a good starting tour may not help much. Krentel [1989] has shown that for k sufficiently large, the problem of finding a locally optimal solution under the k-Opt neighborhood structure is 'PLS-complete' as defined by Johnson, Papadimitriou & Yannakakis [1988] (see Chapter 2). This means that the overall time to find a locally optimal tour by *any* method cannot be polynomially bounded unless all PLS problems can be solved in polynomial time. A value of $k = 8$ appears to suffice for Krentel's results, and this might conceivably be reduced to $k = 6$, but further reductions would require substantially new ideas [Krentel, 1994], so the question of PLS completeness for 2-Opt and 3-Opt remains open.

Another *caveat* about some of the above results, in particular those of Lueker and Chandra et al., is that they also assume that the adversary is able to choose the improving move to be made when more than one exists. In the terminology of Chapter 2, the adversary gets to choose the 'pivoting rule'. Even if the algorithm can make its own choices, however, making the best choices can be hard. Fischer [1995] has shown that, given an instance of the TSP and a tour, the problem of finding the closest locally optimal tour is NP-hard, even if that local optimum is only a polynomial number of moves away.

At present, we do not know any nontrivial upper bounds on the numbers of moves that may be needed to reach local optimality, but a related result appears to be interesting. Suppose we restrict our attention to two-dimensional instances, and consider only 2-Opt moves that remove *crossings*, i.e., those that delete from the tour two edges that intersect at a common internal point. By the triangle inequality, such moves cannot cause the length of the tour to increase (under the Euclidean norm, the length must decrease), but they may introduce new crossings. (Indeed, there can be improving moves that increase the total number of crossings.) Nevertheless, as shown by Van Leeuwen & Schoone [1980], at most N^3 uncrossing moves suffice to remove all crossings. Thus the restricted version of 2-Opt in which only uncrossing moves are allowed will perform at most $O(N^3)$ moves for any two-dimensional instance. (Examples exist where $\Theta(N^2)$ such moves are required).

To complete our discussion of running times, we need to consider the time per move as well as the number of moves. This includes the time needed to *find* an improving move (or verify that none exists), together with the time needed to *perform* the move. In the worst-case, 2-opt and 3-opt require $\Omega(N^2)$ and $\Omega(N^3)$ time respectively to verify local optimality, assuming all possible moves must be considered. The actual cost may be higher, depending on the data structures used for representing the tour. The time to perform a move also depends on the data structures used, and there are trade-offs involved [Fredman et al., 1995]. Using an array data structure, one can reduce the cost of evaluating a move to $\Theta(1)$ while spending $\Theta(N)$ time to actually make a move. Using the splay tree data structure of Sleator & Tarjan [1985], one can make both operations cost $\Theta(\log N)$ in an amortized sense. Assuming a reasonable computational model, the worst-case amortized cost per operation (over both types of operation) must be $\Omega(\log N/\log\log N)$ [Fredman et al., 1995].

Bounds on expected behavior

Many of the questions raised in the previous section have also been addressed from an average-case point of view, in particular for random Euclidean instances and their generalizations to higher dimensions and other metrics. At present, we do not know how to prove tight bounds on the expected performance ratios for sophisticated TSP heuristics like 2-Opt and 3-Opt in these models. In the case of 2-Opt, however, a first step may have been provided by Chandra, Karloff & Tovey [1997], who have shown that for any fixed dimension d, the expected ratio of the length of the *worst* 2-optimal tour to the optimal tour length is bounded by a constant. This means that, on average, 2-Opt can be no worse than some constant times optimal, a significant improvement over the worst case.

Analogous improvements have been obtained with respect to running time, assuming we compare the unrestricted worst case to the above two-dimensional average-case model. Whereas in the worst case an exponential number of moves may be required before local optimality is reached, the expected number of moves (even starting from the worst possible tour) is polynomially bounded in the

two-dimensional model. Under the Euclidean metric the bound is $O(N^{10} \log N)$ and under the rectilinear metric it is $O(N^6 \log N)$, as shown by Chandra, Karloff & Tovey [1997], improving on earlier results of Kern [1989]. Given a method for generating starting tours with expected length $cN^{1/2}$ (as we would presumably get from most of our tour construction heuristics), these bounds can be reduced by a factor of $N^{1/2}$. Even so, as we shall see in the next section, the bounds substantially overestimate the true expected value of the number of moves.

3.2 Experimental results for 2-Opt and 3-Opt

In this section we shall summarize experimental results obtained for our tested instances using the *neighbor-list* implementations of 2-Opt and 3-Opt by Johnson et al. [1997b]. Key details of the implementation will be sketched in the next section. The full picture and additional results can be found in the reference itself. For now it should be noted that these implementations make certain trade-offs, giving up the guarantee of true 2- or 3-optimality in favor of greatly reduced running time. Neither tour quality nor the numbers of moves made appears to be substantially affected by these changes, however, at least for our random Euclidean instances.

Table 8.3 covers tour quality for the random Euclidean and random distance matrix instances in our test bed, with the best of the tour construction heuristics for each class included as a point of comparison. For the 2-Opt and 3-Opt results, starting tours were generated using a randomized version of the greedy heuristic. Instead of choosing the shortest legal edge to add to the partial tour, we choose among the shortest two, with the shortest having probability 2/3 of being chosen. On average, this randomized version has essentially the same tour quality as the true greedy heuristic. The results presented are averages over multiple runs for

Table 8.3 Tour quality for 2-Opt and 3-Opt, plus selected tour generation heuristics: average percent excess over the Held Karp lower bound

$N =$	10^2	$10^{2.5}$	10^3	$10^{3.5}$	10^4	$10^{4.5}$	10^5	$10^{5.5}$	10^6
				Random Euclidean instances					
GR	19.5	18.8	17.0	16.8	16.6	14.7	14.9	14.5	14.2
CW	9.2	10.7	11.3	11.8	11.9	12.0	12.1	12.1	12.2
CHR	9.5	9.9	9.7	9.8	9.9	9.8	9.9		
2-Opt	4.5	4.8	4.9	4.9	5.0	4.8	4.9	4.8	4.9
3-Opt	2.5	2.5	3.1	3.0	3.0	2.9	3.0	2.9	3.0
				Random distance matrices					
GR	100	160	170	200	250	280			
2-Opt	34	51	70	87	125	150			
3-Opt	10	20	33	46	63	80			

each instance, although not enough to ensure narrow confidence intervals on the averages, especially for the small instances.

Note that for Euclidean instances, 2-Opt averages about 5% better than Christofides, the best of our tour construction heuristics, even though it is started from tours that average 5–10% worse. And 3-Opt is another 2% points better still; moreover, at 3% excess over the Held–Karp lower bound, it may already be as good as one needs in practice. For random distance matrix instances, the story is considerably less impressive. Although both 2-Opt and 3-Opt offer substantial improvements over the best of our tour generation heuristics on these instances, their tour quality is still decaying at a relatively steady rate as N increases, the excess over the Held–Karp lower bound seeming to grow roughly as $\log N$, just as we observed for Greedy and NN. Results for our test bed of `TSPLIB` instances are once again similar to those for the random Euclidean instances, although somewhat worse. They average about 6.9% above Held–Karp for 2-Opt and 4.6% for 3-Opt, although these figures can be improved a bit if we are willing to spend more time and space in looking for moves.

There is a gross disparity between the results in Table 8.3 (even the results for random distance matrices) and the worst-case bounds of Section 3.1. Given that for those bounds we allowed an adversary to choose the starting tour, it is natural to ask how much our experimental results depend on our choice of starting tour. In Table 8.4 we compare results for different starting heuristics (including the use of random starting tours) on the 1000-city instances in our two test beds. The same general picture emerges for other choices of N, although in the case of random distance matrices, all the numbers are bigger for larger N.

For random Euclidean instances, the most striking observation is that the worst percentage excesses do not correspond to the worst starting heuristic but to the best. Although CW produces the best starting tours by a significant margin, the results of applying 2-Opt and 3-Opt to such tours are worse than those for any of the other starting heuristics, including random starts. This is not an isolated property of CW. As first observed by Bentley [1992], Greedy starts provide better final results for 2-Opt and 3-Opt than any other known starting heuristic, including those that provide better tours per se, such as Farthest Insertion and

Table 8.4 Tour quality for 2-Opt and 3-Opt using different starting heuristics: average percent, excess over the Held–Karp lower bound, $N = 1000$

	Random Euclidean instances			Random distance matrices		
	Start	2-Opt	3-Opt	Start	2-Opt	3-Opt
Random	2 150.0	7.9	3.8	24 500	290	55
NN	25.9	6.6	3.6	240	96	39
Greedy	17.6	4.9	3.1	170	70	33
CW	11.4	8.5	5.0	980	380	56

fast approximations to Christofides. It appears that if local optimization is to make substantial improvements, its starting tour must contain a number of exploitable defects, and if a tour is too good it may not have enough of these. Note, however, that among the remaining three heuristics covered in Table 8.4, better starting tours do yield better final tours, albeit with the range of differences substantially compressed, especially in the case of the comparison between random starts and the other two heuristics. Ignoring CW, the same holds for instances of our random distance matrices; better starting tours yield better results (although here the range of initial differences may not be compressed quite as much by local optimization). CW again is an outlier, yielding final results that are worse than for random starts, even though its initial tours are far better than random tours.

Let us now turn to issues related to running time, beginning with the question of the number of moves made. How well do the actual numbers stack up against the worst-case and average-case bounds of the previous two sections? Table 8.5 gives the average numbers of improving moves made under Greedy and random starts for our two test beds of random instances. To ease comparisons for different numbers of cities, we have normalized by dividing each average count by N. For random Euclidean instances and Greedy starts, the average numbers of moves made by 2-Opt and 3-Opt are $O(N)$ and may well even be $o(N)$, with 3-Opt making about 50% more moves than 2-Opt. For random starts, on the other hand, the average number of improving moves seems to grow superlinearly with N; $\Theta(N \log N)$ is a reasonable guess for both algorithms. The numbers also start out much larger, with 2-Opt making 10 times as many moves as it did from Greedy starts when $N = 100$. And 2-Opt now makes more moves than 3-Opt

Table 8.5 Normalized numbers of moves made by 2-Opt and 3-Opt: (average number of moves made)/N

$N =$	10^2	$10^{2.5}$	10^3	$10^{3.5}$	10^4	$10^{4.5}$	10^5	$10^{5.5}$	10^6
				Random Euclidean instances					
Greedy starts									
2-Opt	0.189	0.185	0.182	0.173	0.169	0.154	0.152	0.150	0.147
3-Opt	0.273	0.256	0.248	0.238	0.233	0.211	0.213	0.209	0.207
Random starts									
2-Opt	1.87	2.36	2.67	2.87	3.01	3.14	3.26	3.37	3.48
3-Opt	1.19	1.36	1.48	1.64	1.77	1.90	2.04	2.18	2.33
				Random distance matrices					
Greedy starts									
2-Opt	0.131	0.086	0.042	0.021	0.010	0.004			
3-Opt	0.270	0.216	0.144	0.082	0.057	0.030	–		
Random starts									
2-Opt	1.44	1.56	1.56	1.55	1.55	1.55	–		
3-Opt	1.19	1.30	1.28	1.21	1.12	1.07	–		

Table 8.6 Running times for 2-Opt and 3-Opt, plus selected tour construction heuristics: seconds on a 150 MHz SGI Challenge

$N =$	10^2	$10^{2.5}$	10^3	$10^{3.5}$	10^4	$10^{4.5}$	10^5	$10^{5.5}$	10^6
				Random Euclidean instances					
CHR	0.03	0.12	0.53	3.57	41.9	801.9	23 009		
CW	0.00	0.03	0.11	0.35	1.4	6.5	31	173	670
GR	0.00	0.02	0.08	0.29	1.1	5.5	23	90	380
Preprocessing	0.02	0.07	0.27	0.94	2.8	10.3	41	168	673
Starting tour	0.01	0.01	0.04	0.15	0.6	2.7	12	50	181
2-Opt	0.00	0.01	0.03	0.09	0.4	1.5	6	23	87
3-Opt	0.01	0.03	0.09	0.32	1.2	4.5	16	61	230
Total 2-Opt	0.03	0.09	0.34	1.17	3.8	14.5	59	240	940
Total 3-Opt	0.04	0.11	0.41	1.40	4.7	17.5	69	280	1 080
				Random distance matrices					
GR	0.02	0.12	0.98	9.3	107	1 400	–		–
Preprocessing	0.01	0.11	0.94	9.0	104	1 370			
Starting tour	0.00	0.01	0.05	0.3	3	27	–		–
2-Opt	0.00	0.01	0.03	0.1	0	1	–	–	–
3-Opt	0.01	0.04	0.16	0.6	3	15	–	–	–
Total 2-Opt	0.02	0.13	1.02	9.4	108	1 400	–		
Total 3-Opt	0.02	0.16	1.14	9.8	110	1 410		–	–

(presumably because 3-Opt is able to obtain bigger improvements per move when one starts from a horrible tour).

With random distance matrices, the story is somewhat different. Now the average numbers of moves made from Greedy starts is definitely sublinear within the range of N covered by the table. Indeed, it doesn't even seem to grow as fast as $N^{1/2}$. The inability of 2-Opt and 3-Opt to find many improving moves may help explain their inability to overcome the poor quality of their starting tours on these instances. For random starts, where a lot more improvement is possible, many more improving moves are made, although the number appears still to be $O(N)$ for 2-Opt and sublinear for 3-Opt.

Finally, let us look at running times. Table 8.6 restricts its attention to results for Greedy starts. We include the times for the best of the starting heuristics as a point of comparison, and we also show the breakdown of the running time for 2-Opt and 3-Opt into preprocessing (the construction of neighbor lists, as will be explained in the next section), starting tour construction, and the time specifically devoted to 2- or 3-optimization. Note that the overall running times for 2-Opt and 3-Opt differ by only a small amount for random Euclidean instances and by even less for random distance matrix instances. This is because the overall time is

dominated by that for preprocessing, which for random Euclidean instances contributes more than 60% of the overall time even for 3-Opt and $N = 10^6$. For random distance matrices, preprocessing accounts for over 95% of the running time for large instance. Furthermore, Greedy starting tour generation, even though it is faster than performing standalone Greedy because it can exploit the existence of the neighbor lists, still takes substantially longer than 2-optimization and almost as long as 3-optimization (longer on random distance matrices). Thus, although the process of 3-optimization takes roughly three times as long as 2-optimization for the geometric instances, this effect is swamped by the other factors.

Let us now consider how these times relate to the theoretical bounds of Section 3.1. For random Euclidean instances the times are much better than our worst-case estimates would have suggested, at least with respect to the range of N covered by the table. As noted in Section 3.1, the only certain bound we can place on the cost of finding an improving move is $\Theta(N^2)$ for 2-Opt and $\Theta(N^3)$ for 3-Opt. Thus, even if we take into account the actual number of moves, which is roughly $\Theta(N)$, the overall time could be something like $\Theta(N^3)$ for 2-Opt and $\Theta(N^4)$ for 3-Opt. The times we actually observe for the 2- and 3-optimization phases are definitely subquadratic, and if not $O(N \log N)$ then at least no worse than $O(N^{1.2})$. As mentioned above, the time for 3-optimization is only three times greater than that for 2-optimization, not the factor of N that worst-case theory would imply. Overall time is also decidedly subquadratic. Indeed, as N increases 3-Opt's overall time drops to less than twice that for CW!

For random distance matrices, the time is slightly worse than quadratic for large N. This is most likely due to memory hierarchy effects, given that it also holds true for preprocessing alone, which is theoretically an $\Theta(N^2)$ algorithm. The added time for 2-opting or 3-opting is itself subquadratic and indeed essentially negligible once N gets large enough. This is consistent with our observation that the number of moves made for these instances under greedy starts is substantially sublinear.

In the next section we describe the main implementation ideas that make these high speeds possible. We exploit the key fact that, in local optimization as opposed to approaches such as tabu search and simulated annealing, we can ignore all nonimproving moves.

3.3 How to make 2-Opt and 3-Opt run quickly

In order to obtain running times for 2-Opt and 3-Opt that are subquadratic in practice, we must first figure out a way to avoid looking at all possible moves in our search for one that shortens the tour. Fortunately, there is a simple and effective way to do this.

To illustrate this, let us consider the case of 2-Opt, and let us assume a fixed orientation of the tour, with each tour edge having a unique representation as a pair (x, y), where x is the immediate predecessor of y in tour order. Then each possible 2-Opt move can be viewed as corresponding to a 4-tuple of cities $\langle t_1, t_2, t_3, t_4 \rangle$, where (t_1, t_2) and (t_4, t_3) are the oriented tour edges deleted and

$\{t_2, t_3\}$ and $\{t_1, t_4\}$ are the edges that replace them. Under this convention, each move actually corresponds to two distinct 4-tuples. For instance, the 2-Opt move illustrated in Figure 8.1 corresponds, under the counterclockwise tour orientation, to the 4-tuples $\langle a, b, c, d \rangle$ and $\langle d, c, b, a \rangle$. As first observed by Steiglitz & Weiner [1968], we can exploit this double representation as follows. Observe that for an improving move it must be the case that either (a) $d(t_1, t_2) > d(t_2, t_3)$ or (b) $d(t_3, t_4) > d(t_4, t_1)$ or both. But this means that (a) must hold for at least one of the two representations of the move. Consequently, we cannot miss an improving move if we restrict attention to 4-tuples satisfying (a). This means that in our search, having fixed t_1 and t_2, the possibilities for t_3 can be limited to those cities that are closer to t_2 than is t_1. (Note that, given the choice of t_3, there is only one possible choice of t_4 such that the move will yield a tour rather than two disjoint cycles.) As the algorithm progresses, the set of candidates for t_3 is more and more likely to be a small set. Analogously for 3-Opt, as observed by Lin & Kernighan [1973], we may restrict attention to 6-tuples $\langle t_1, t_2, t_3, t_4, t_5, t_6 \rangle$ in which $d(t_1, t_2) > d(t_2, t_3)$ and $d(t_1, t_2) + d(t_3, t_4) > d(t_2, t_3) + d(t_4, t_5)$. This limits the search for t_3 as in 2-Opt but also imposes a substantial constraint on the possibilities for t_5. (Degenerate 6-tuples that repeat some cities and hence correspond to 2-Opt moves are allowed.)

In order to exploit these observations, we need a data structure that allows us quickly to identify the allowable candidates for t_3 (t_5). Steiglitz & Weiner proposed storing for each city c a list of the remaining cities in order of increasing distance from c. To consider candidates for t_3, we then need only start at the beginning of t_2's list and proceed down it until a city x with $d(t_2, x) \geqslant d(t_1, t_2)$ is encountered. (The search for t_5 candidates in 3-Opt works similarly, using t_4's list.) The drawback to this approach is that it would take $\Theta(N^2 \log N)$ time to set up the lists and $\Theta(N^2)$ space to store them. Our *neighbor-list* implementations make the reasonable compromise of storing sorted lists containing only the k nearest neighbors for each city for a suitable fixed k, typically $k = 20$. We then halt our searches for t_3 (t_5) if we reach a city that is too far away or we get to the end of a list. This data structure takes only $\Theta(Nk)$ space and can be constructed in time $\Theta(N^2 \log k)$, a bound that can be improved to roughly $\Theta(N \log N + Nk)$ for geometric instances using k-d trees. As we saw in the last section, even with these speedups, the construction time for neighbor lists dominates the overall running time for the neighbor-list implementations of 2-Opt and 3-Opt. For future reference, note that the neighbor lists are static structures and could be reused by subsequent runs of the algorithms on the same instance, thus amortizing the cost of their construction. Thus the time needed to perform m runs of 3-Opt from different randomized Greedy starts would be substantially less than m times the time to perform a single run.

The loss of tour quality involved in using truncated neighbor lists is not a major issue in practice, although it is possible that the final solution may not be truly locally optimal. For random Euclidean instances, going from $k = 20$ to $k = 80$ only tends to improve the final tour by 0.1% or 0.2% on average, at significant running time cost. For some `TSPLIB` instances the difference can be greater, but

even here the point of diminishing returns sets in around $k = 40$, especially if one uses a variant in which the 40 neighbors of c are chosen so as to include the 10 closest cities in each of the four quadrants of the plane surrounding c. For random distance matrices and random starts, increasing k may actually make the final tour get worse; the optimal choice seems to be about $k = 15$.

For fixed k, the above ideas can reduce the time for finding an improving 2-Opt (3-Opt) move from $\Theta(N^2)$ ($\Theta(N^3)$) to $O(N)$, but given that there are roughly $\Theta(N)$ moves made for random Euclidean instances, a further idea is needed. This is provided by the concept of the *don't-look* bit, introduced by Bentley [1992], another way of trading a slight increase in the possibility of a nonlocally optimal final tour for a major reduction in running time, again based on the idea that we need only consider improving moves. Here the observation is that if, for a given choice of t_1, we previously failed to find an improving move, and if t_1's tour neighbors have not changed since that time, it is unlikely we will find an improving move if we look again starting at t_1. We exploit this observation by having special flags for each of the cities, which we call *don't-look* bits. Initially, they are all turned off. The bit for city c is turned on whenever a search for an improving move with $t_1 = c$ fails and is turned off whenever a move is performed in which c is an endpoint of one of the deleted edges. In considering candidates for t_1, we ignore all cities whose don't-look bits are on. This is done by maintaining a first-in, first-out queue of cities whose bits are off.

As a result of all this, the total number of t_1's considered throughout the course of the algorithm on random Euclidean instances is reduced to roughly $\Theta(N)$, although the constant is fairly large. The total search time is also $\Theta(N)$. The running time bottleneck thus becomes the time to perform the moves that are found, which depends intimately on the method used for representing the current tour. For the results reported in the previous section, we use the *two-level tree* data structure of Fredman et al. [1995], which theoretically yields a cost of $O(N^{1/2})$ per move. Although this is asymptotically worse than the splay tree representation mentioned in Section 3.1, it is faster for the range of N we consider; see Fredman et al. [1995] for more details. An alternative *segment tree* data structure due to Applegate, Chvátal & Cook [1990] is also competitive for $N \leqslant 100\,000$; see Fredman et al. [1995] and Reinelt [1992, 1994].

Beyond these algorithmic ideas, our implementation includes several tricks that merely yield constant factor improvements, such as caching of distances; see Johnson et al. [1997b] for fuller details. That paper also examines in more detail the effects of various choices one must make in an implementation, such as the question of which improving move to make when more than one exists. Choosing the best move seems a good policy, but the best move can be expensive to find, so there are trade-offs involved. The neighbor-list implementations typically choose the first improving move found, but the search is biased so as to find better moves first. The paper also includes an extensive evaluation of the alternative *on-the-fly* implementation of Bentley [1992] for geometric instances, which dispenses with the neighbor lists entirely and uses a k–d tree directly to find candidates for t_3 and t_5. This enables us to consider all the legal candidates in order, without trunca-

tion. It also reduces preprocessing time and the storage space needed by the algorithm, which for the neighbor-list implementation can be substantial when N is large. For $N = 10^6$ the reduction is roughly from 275 to 72 megabytes. The drawback is that each candidate takes longer to identify, so the local optimization phase takes more time. For 2-Opt the overall result is slightly better running times than for the neighbor-list implementation and slightly better tours; for 3-Opt it is slightly slower running times and slightly worse tours (but the reduction in tour quality may be due to implementation details other than the data structure chosen). Another approach, advocated by Reinelt [1991, 1994] and Jünger, Reinelt & Rinaldi [1995] is to stick to neighbor lists, but to use Delaunay triangulations instead of k–d trees when constructing them. As implemented by Reinelt and coworkers, this approach yields comparable running times for 2-Opt but, because of other choices made in the implementation, worse tours. For 3-Opt both running time and tour quality appear to be worse than with the neighbor-list implementation.

3.4 Parallel 2-Opt and 3-Opt

One oft-mentioned method for speeding up TSP algorithms is the use of parallelism. Although the times reported above for 2-Opt and 3-Opt on a single processor seem reasonable for most applications, one can conceive of situations where much greater speed would be needed because of real-time constraints, or where the number of cities may grow so large that many hours would be required if one were to use only a single processor. More significant, perhaps, is the memory requirement of roughly 275 bytes per city for our neighbor-list implementations. This means that the largest instance a 30 megabyte workstation can handle is only on the order of 100 000 cities, so if one wants to use such machines to solve larger instances, one must resort to parallelism of some sort, for example by distributing the problem across a network of workstations. Various schemes have been proposed for this, all exploiting a coarse-grained parallelism where each processor has associated with it enough memory to run TSP algorithms on instances up to some reasonable size. We shall discuss three basic approaches that can be applied not only to 2-Opt and 3-Opt but also to many of the alternative approaches to be covered in later sections.

Geometric partitioning

A partitioning scheme proposed by Karp [1977] can be used for two-dimensional geometric instances. This scheme partitions the cities in a manner similar to the construction of the k–d tree data structure. It is based on a recursive subdivision of the overall region containing the cities into rectangles, with the set of cities corresponding to a given rectangle being comprised of all the cities in the rectangle's interior together with some of the cities on its boundary. Suppose we wish to construct a partition in which no set contains more than K cities. The recursive step of the partitioning scheme works as follows. Let C_R be the set of

cities assigned to rectangle R, and suppose $|C_R| > K$. One subdivides R into two subrectangles as follows. Suppose without loss of generality that the x-coordinates of the cities in C_R have a larger range than the y-coordinates. Find a city c in C_R whose x-coordinate has the median value for cities in C_R. Divide R into two subrectangles R_1 and R_2 by drawing a vertical line through c, letting R_1 be the rectangle to the left of the line. Each city in C_R to the left of the line is assigned to R_1, each city to the right is assigned to R_2, and cities on the line are divided as equally as possible between the two rectangles, except that city c itself is assigned to *both* R_1 and R_2.

Once the cities have been partitioned in this way into subrectangles, none of which has more than K cities assigned to it, one can send each subrectangle to a processor, and have that processor run 2-Opt or 3-Opt (or any other TSP algorithm) on the corresponding set of cities. The union of the tours thus found will be a connected Eulerian subgraph, so it can be converted to a tour for the entire instance by using shortcuts as in the Christofides algorithm.

Although we have not implemented this scheme as a parallel algorithm, its value as a way to reduce memory requirements can be exploited even on a single processor: we simply solve the subproblems one after another, reusing the space required for each one. To the extent that the running time of our basic TSP algorithm is superlinear, this sequential scheme can even provide speedups over performing the algorithm on the complete instance. For $N = 10^6$ and 3-Opt, the savings in runtime obtained by partitioning into 1024 subproblems was over a factor of 2. Unfortunately, there was also a corresponding loss in tour quality, with the original 3.0% average excess over the Held–Karp bound increasing to 5.5%. Partitioning into just 16 subproblems, which just might be enough to allow us to handle 10^6 cities on our hypothetical 30 megabyte workstation, greatly reduces the tour-quality penalty; now the increase is only to 3.2%. This general scheme can also be applied to instances with fewer cities, although for such instances the deterioration in tour quality sets in earlier, i.e., for smaller numbers of subproblems. A rule of thumb seems to be that one can expect significant deterioration as soon as the number of cites in a subproblem drops below 1000.

Reinelt [1994] has proposed two variants on the above scheme, both based on generating a sparse graph whose vertices are the cities and whose connected components can serve as the subproblems to be solved by local optimization. In the first scheme, the sparse graph simply contains for each city the edges to the city's nearest two (or three) neighbors. This approach tends to yield some very small connected components, and Rohe [1995] has suggested a process whereby one successively merges nearby small components until each merged component contains between $M/2$ and M cities, where M is the maximum size of subproblem one wishes to consider. In Reinelt's second scheme, the sparse graph is generated using edges from the Delaunay triangulation of the set of cities. The edges of the triangulation are considered in order of increasing length, and an edge is added to the graph unless it would cause the creation of a subproblem with more than M cities or would reduce the total number of subproblems below some prescribed threshold.

Given such a partition into connected components, a tour can be constructed as follows. The global ordering of the components within the tour is determined first, along with the identities of the edges that will link each component to its successor. Each subproblem then has two distinguished cities, i.e., those that are involved in intercomponent links. Before applying local optimization to a subproblem, we add an unbreakable edge between these two. This insures that the resulting tour for the subproblem will contain a Hamiltonian path linking the entry and exit cities. Once all subproblems have been solved in this way, it will be straightforward to assemble the overall tour. Based on experiments with 24 selected instances from TSPLIB, Reinelt concludes that neither of these two approaches is preferable to a rectangle-based scheme. But note that his first variant is more general than the others; it does not explicitly rely on the geometric nature of the instance (although having the triangle inequality hold would probably be helpful).

Tour-based partitioning

One reason for the loss in tour quality engendered by geometric partitioning schemes is the fact that each subproblem is handled in isolation, with only limited opportunities for intelligent patching together of the subtours. Another drawback of the scheme is that, although it can be generalized to higher dimensions, it is restricted to geometric instances. The alternative *tour-based* partitioning scheme avoids these pitfalls by basing its partition of the cities on the structure of the current tour and by performing more than one partitioning phase.

One begins by using a simple heuristic to generate an initial tour and then breaks that tour up into k segments of length N/k, where k is greater than or equal to the number of processors available. Each segment is then handed to a processor, which converts the segment into a tour by adding an edge between its endpoints and attempts to improve the tour by local optimization (2-Opt, 3-Opt, etc.), subject to the constraint that the added edge cannot be deleted. The resulting tour can thus be turned back into a segment with the same endpoints, and the improved segments can then be put back together into a new overall tour. We can then construct a revised partition where each new segment takes half its cities from each of two adjacent old segments and repeat the parallel local optimization phase. Additional phases can be performed until a time limit is exceeded or no significant further improvement is obtained. This scheme has been tried by Applegate & Cook [1994] on a 10 907 064-city instance using a network of 10 workstations and a k of about 1000. They assigned 100 subproblems to each processor, and each processor ran its assigned subproblems in sequence. A neighbor-list implementation of the Lin–Kernighan algorithm of Section 4.2 was used as the local-optimization engine. The resulting tour was within 4.3% of the Held–Karp bound for the instance.

Note that there still appears to be a tour-quality penalty for partitioning in this way. Despite the shifting of the segment boundaries, it remains difficult to move a city very far away from its original position in the tour. An alternative

tour-based partitioning scheme suggested by Allwright & Carpenter [1989] helps alleviate this drawback. As in the above scheme, they partition the current tour into segments, but now they create $2k$ segments and hand out *pairs* of segments to processors. If the segments are $S_1, S_2, \ldots, S_{2k}$ in order around the tour, we pair S_1 with S_{2k}, S_2 with S_{2k-1}, and in general S_j with S_{2k-j+1}, $1 \leqslant j \leqslant k$. Note that the first and last pairs are degenerate, as the segments share an endpoint, so each can be viewed as a single segment and treated as in the previous scheme. A non-degenerate pair of segments is converted into a tour by adding two edges, each connecting one of the endpoints of the first segment to one of the endpoints of the second. The added edges are chosen so that each shortcuts a path made up of tour edges not in either segment. The processor is then free to perform any local optimization step on its subtour that does not break an added edge. After a phase of local optimization based on such a partition, an overall tour can then be reassembled from its parts, and we can repartition the tour and try again.

Allwright & Carpenter proposed this scheme in the context of 2-Opt, but did not attempt a serious test of the algorithm. Verhoeven, Aarts & Swinkels [1995] modified the approach and tested it on TSPLIB instances with as many as 11 849 cities, using a network of 512 INMOS T805 transputers. Their modification includes a clever repartitioning scheme that guarantees their algorithm will not halt until it has found a solution which is locally optimal with respect to the 2-Opt neighborhood for the complete problem. Consequently, they experience little if any degradation in tour quality as the number of processors increases. Unfortunately, this scheme requires $\Omega(N^2/p + pN)$ time, where p is the number of processors. This is $\Omega(N^{1.5})$ regardless of p and so can provide significant speed-ups only when compared to the naive $\Omega(N^2)$ implementation of 2-Opt, which we already know how to speed up even more while using only a single processor. Thus for the 11 849-city instance and 512 processors, the running time is almost a factor 200 greater than the time for neighbor-list 2-Opt on a single SGI Challenge processor, a factor that appears to be growing with instance size.

Some of this factor can be blamed on the fact that the transputers are slower processors, but most is due to the nature of the approach itself, which is driven by the desire to obtain local optimality with respect to the full 2-Opt neighborhood. Nor is this necessarily very desirable. The tours found by Verhoeven et al., although as good as the tours they obtained with their single-processor implementation of full 2-Opt, are still not as good as those obtained using neighbor-list 2-Opt. For the 11 849-city instance they are some 3% worse. This appears to be typical and is probably a consequence of the different orders in which moves are considered and made in the two algorithms. For an analogous parallel implementation of a 3-Opt variant, Verhoeven et al. [1992] report much better tour quality (results comparable to the neighbor-list implementation), although their running times are still far in excess of those for the single-processor neighbor-list implementation, so this does not appear to be a competitive approach. For a partition-based parallel scheme to succeed, it probably needs to be organized so it can use

the same speedup tricks our sequential implementations exploited, even if this means that local optimality cannot be guaranteed.

One might also wish to consider the following hybrid between the geometric and tour-based partitioning schemes proposed by Rohe [1995]. Given one's current tour, one generates subproblems consisting of cities closely related in a geometric sense; for example, all cities in a given rectangle or the M nearest neighbors of a given city. For a given subproblem S, the current tour induces a collection of paths through the cities of S. We add additional edges to link up these paths in the same order they occur in the overall tour, yielding a tour for S, then perform local optimization on this tour, with the added edges being fixed and undeletable. This produces a new set of paths through S which can replace the old set in the tour for the entire instance. Assuming that local optimization improved the tour for S, the new overall tour will be shorter as well. When multiple subproblems of this sort are handled in parallel, there is some danger that the overall result will be a collection of disjoint cycles rather than a tour, even if no two subproblems share a city. In Rohe's scheme, one checks for such subtours and patches them together using additional 2-Opt moves. Rohe applied this scheme in a multiphase approach to the same 10 907 064-city instance for which we reported Applegate & Cook's results in the previous section. Using the same Lin–Kernighan local search engine and two weeks on a network of four IBM 550 workstations, he reduced the excess over the Held–Karp bound from Applegate & Cook's 4.3% to 1.57%. This is roughly the tour quality that Lin–Kernighan might be expected to produce were it to be run on the whole (unpartitioned) instance. Rohe has also obtained a tour of this quality for an even bigger instance, one whose cities correspond to the locations in the sky of some 18 million stars.

Using parallelism in preprocessing and the search for improving moves

When memory is not a constraint, there are much simpler ways to obtain significant speedups over our fast sequential implementations of local optimization algorithms without paying a penalty in tour quality. First recall that typically over 2/3 of the time for the neighbor-list implementations of 2-Opt and 3-Opt is spent in building the neighbor lists themselves, which is readily parallelizable assuming each processor has enough memory to store the full instance.

Similarly, the search for improving moves typically dominates the remaining time and offers ample opportunities for parallelism. For example, when neighbor-list 3-Opt is applied to a random Euclidean instance, we typically evaluate 50 or more moves for every move actually made. There is no reason why these searches cannot be performed in parallel. Each processor may need access to the entire instance and all the neighbor lists, but assuming there is enough memory so that all this information can be replicated at each processor, significant reductions in running time should be possible.

We know of no serious attempt to exploit these two avenues of parallelism in the context of 2-Opt and 3-Opt, but search parallelization becomes even

more attractive for some of the algorithms we shall discuss in later sections, and an application of it to the Lin–Kernighan algorithm will be mentioned in Section 4.

3.5 Other simple local optimization algorithms

There are simpler local optimization algorithms than 2-Opt and 3-Opt. For example, one could restrict exchanges to those in which two adjacent cities in the current tour are interchanged, thus reducing the size of a neighborhood from the $N(N-2)/2$ of 2-Opt to N. However, such simpler neighborhoods are not powerful enough to yield results at all close to those for 2-Opt, and as observed above, versions of 2-Opt can be implemented to run far more quickly than its neighborhood size would suggest. Thus the real game is to devise local optimization algorithms that are a bit more complicated than 2-Opt (or 3-Opt) but produce better tours.

Between 2-Opt and 3-Opt

A simple extension is what Bentley [1992] calls *2.5-Opt*. In this algorithm we expand the 2-Opt neighborhood to include a simple form of 3-Opt move that can be found with little extra effort. In a 2.5-Opt move one relocates a single city from its current location to a position between two current tour neighbors elsewhere in the tour. In Figure 8.2, this corresponds to the situation where b and c are the same city. The search for such moves can be incorporated into our basic 2-Opt search as follows: for each t_3 candidate (i.e., each city that is closer to t_2 than is t_1) we evaluate the corresponding 2-Opt move *plus* the 2.5-Opt move that places t_2 between t_3 and t_4 and the 2.5-Opt move that places t_3 between t_1 and t_2. Bentley [1992] has implemented 2.5-Opt for geometric instances using the on-the-fly approach. For random Euclidean instances it yields a 0.5% improvement in average tour length over 2-Opt at the cost of a 30–40% increase in running time. For our test bed of `TSPLIB` instances, the improvement in tour length over 2-Opt was more like 1%. This was still 1% worse than obtained by our *neighbor-list* implementation of full 3-Opt, but it took only half the time.

A second algorithm that is intermediate between 2-Opt and 3-Opt is the *Or-Opt* algorithm originally proposed by Or [1976] and popularized by Golden & Stewart [1985]. Here we restrict attention to 3-Opt moves in which a segment consisting of three or fewer consecutive cities is excised from the tour and placed between two tour neighbors elsewhere in the tour. In Figure 8.2 this corresponds to the situation where cities b and c are 0, 1, or 2 cities apart in the tour, a natural generalization of the 2.5-Opt move. A version of Or-Opt could be implemented by slightly modifying an implementation of 2.5-Opt. Such an implementation should be a little faster than full 3-Opt, although a slight loss in tour quality is to be expected. To data, however, Or-opt has only been studied by researchers who assumed that 3-Opt requires $\Theta(N^3)$ time and so settled for Or-Opt implementations that take quadratic time. Such implementations are not competitive

with our *neighbor-list* implementation of full 3-Opt in either running time or tour quality.

k-Opt for $k > 3$

Both 2.5-Opt and Or-Opt only attempt to improve on 2-Opt. What about improving on 3-Opt? One obvious direction is to consider 4-Opt, where up to four tour edges can be changed in one move. This apparently yields little in the way of further improvement, however, as was observed by Lin [1965]. Lin's observation, together with the subsequent development of the highly successful variable-opt algorithm of Lin & Kernighan [1973], combined to quash any further research into the fixed-k-Opt algorithms with $k > 3$. The Lin–Kernighan algorithm itself would be hard to characterize as a *simple* local optimization algorithm, and we will postpone its discussion until Section 4.

One special case of the 4-Opt move is worth discussing, however, as it will come up again in Section 6. This is the *double-bridge* move. Such moves were first mentioned in the original Lin–Kernighan paper [1973], where they were used as an example of a simple move that, because of its nonsequential nature, would not normally be found by the Lin–Kernighan algorithm. A double-bridge move can be viewed as the combination of two illegal 2-Opt moves (the bridges), each of which by itself converts the tour into two disjoint cycles, with the proviso that the two moves be so interleaved that performing both takes us back to a full tour. Operationally, one starts by breaking four edges in the tour. Suppose the resulting four tour segments are $A_1 A_2 A_3 A_4$, in order. The move permutes them into the new ordering $A_2 A_1 A_4 A_3$ (without reversing any of the segments) and this yields the new tour. Algorithms based solely on double-bridge moves have not to date been very effective, although Schnetzler [1992] has had some success with a generalization in which several bridge moves are performed and the resulting subtours are then recombined, one pair at a time, using additional 2-Opt moves. Schnetzler does not provide full details of his algorithm and only reports results for two instances, but for one of these, the 532-city instance att532 from TSPLIB, he finds substantially better tours than 3-Opt. For the other, a 10 000-city random Euclidean instance, his algorithm does significantly worse and also takes substantially more time, even accounting for differences in machine speed. We thus suspect the approach does not scale well, at least as currently implemented.

Dynasearch

Potts & Van de Velde [1995], building on earlier work of Carlier & Villon [1990], have suggested another interesting approach that combines several 2-Opt moves into a single overall move. In a sense, their algorithm is simply a generalization of the variant of 2-Opt in which one always chooses to make the best improving move. Typically many of the improving moves found during one step of the 'choose-the-best' variant remain legal even after one of them has been made, so

starting over from scratch in one's search for the next improving move can involve substantial amounts of redundant effort. Potts & Van de Velde attempt to eliminate some of this redundancy as follows. For a given move M (2-Opt, 2.5-Opt, or 3-Opt), let $c(M)$ be the set of cities that have a tour neighbor changed by M. In the terms of Section 3.3, this would be $\{t_1, t_2, t_3, t_4\}$ for 2-Opt. If we picture the current tour as a Hamiltonian path running from $\pi(1)$ to $\pi(N)$, Potts & Van de Velde declare two moves M_1 and M_2 to be *independent* so long as either every city in $c(M_1)$ comes before every city in $c(M_2)$ or vice versa. Clearly, any set of independent improving moves can be made simultaneously and the result will be a tour whose length is improved by the sum of the individual improvments. For 2-Opt, Potts & Van de Velde observe that the best such set can be determined in $O(N^2)$ time using dynamic programming, and their algorithm works by repeatedly finding such sets and performing them until no improving 2-Opt move remains. They tested this *dynasearch* approach to 2-Opt, along with analogous schemes involving 2.5-Opt and 3-Opt, on random Euclidean instances with from 100 to 1000 cities.

The most promising results concern the tour quality yielded by the dynasearch variants of 2-Opt and 2.5-Opt, which appear to do better on average than the standard neighbor list implementations of the corresponding algorithms. The running times unfortunately appear to be far slower than those for neighbor-list 3-Opt, which yields better tours than any of the dynasearch variants. Potts & Van de Velde do observe speedups for dynasearch 2.5-Opt and 3-Opt over the corresponding choose-the-best variants of 2.5-Opt and 3-Opt. But the choose-the-best variants evaluate all possible moves and are far slower than the neighbor-list implementations described here, which restrict the set of moves to be examined and make the first improving move found. Thus, although the dynasearch variants could be sped up substantially by using neighbor lists themselves, we are not optimistic about their ultimate competiveness.

GENI and GENIUS

We conclude this section with a brief discussion of two other recent additions to the local optimization category: the GENI and GENIUS algorithms of Gendreau, Hertz & Laporte [1992]. GENI is a hybrid of tour construction with local optimization. Suppose that $c_1, c_2, \ldots, c_N$ is an arbitrary ordering of the cities. We begin with the partial tour consisting of the first three cities, c_1, c_2, c_3. We then add cities to the current tour in the order given, starting with c_4. To add city c_i, we consider possible ways of inserting it into the tour and then performing a 3-Opt or 4-Opt move with c_i as an endpoint of one of the deleted edges. The range of possibilities is restricted by requiring that certain of the inserted edges link cities to members of their nearest neighbor lists, where only cities currently in the tour qualify to be on such lists, and the lists are constrained to have maximum length p. See the paper for more complete details.

Gendreau, Hertz & Laporte [1992] present results for GENI on random Euclidean instances with N ranging from 100 to 500 and p ranging from 2 to 7.

The running times they report do not include the time for constructing the neighbor lists, but the authors have provided us with their code, thus enabling us to measure the full running times and also to test the algorithm on much larger instances. Our conclusion: if one were willing to classify this algorithm as a legitimate tour construction heuristic, it would be the best tour construction currently known. Even with $p = 3$ it finds better tours than Christofides, with an average percentage excess over the Held–Karp bound that seems to be approaching 9.1% in the limit, as compared to 9.8% for Christofides. If we take $p = 20$, the limiting average percentage excess drops to around 5.6%, close to the performance of our 2-Opt implementation, which has a limiting average of 4.9%. Moreover, GENI's average excess approaches its limit from below; for $N = 100$ it is actually better than 2-Opt so long as we take $p \geqslant 5$ (although still not as good as 3-Opt even with $p = 20$).

Running times are another matter, however. For $N = 1000$, GENI takes 20 times longer than *neighbor-list* 2-Opt even when $p = 3$. It is 500 times slower than 2-Opt when $p = 20$. These ratios become even larger as N increases. Some of this gap is no doubt due to the fact that the code provided has yet to be seriously optimized. It is likely that several of the ideas described above for 2-Opt and 3-Opt are applicable, although at this point it seems unlikely that they will be able to close the gap entirely.

The GENIUS algorithm is a true local optimization algorithm based on somewhat the same principles as GENI. Given a starting tour generated by GENI, it cycles through the cities looking for improvements. If c_i is the current city, GENIUS first looks for the best way to delete c_i from the tour, following that deletion by a restricted 3-Opt or 4-Opt move (restrictions analogous to those in GENI). This is called an *unstring* operation. GENIUS then looks for the best way to reinsert c_i as it would under GENI, a *string* operation. Although the paper suggests that uphill moves are allowed, the code as provided to us by the authors actually works in true local optimization mode: if the unstring/string operations chosen for c_i fail to improve the tour, they are undone before proceeding to the next choice for c_i. GENIUS improves significantly on GENI, but not enough to yield better average tours than 3-Opt except for our $N = 100$ instances. (For these it beats 3-Opt by about 0.1% as soon as $p \geqslant 8$). On the other hand, GENIUS's running time is worse than that for GENI by a factor of 3 or more, with the ratio growing with N (at least in the implementations provided to us). For $N = 1000$, it is 50 times slower than 3-Opt even when $p = 3$.

Thus if we want a practical way to improve on 3-Opt, it appears we may have to go beyond simple local optimization. In Sections 4 through 7 we examine the current alternatives.

4 TABU SEARCH AND THE LIN–KERNIGHAN ALGORITHM

Tabu search, like simulated annealing and several of the other techniques discussed in this book, is motivated by the observation that not all locally optimal solutions need be good solutions. Thus it might be desirable to modify a pure

local optimization algorithm by providing some mechanism that helps us escape local optima and continue the search further. One such mechanism would simply be to perform repeated runs of a local optimization algorithm, using a randomized starting heuristic to provide different starting solutions. The randomized Greedy heuristic we use for generating starting tours in our codes for 2-Opt and 3-Opt was chosen with just this thought in mind. The performance gain from such a random restart approach turns out to be limited, however, and decreases with increasing N. For instance, although the best of 100 runs of 2-Opt (from randomized Greedy starts) on a 100-city random geometric instance will typically be better than an average 3-Opt solution, for 1000-city instances, the best of 100 runs of 2-Opt is typically worse than the *worst* of 100 runs of 3-Opt.

One reason for the limited effectiveness of the random restart policy is that it does not exploit the possibility that locally optimal solutions might cluster together, that is, for any given local optimum, a better one might well be nearby. If this were true, it would be better to restart the search close to the solution just found, rather than at a randomly chosen location. This is in essence what tabu search does. The general strategy is always to make the best move found, even if that move makes the current solution worse, i.e., is an *uphill* move. Thus, assuming that all neighbors of the current solution are examined at each step, tabu search alternates between looking for a local optimum and, once one has been found, identifying the best neighboring solution, which is then used as the starting point for a new local optimization phase. If one did just this, however, there would be a substantial danger that the best move from this 'best neighboring solution' would take us right back to the local optimum we just left or to some other recently visited solution. This is where the *tabu* in tabu search comes in. Information about the most recently made moves is kept in one or more *tabu lists*, and this information is used to disqualify new moves that would undo the work of those recent moves.

A full-blown tabu search algorithm also involves several other factors as described in Chapter 5. For instance, there are *aspiration level* conditions, which provide exceptions to the general tabu rules, typically in situations where there is some guarantee that the supposedly forbidden move will not take us back to a previously see solution. There are also *diversification* rules, which can provide something like a random restart, *intensification* rules, which constrain the search to remain in the vicinity of a previously found good solution, and a host of other possible modifications, including various schemes for dynamically modifying the underlying neighborhood structure; see Glover [1986, 1989,1990]. In Section 4.1, we survey the literature on simple tabu search algorithms for the TSP based on 2-Opt and 3-Opt moves. These have not to date been notable for their successes. The most successful application of the underlying principles is in fact the famous algorithm of Lin & Kernighan [1973], which was invented over a decade before Glover first proposed tabu search as a general approach. This algorithm uses what are in effect tabu lists, but in a much more complicated scenario than that presented above. We describe the Lin–Kernighan algorithm and its behavior on

our test beds in Section 4.2. We postpone to Section 6 the coverage of more complicated tabu search algorithms, such as those that use Lin–Kernighan itself as a subroutine, since they have much in common with genetic algorithms, the topic of that section.

4.1 Simple tabu search algorithms

Given all the flexibility inherent in the tabu search approach, not to mention the latitude one always has with respect to picking neighborhood structures and cost functions for particular problems, it is not surprising that a wide variety of tabu search algorithms have been proposed for the TSP; see Glover [1986], Rossier, Troyon & Liebling [1986], Troyon [1988], Malek, Guruswamy & Pandya [1989], Heap, Kapur & Mourad [1989], Knox & Glover [1989], Glover [1991, 1996], Fiechter [1994], and Knox [1994]. In this section we will restrict attention to those algorithms for which we have seen experimental results reported, thus narrowing the field somewhat.

The first tabu search algorithm implemented for the TSP appears to be the one described by Glover [1986]. Limited results for this implementation and variants on it were reported by Glover [1989], Knox & Glover [1989], Knox [1989, 1994], and similar approaches were studied by Troyon [1988] and Malek, Guruswamy & Pandya [1989]. All these algorithms use 2-Opt exchanges as their basic moves, but they differ as to the nature of the tabu lists and the implementation of aspiration levels. Glover's original papers are unclear as to which approaches were actually implemented, but they mention several alternatives. For instance, Glover [1986] suggests adding the shortest of the two edges deleted by a move to the tabu list, with a subsequent move declared tabu if it tries to reinsert an edge currently on the tabu list. Aspiration levels can then be associated with tour lengths, and work essentially as follows: the aspiration level $A(L)$ is the length of the shortest tour that has ever been converted to one of length L in a single move. A tabu move can be declared acceptable only if it yields a tour length less than $A(L)$, where L is the current tour length. Note that this implementation of aspiration levels is far more restrictive than would be needed simply to prevent cycling. Knox [1994] designed his tabu lists and aspiration levels to allow significantly more moves to be considered. The tabu list consists of pairs of edges, each pair being the set of edges deleted in some previous move. Note that, given the overall tour, this uniquely specifies the 2-Opt move, although depending on the tour there are two possible 2-Opt moves to which a given pair of deleted edges might correspond. A subsequent move is tabu only if it reinserts a pair of edges currently on the tabu list. Aspiration levels are associated directly with the tabu moves and are much more forgiving; a tabu move is acceptable so long as it produces a tour that is better than the tour which existed when the corresponding pair of edges was most recently deleted. The tabu mechanisms used by Troyon [1988] and Malek, Guruswamy & Pandya [1989] are based on the endpoints of the changed edges rather than the edges themselves, but they appear to be similar in flavor.

It is not clear which of these various mechanisms is to preferred. For the above algorithms the question is moot, however, as all of them require $\Omega(N^3)$ running time and are unlikely to be practical for instances of significant size. The implementations of Glover [1989], Malek, Guruswamy & Pandya [1989], and Knox [1994] examine all possible 2-Opt moves at each step (to find the best one), for a total time of $\Theta(N^2)$ per step, and each performs $\Omega(N)$ steps. Counting repeated restarts, Knox [1994] actually performs $\Theta(N^4)$ moves, for an overall time of $\Theta(N^6)$, as confirmed by his reported running time data.

Are such high running times necessary? The need for at least $\Omega(N)$ steps seems consistent with the number of moves we observed for 2-Opt in Table 8.5, but must it really cost $\Theta(N^2)$ to determine the next move? If we truly want to find the best neighbor at each step, our options for reducing this cost are limited, although some obvious improvements can be made. For instance, we can exploit the fact that many potential moves do not experience a change in cost when a 2-Opt move is performed. Recall that a 2-Opt move may be viewed as breaking the current tour into two segments, reversing one, then gluing the tour back together. After such an alteration, any 2-Opt move with both its deleted edges on the same one of the two segments retains its previous cost. Thus the total number of moves that need to be reevaluated is roughly bounded by the product of the lengths of the two segments. This will yield at least a factor of 4 improvement in running time. For random Euclidean instances, the improvement is even better, because the length of the shorter segment has been observed to grow as about $\Theta(N^{0.75})$ under 2-Opt [Bentley, 1992; Fredman et al., 1995]. The product of the segment lengths will thus be only $\Theta(N^{1.75})$ and the overall running time will be reduced by a factor of $\Theta(N^{0.25})$.

But this still leaves us far behind implementations of 2-Opt and 3-Opt, which can substantially truncate the search space by exploiting the fact that uphill moves can be ignored (a luxury not allowed in tabu search). Some more savings are possible, however, if we take a hybrid approach: proceed as in 2-Opt until one reaches a locally optimal solution (with respect to nontabu moves and tabu moves that meet the aspiration criteria), and only then spend $\Theta(N^2)$ to find the best uphill move. To obtain full advantage from this approach, we would use the truncated neighbor lists and don't-look bits of our neighbor-list implementation during the local optimization phase, settling for the possibility that the best possible improving move will occasionally be missed. Troyon [1988] takes the more radical approach of simply taking the *first* (rather than the best) improving move found that is either not tabu or else yields an improvement over the best tour previously seen. If no improving moves are seen among the first cN examined (for some constant c, typically 1/2, 1, or 2), then the best nontabu uphill move seen so far is chosen. This reduces the time per move from $\Theta(N^2)$ to $\Theta(N)$, although the savings are counterbalanced by the fact that Troyon performs $\Theta(N^2)$ steps, yielding a $\Theta(N^3)$ running time after all. Nevertheless, he was able to run his implementation on instances with as many as 1000 cities on a machine roughly 15 times slower than our SGI Challenge. (His running time for the 1000-city instance however, was some 12 000 times slower than we report for neighbor-list 3-Opt in Table 8.6, 4 hours versus 1.14 seconds.)

We thus can conclude that straightforward tabu search implementations cannot to date compete with neighbor-list 3-Opt in terms of running time. What about tour quality? Here it is a bit difficult to say. Glover [1989], Malek, Guruswamy & Pandya [1989], and Knox [1994] restrict themselves to instances of 110 cities or less (quite natural, given the running times involved). Moreover, the instances they test are sufficiently easy that the key question seems to be how often the optimal solution is found, not how close to optimal is the average solution. Knox [1994] is the only author to have undertaken a comparison with 3-Opt, but he looks at how often the optimal solution is found. Knox reports that, even when he devoted as much time to 3-Opt as to his $\Theta(N^6)$ tabu search algorithm (by performing multiple runs of 3-Opt from random starts), tabu search had a higher probability than 3-Opt of finding the optimum. The tests only covered instances of up to 75 cities, however, and involved what appears to be a very inefficient implementation of 3-Opt, so the implications of these results are unclear.

Among the above researchers, only Troyon [1988] considers instances large enough for our key question of closeness to optimality (rather than optimality itself) to be relevant. In particular, he considers the instances of size 442 and 532 from TSPLIB, plus a 1000-city random Euclidean instance. Unfortunately, the results are not promising. For the first two instances, the average tour lengths he reports are substantially worse than the neighbor-list 3-Opt averages for this instance (although they are at least better than the average for neighbor-list 2-Opt). Comparisons are more difficult for the 1000-city instance, since we do not have access to it, but the tour length reported is worse than the average tour lengths obtained by 2-Opt for each of the six 1000-city random Euclidean instances in our test bed, suggesting this implementation of tabu search degrades substantially as N increases.

It would seem that straightforward tabu search implementations, like those above, are unlikely to be competitive with 3-Opt, either in terms of speed or tour quality. One idea for addressing the speed issue is to use parallelism. Heap, Kapur & Mourad [1989] propose a geometric partitioning scheme, as discussed in Section 3.4, with tabu search applied to the subproblems. Here with k processors one can hope for a speedup factor of k^3, assuming that the basic tabu search algorithm takes time $\Omega(N^3)$. Unfortunately, for the 532-city instance from TSPLIB on which they tested the approach, values of k greater than 8 appear to reduce tour quality below that of neighbor-list 2-Opt, and the speedup obtainable by taking $k \leqslant 8$ is not nearly enough to reduce the running time below that of 3-Opt. On a machine whose processors are roughly 150 times slower than those on our SGI Challenge, their best parallel running time is still 750 times slower than the average time for neighbor-list 3-Opt, and the tour they produce is significantly worse than an average 3-Opt tour.

Parallelism is also exploited in the much more sophisticated tabu search algorithm of Fiechter [1994]. This algorithm incorporates its tabu search ideas at a much higher level than the basic 2-Opt move, however, and is closer in structure to the genetic algorithms of Section 6. We shall therefore postpone its coverage until then. In the remainder of this section we discuss the famous Lin–Kernighan

algorithm, whose inner loop can be viewed as a variant of tabu search over the 2-Opt neighborhood structure, designed to fully exploit the techniques for search space truncation used in neighbor-list 2-Opt and 3-Opt. This inner loop in turn is embedded in an overall procedure that can be viewed as embodying the tabu search idea of *intensification*. The result is an algorithm that in practice yields significantly better results than 3-Opt with only a modest increase in running time.

4.2 The Lin–Kernighan algorithm

For over a decade and a half, from 1973 to about 1989, the world champion heuristic for the TSP was generally recognized to be the local search algorithm of Lin & Kernighan [1973]. This algorithm is both a generalization of 3-Opt and an outgrowth of ideas the same authors had previously applied to the graph partitioning problem in Kernighan & Lin [1970], ideas that have much in common with tabu search.

Both algorithms are described at a high level in Chapter 2. In this section we will give a more complete description of the algorithm for the TSP. We base our description on the implementation of Johnson et al. [1997b]. This implementation was derived from the original Lin–Kernighan algorithm, but it incorporates more modern data structures and new ideas such as the don't-look bits used in the neighbor-list implementations of 2-Opt and 3-Opt described above. We then summarize some of the performance results obtained by Johnson et al. [1997b] for this implementation. Because of the complexity of the algorithm's description, and perhaps because Lin & Kernighan [1973] did not directly apply it to any instances with more than 106 cities, many practitioners have apparently been under the impression that the Lin–Kernighan algorithm is too slow to handle instances much larger than this. As we shall see, however, its modern incarnation needs less than 50 minutes on a modern machine to handle a *million-city* instance.

In general, the fame of the Lin–Kernighan algorithm seems to have spread far wider than knowledge of its details, and not all claims in the literature about the behavior of 'Lin–Kernighan' should be taken at face value. Many authors rely on implementations that omit key components of the algorithm and end up producing tours that are worse on average than those produced by 3-Opt. Other authors seem to think that 'Lin–Kernighan' is a synonym for 3-Opt, or even 2-Opt! The description that follows, although not fully detailed, should provide enough information to enable readers to spot such spurious claims on their own. We begin our description of Lin–Kernighan with the inner loop, which we shall call an *LK search*.

The Lin–Kernighan inner loop

As in the tabu search algorithms described above, an LK search is based on 2-Opt moves, although a significantly restricted subset of them. There are several levels of restriction, with a tabu criterion being only one of them. To understand the

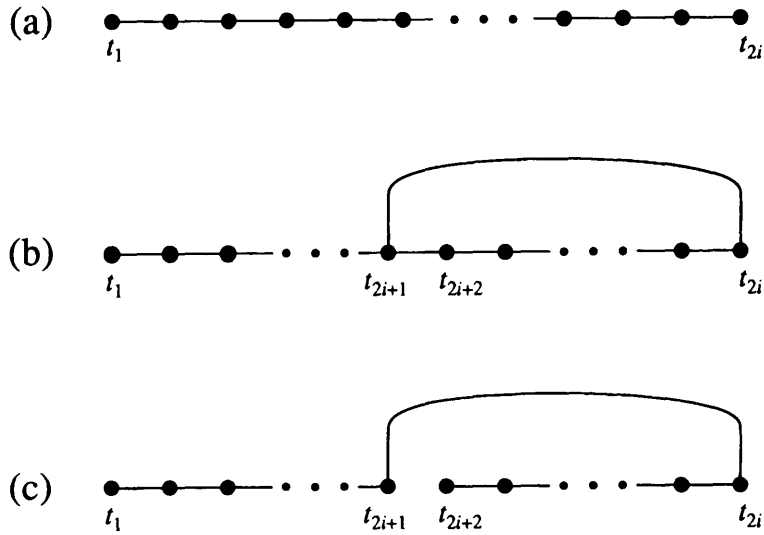

Figure 8.3 LK search: (a) path P_i; (b) one-tree determined by the ith move, required to be shorter than the best tour seen so far; (c) path P_{i+1}

other criteria, it is convenient to view the current solution as an anchored Hamiltonian path P rather than as a Hamiltonian circuit. The *anchor* of the path is a fixed endpoint city t_1, as illustrated in Figure 8.3(a). Let t_{2i} denote the other endpoint of the path P_i that exists at the beginning of step i of the LK search. The tour corresponding to path P_i can then be obtained by adding the edge $\{t_{2i}, t_1\}$.

At each step we only consider 2-Opt moves that flip some suffix of the path, i.e. ones in which one of the tour edges being broken is $\{t_1, t_{2i}\}$. Furthermore, the new neighbor t_{2i+1} of t_{2i} must be such that the length of the *one-tree* (spanning tree plus one edge) obtained by adding the edge $\{t_{2i}, t_{2i+1}\}$ to P_i (as in Figure 8.3(b)) is less than the length of the best tour seen so far. This restriction is a generalization of the criterion in 2-Opt that $d(t_2, t_3)$ be less than $d(t_1, t_2)$. The general one-tree restriction can be implemented using neighbor lists in much the same way as the corresponding restriction for 2-Opt, since all qualifying candidates for t_{2i+1} will be at the head of the neighbor list for t_{2i}. As in the neighbor-list implementations of 2-Opt and 3-Opt, using neighbor lists of length k imposes the additional constraint that only the k nearest cities to t_{2i} can be considered as candidates for t_{2i+1}, even if additional cities would satisfy the one-tree restriction.

As far as tabu conditions, Lin & Kernighan [1973] propose maintaining two lists, one of *added* edges (those that play the role of $\{t_{2i}, t_{2i+1}\}$ in Figure 8.3), and one of *deleted* edges (those that play the role of $\{t_{2i+1}, t_{2i+2}\}$). A move is tabu if it attempts either to add an edge on the *deleted* tabu list or to delete an edge on the *added* tabu list. But unlike standard tabu techniques, the LK search places no bound on the length of the tabu list and makes no use of aspiration levels. Thus

there is no escape from tabuhood: once a particular edge has been added to (deleted from) the path, it can no longer be deleted (added). This in particular implies that at most N moves can be made before the process must halt. (An LK search terminates when there are no qualifying nontabu moves for the current path P_i.)

The Johnson et al. implementation omits the *deleted* tabu list, so a move is tabu in this implementation only if it attempts to delete a previously added edge. This is still enough to insure at most N moves can be made, but allows a wider range of moves to be considered. Papadimitriou [1992] analyzes the alternative variant in which the *added* tabu list is omitted, so the only restriction is that we cannot add a previously deleted edge.This variant, which to the best of our knowledge has not been implemented, could conceivably make $N(N-1)/2$ moves and has the additional disadvantage of being PLS-complete (see Chapter 2), a property that has not yet been proved for the original algorithm or the Johnson et al. variant.

To complete our description of an LK search, we must say which of the qualifying, nontabu moves is selected. First, for each such move, we check to see if the resulting tour is better than the best previously seen, and if so we save it for future reference. We do not necessarily choose this move as the basis for subsequent exploration, however. Instead, we choose the (possibly different) qualifying, nontabu move that yields the shortest new path P_{i+1}. Note that this does not mean that the resulting path P_{i+1} will be shorter than P_i or that the corresponding *tour* will be shorter than the one corresponding to P_i. Thus uphill moves are allowed (as in tabu search).

The key differences from standard tabu search are (a) unbounded tabu lists with no aspiration levels, and (b) restrictions on the set of moves that are considered at each step. Both these differences serve to limit the amount of computation needed. The first restricts us to at most N steps per LK search, although in practice the number of steps averages something more like 3 for our random Euclidean instances, independent of size. The second restricts us to at most k alternatives, where k is typically 20 or 40 (and in practice the number that qualify is more like 3.5 for $k \geqslant 20$).

This saving in time comes at a price, since by limiting the search so substantially, we limit its effectiveness. The key to the success of the Lin–Kernighan algorithm lies in the fact that a large number of LK searches are performed and in the way in which the starting tours for these searches are obtained, which embodies the tabu search principle of *intensification*.

Restart mechanisms for the LK search

The simplest way to view the overall Lin–Kernighan algorithm is as neighbor-list 3-Opt with LK search grafted on. More specifically, we proceed as in 3-Opt, considering all possibilities for t_1, t_2, t_3, t_4, and t_5 that satisfy $d(t_2, t_3) < d(t_1, t_2)$ and $d(t_2, t_3) + d(t_4, t_5) < d(t_1, t_2) + d(t_3, t_4)$ (i.e., all choices of t_1 through t_5 meeting the one-tree restriction at the first and second levels). For each of these

choices, we use the tour that would result from performing the corresponding 3-Opt move as the starting point for an LK search.

Note that for some choices of t_1 through t_5 there will be two possible ways of generating a corresponding 3-Opt move (two potential candidates for t_6 that yield legal 3-Opt moves), and both are considered. For other choices, there may be no way to generate a legal 3-Opt move, and in these cases we attempt to find cities t_6, t_7, and t_8 that together with t_1 through t_5 yield a legal 4-Opt move that meets the one-tree restriction (with t_7 restricted to those cities on the neighbor list for t_6). The first such move found is used to produce the starting tour. In addition, we preload the tabu lists for the LK search with the edges added by the 3-Opt or 4-Opt move that yielded the initial path for the search. (The average of three steps per LK search mentioned in the previous section did not include the work performed in creating these initial paths.)

The algorithm proceeds in a series of phases based on the notion of a *champion* tour, i.e., the best tour seen so far. Initially, this is just the tour produced by a starting heuristic, such as the Greedy algorithm. Until a new champion is crowned, all of our LK searches are based on tours obtained from this champion by 3-Opt (4-Opt) moves, with the search proceeding systematically through possible choices of t_1 through t_5 as in 3-Opt. Note that this method for restarting an LK search naturally embodies the tabu search concept of *intensification*, since it ensures that we keep exploring the vicinity of the current champion tour.

Whenever a tour is found that is better than the current champion, we complete the current LK search and then take as our new champion the best tour found during that search. (If a given choice of t_1 through t_4 yields a legal 2-Opt move that improves the current champion, and none of the LK searches derived from this choice yields a better improving move, we take as our new champion the tour which results from performing that 2-Opt move.) Whenever a new champion is crowned, we enter a new phase based on this new champion, restarting the basic 3-Opt loop with the next available value for t_1. (The Johnson et al. implementation uses don't-look bits to restrict the choices for t_1, as in neighbor-list 2-Opt and 3-Opt.) The algorithm terminates when all possible choices of t_1 through t_5 have been considered for a given champion without yielding an improvement, i.e., when a tour is found that is locally optimal with respect to the expanded neighborhood structure implicit in the Lin–Kernighan algorithm itself.

Variants on this restart strategy have been considered in the literature, mostly with the idea of speeding up the algorithm. A common approach is to restrict attention to choices of t_1 through t_6 that yield valid 3-Opt moves and are such that t_1 through t_4 also yield a valid 2-Opt move. In addition, the number of LK searches made for each choice of t_1 can be more directly restricted. Whereas the Johnson et al. implementation considers both tour neighbors of t_1 as candidates for t_2, many implementations consider only one possibility for t_2, typically the successor of t_1 in the current tour. Given this, the 'no backtracking' strategy of Mak & Morton [1993] simply starts an LK search as soon as t_1 has been chosen and its successor t_2 identified. Reinelt [1994] and Jünger, Reinelt & Rinaldi [1995] suggest considering only the first three possibilities for t_3 and starting an

LK search as soon as t_3 has been chosen. (This yields at most three LK searches for each choice of t_1.) Mak & Morton [1993] suggest allowing alternatives for both t_5 and t_3, but only considering the first five choices for each (for a bound of 25 on the total number of LK searches for a given choice of t_1). In another variant suggested by Reinelt [1994] and Jünger, Reinelt & Rinaldi [1995], two options each are allowed for t_3, t_5 and t_7, yielding eight possible LK searches for each choice of t_1. But only the more Draconian of these approaches are likely to provide substantial speedups by themselves. With neighbor lists of length 20, the Johnson et al. implementation can theoretically perform 800 or more LK searches per choice of t_1, but the actual average number of calls is more like six for random Euclidean instances and eight for the larger of our random distance matrix instances.

Another common modification that has been proposed is to limit the depth of LK searches, say to 50 steps [Applegate, Chvátal & Cook, 1990] or even 15 [Reinelt, 1994; Jünger, Reinelt & Rinaldi, 1995]. This again is unlikely to cause any significant speedup of the algorithm by itself, given that the average depth searched even when no bounds are imposed is only three moves beyond the level at which the LK search is initiated. It does however, enable use of the *segment-tree* data structure for tour representation proposed by Applegate, Chvátal & Cook [1990], which for certain classes of instances is a serious competitor to the two-level tree data structure used in the Johnson et al. implementation. See Fredman et al. [1995] for a detailed experimental comparison.

Finally, there have been various proposals to modify the LK search method more drastically, either by using shorter neighbor lists to further limit the alternatives considered for t_{2i+1} [Reinelt, 1992; Jünger, Reinelt & Rinaldi, 1995], or conversely, by augmenting the class of possible moves considered. Mak & Morton [1993] suggest allowing 2-Opt moves that flip a prefix of the current path (as well as the standard ones that flip a suffix). Reinelt [1994] and Jünger, Reinelt & Rinaldi [1995] suggest allowing moves in which a single city is moved from its current position to the end of the path. Dam & Zachariasen [1994] have spelled out an even more flexible variant called a *flower transition*, based on proposals made by Glover [1991, 1996]. In this search method, the base configuration is not a Hamiltonian path, as in LK search, but a one-tree consisting of a cycle, the *blossom*, attached to a path, the *stem*. Such a graph has more flexibility as there are two ways it can be turned into a tour, depending on which of the cycle edges adjacent to the stem is deleted. Individual steps of a search again involve adding an edge from the free end of the path, but now that edge may go either to another stem city or to a city in the cycle, and in the latter case there are again two choices for which edge to delete.

Unfortunately, it is difficult to determine which of these suggested augmentations to LK search have merit. Although experimental results are reported for each of them, the implementations studied differ from the Johnson et al. implementation of Lin–Kernighan in more ways than just the choice of search method. Dam & Zachariasen use the flower transition in a much more elaborate algorithm, which we shall describe in Section 6. The other variants were studied in

implementations that imposed additional restrictions on t_1 through t_7, like those mentioned above. Perhaps because of these restrictions, they yielded tours that were typically much worse that those of the Johnson et al. implementation. The Johnson et al. implementation also appears to be faster, although this may be more a result of all the tuning that went into its construction rather than any key algorithmic advantages. The next section provides specific details on its performance.

Experimental results for Lin–Kernighan

Tables 8.7 and 8.8 summarize the baseline experimental results for the Johnson et al. implementation. Table 8.7 covers tour quality and Table 8.8 covers running time. Results are reported for both random geometric instances and random distance matrix instances. For comparison, the tables include the corresponding results for 3-Opt from Section 3.2. As with 3-Opt, the results for Lin–Kernighan in these tables were obtained using starting tours generated by the randomized Greedy heuristic, and with neighbor lists of length $k = 20$.

Table 8.7 Tour quality for Lin–Kernighan in comparison to 3-Opt: average percent excess over the Held–Karp lower bound

$N =$	10^2	$10^{2.5}$	10^3	$10^{3.5}$	10^4	$10^{4.5}$	10^5	$10^{5.5}$	10^6
				Random Euclidean instances					
3-Opt	2.5	2.5	3.1	3.0	3.0	2.9	3.0	2.9	3.0
LK	1.5	1.7	2.0	1.9	2.0	1.9	2.0	1.9	2.0
				Random distance matrices					
3-Opt	10.0	20.0	33.0	46.0	63.0	80.0	–	–	–
LK	1.4	2.5	3.5	4.6	5.8	6.9	–	–	–

Table 8.8 Running times for Lin–Kernighan in comparison to 3-Opt: seconds on a 150 MHz SGI Challenge

$N =$	10^2	$10^{2.5}$	10^3	$10^{3.5}$	10^4	$10^{4.5}$	10^5	$10^{5.5}$	10^6
				Random Euclidean instances					
3-Opt	0.04	0.11	0.41	1.40	4.7	17.5	69	280	1 080
LK	0.06	0.20	0.77	2.46	9.8	39.4	151	646	2 650
				Random distance matrices					
3-Opt	0.02	0.16	1.14	9.8	110	1 410	–	–	–
LK	0.05	0.35	1.90	13.6	139	1 620	–	–	–

Note that for random Euclidean instances, Lin–Kernighan finds tours that on average are about 1% shorter than those provided by 3-Opt, reducing the excess over the Held–Karp lower bound to just 2/3 of the amount under 3-Opt. For random distance matrices, the reduction is much more dramatic, more like a factor of 10, although the percentage excess does still appear to be growing as $\log N$. Nevertheless, the tours found are quite reasonable throughout the range of instance sizes studied, with the average excess over the Held–Karp bound only growing from 1.4% to 6.9%, compared to the excess of 80% for 3-Opt at the top of the size range.

As to running times, note that for random Euclidean instances, the ratio of the running time for Lin–Kernighan to that for 3-Opt, although growing with N, has only reached a factor of 2.5 for $N = 10^6$, with a 10 000-city instance taking less than 10 seconds, and a million-city instance taking less than 45 minutes. For random distance matrices, because the $\Theta(N^2)$ time for constructing neighbor lists dominates the total running time, the ratio actually decreases with N and is only 1.15 for $N = 31\,623$. The ratios of time spent in local optimization for the two algorithms (ignoring the time spent in the common operations of constructing neighbor lists and starting tours) are significantly larger, although not as large as one might expect. For the random geometric instances, the ratios grow from a factor of roughly 3 to a factor of 8 as one goes from 100 to 10^6 cities. For random distance matrices the ratios grow from a factor of roughly 4 to a factor of 14 as one goes from 100 to 31 623 cities.

Lin–Kernighan's performance on the more structured instances of `TSPLIB` is comparable to, but slightly worse than, its behavior for random geometric instances. In terms of tour quality, Lin–Kernighan's degradation is somewhat less than we observed for 3-Opt. The average excess of roughly 1.9% for our larger random geometric instances grows to an average of 2.8% for our suite of 1.5 of the largest instances in `TSPLIB`, whereas the increase for 3-Opt was from 3.0% to 4.6%. An even smaller degradation (to an average excess of just 2.3%) is possible if we use neighbor lists of 40 quadrant neighbors as described in Section 3.3, although this typically increases running time by about a factor of 2 over that needed when just the 20 nearest neighbors are used. (For random geometric instances this quadrant neighbor scheme yields essentially the same average percentage excess as the 20-nearest-neighbor scheme, but it appears to have real value on more structured instances.)

As to running time on these `TSPLIB` instances, it is fair to say that the good tours found by Lin–Kernighan are purchased at a cost. Whereas running times for 3-Opt on the `TSPLIB` instances were roughly the same as they would have been on random geometric instances of the same size, the `TSPLIB` times for Lin–Kernighan, even when using just 20 neighbors, were often a factor of 2 or 3 worse than the corresponding random geometric times (and in one case, `TSPLIB`'s 3795-city instance, the factor was more like 25). This slowdown is typically due to the combined effect of a deeper average search depth for the LK search (which increased from 3 to 37 for the 3795-city instance) and an increased number of qualifying moves at each level (which increased from 3.5 to 10.5 for

that instance). Even so, the running times are not exorbitant; the longest is for 40-quadrant-neighbor Lin–Kernighan on TSPLIB's 85900-city instance (pla85900), which took roughly 7 minutes but produced a tour that was within 1.6% of the Held–Karp lower bound.

Parallelizing Lin–Kernighan

Several schemes for constructing parallel versions of Lin–Kernighan have been proposed. Schemes based on geometric partitioning offer much the same trade-off between tour quality and speedup as they do for 2-Opt and 3-Opt. For pla85900 a geometric partition into four roughly equal subproblems can reduce running time by more than a factor of 4 (a superlinear speedup because Lin–Kernighan has superlinear running time), but the resulting tours average some 0.3% longer than those for the sequential version. A similar observation holds for schemes using tour-based partitioning. To date, the most effective application of parallelism to Lin–Kernighan seems to be that of Verhoeven, Swinkels & Aarts [1995]. Their scheme assumes that each processor has enough local memory to store the entire instance, its associated data structures, and the current tour, and it simply parallelizes the search for an improving move.

The process takes place in rounds, each of which consists of a *search phase* and an *arbitration phase*. In the search phase, the set of candidates for t_1's is partitioned among the processors, which then look for improving moves based on their assigned set of candidates. The search phase concludes as soon as each processor has either found an improving move or has discovered that none exist for its list of candidates. The arbitration phase deals with the case when more than one processor finds an improving move, and it has the task of deciding which one(s) to perform. In the Verhoeven et al. scheme, the moves are ordered by the amount of reduction in tour length they provide, the first is performed, then each remaining move is examined in turn to see if it is still legal and if so is also performed. (The processors all perform these changes to their local copies of the current tour in parallel.) Using this scheme, Verhoeven et al. obtain substantial speedups with no noticeable degradation in tour quality over their sequential implementation of Lin–Kernighan. With 32 processors, the speedups on TSPLIB instances range from a factor of 5 to a factor of 16 depending on the instance and the type of processor they are using. The version of Lin–Kernighan parallelized by Verhoeven et al. differs in several key ways from the Johnson et al. implementation, and on average the tours it produces are not quite as good and it does not run quite as quickly. Their parallelization scheme seems fairly robust however, and is likely to provide the same sort of speedups without loss of tour quality for a wide variety of underlying Lin–Kernighan implementations.

5 SIMULATED ANNEALING AND ITS VARIANTS

The invention of simulated annealing actually preceded that of tabu search, although it is more convenient for us to treat it second. Like tabu search,

simulated annealing allows uphill moves. However, whereas tabu search in essence only makes uphill moves when it is stuck in local optima, simulated annealing can make uphill moves at any time. Moreover, simulated annealing relies heavily on randomization, whereas tabu search in its basic form chooses its next move in a strictly deterministic fashion (except possibly when there is a tie for the best nontabu neighbor). Nevertheless, simulated annealing is still basically a local search algorithm, with the current solution wandering from neighbor to neighbor as the computation proceeds. The key difference from other approaches is that simulated annealing examines neighbors in random order, moving to the first one seen that is either better or else passes a special randomized test. As originally proposed by Kirkpatrick, Gelatt & Vecchi [1983] and Černý [1985], the randomized test is the one invented by Metropolis et al. [1953] for simulating the physical behavior of atoms in a heat bath. It involves a control parameter called the *temperature*, and in simulated annealing that control parameter is continually lowered as the search proceeds, in a simulation of the physical annealing process.

For a detailed description of simulated annealing, its motivation, and its theoretical underpinnings, see Chapter 4. The TSP was one of the first problems to which simulated annealing was applied, serving as an example for both Kirkpatrick et al. [1983] and Černý [1985]. Since then the TSP has continued to be a prime test bed for the approach and its variants. In this section we report on the resulting TSP algorithms and how they perform. Most adaptations have been based on the simple schema presented in Figure 8.4, with implementations

1. Generate a starting solution S and set the initial champion solution $S^* = S$.
2. Determine a starting temperature c.
3. While not yet *frozen* do the following:
   3.1. While not yet at *equilibrium* for this temperature, do the following:
      3.1.1. Choose a *random neighbor* S' of the current solution.
      3.1.2. Set $\Delta = Length\,(S') - Length\,(S)$.
      3.1.3. If $\Delta \leq 0$ (downhill move):
         Set $S = S'$.
         If $Length\,(S) < Length\,(S^*)$, set $S^* = S$.
      3.1.4. Else (uphill move):
         Choose a random number r uniformly from [0, 1].
         If $r < e^{-\Delta/c}$, set $S = S'$.
      3.1.5. End 'While not yet at equilibrium' loop.
   3.2. Lower the temperature c.
   End 'While not yet frozen' loop.
4. Return S^*.

Figure 8.4 General schema for a simulated annealing algorithm

differing as to their methods for generating starting solutions (tours) and for handling temperatures, as well as in their definitions of *equilibrium*, *frozen*, *neighbor*, and *random*. Note that the test in Step 3.1.4 is designed so that large steps uphill are unlikely to be taken except at high temperatures c. The probability that an uphill move of a given cost Δ will be accepted declines as the temperature is lowered. In the limiting case, when $c = 0$, the algorithm reduces to a randomized version of iterative improvement, where no uphill moves are allowed at all.

From the beginning, there has been a dichotomy between the way the schema of Figure 8.4 is implemented in practice and in theory. In theory, simulated annealing can be viewed as an optimization algorithm. As discussed in Chapter 4, the process can be interpreted in terms of Markov chains and proved to converge to an optimal solution if one insures that the temperature drops no more quickly than $C/\log n$, where C is a constant and n is the number of steps taken so far. Typically, however, the convergence to an optimal solution under this temperature schedule will take longer than finding an optimal solution by exhaustive search. So these theoretical results are essentially irrelevant to what can be accomplished in practice.

Instead, starting with Kirkpatrick et al. [1983] and Černý [1985], researchers have tended to use cooling schedules that drop the temperatures much more rapidly, say roughly as C^n, where $C < 1$. This can be realized by performing a fixed number of trials at each temperature, after which one arbitrarily declares 'equilibrium' and reduces the temperature by a standard factor, say 0.95. Under such an *exponential cooling* regime and after a polynomially bounded amount of time, the temperature will reach values sufficiently close to zero that uphill moves will no longer be accepted and we can declare freezing to have set in. This happens even when, as Kirkpatrick et al. originally suggested, one starts at a temperature c sufficiently high that essentially *all* uphill moves are accepted. With such a polynomially bounded cooling schedule, simulated annealing is only an approximation algorithm (like all our other local search variants), but it would be hard to expect more for an NP-hard problem like the TSP. Theory so far has little to say about this polynomial-time bounded version of simulated annealing, except in the context of specially invented problems and neighborhood structures; again see Chapter 4.

In this section we thus concentrate on empirical results. As we shall see, there are broad performance ranges where an intelligently implemented simulated annealing algorithm would be the method of choice were its only competition 2-Opt and 3-Opt. Even Lin–Kernighan cannot beat it across the board, although the *iterated Lin–Kernighan* algorithm of the next section can. Such powerful alternatives might not exist in other problem domains, however. Thus the lessons learned here may be of broader significance, and it is worth taking time to spell out some of the key algorithmic issues and ideas that are relevant to creating an effective implementation of simulated annealing for the TSP. Much of what follows is derived from the study of Johnson et al. [1997a], to which the reader is referred for additional technical details.

We begin in Section 5.1 by defining a baseline implementation similar to that of the original Kirkpatrick et al. paper and reporting on its behavior. In Section 5.2 we describe and evaluate two key ideas, neighborhood pruning and low-temperature starts, that provide more-than-constant-factor speedups and are essential if annealing is to be competitive with our more traditional TSP heuristics. Section 5.3 is devoted to other ideas for speeding up annealing and/or helping it find better tours. Finally, Section 5.4 discusses variants of simulated annealing where thresholds rather than probabilities govern the decision on whether to accept a move.

5.1 A baseline implementation of simulated annealing

In adapting simulated annealing to the TSP, both Kirkpatrick et al. [1983] and Černý [1985] suggested using a neighborhood structure based on 2-Opt moves, just as was later done for tabu search. Černý [1985] also considered the simpler move in which the positions of two cities are interchanged but the segment between them is left unchanged; this approach is less effective, as experiments have demonstrated. The results in these two papers were limited mainly to small examples and running times were not reported. Kirkpatrick et al. [1983] did run the algorithm on one problem of reasonable size (some 6000 cities), but they did not provide any detailed information on the quality of the solution found, except to say it way 'good'. Thus the value of simulated annealing for the TSP was initially unclear.

There does, however, appear to be a serious defect in the above straightforward approach to adapting simulated annealing to the TSP. As we shall see, the number of steps at each temperature, which we shall call the *temperature length*, needs to be at least proportional to the neighborhood size if we are to obtain worthwhile tour quality. This is not too onerous a restriction for problems like graph partitioning, where typical neighborhood sizes are $O(N)$ [Kirkpatrick et al., 1983; Johnson et al., 1989]. For the 2-Opt TSP neighborhood, however, the size is proportional to N^2, so that even if the number of distinct temperatures considered does not grow with N, we will still have an algorithm whose running time is at least $\Theta(N^2)$ with a large constant of proportionality. Although this may well be faster than the $\Omega(N^3)$ we saw for straightforward tabu search implementations based on the 2-Opt neighborhood structure, the real competition for simulated annealing is not tabu search but 3-Opt and Lin–Kernighan, which as we have already seen are both decidedly subquadratic up to $N = 10^6$.

This issue is illustrated in Table 8.9, which reports results for a baseline implementation of simulated annealing due to Johnson et al. [1997a], which in what follows we shall denote by SA_1. The table covers the random geometric instances in our test bed with $N = 100$, 316, and 1000. For these runs the temperature reduction factor was 0.95, the temperature length was $N(N-1)$, and the starting temperature was such that only about 3% of the proposed moves were rejected. To speed up running times, we do not explicitly compute the exponential in Step 3.1.4 of Figure 8.4, but instead find its value using the lookup

Table 8.9 Results for SA_1 using the full 2-Opt neighborhood and high initial temperature: random Euclidean instances

	10^2	$10^{2.5}$	10^3
SA_1 running time (s)	12.40	188.00	3 170.00
2-Opt running time (s)	0.03	0.09	0.34
3-Opt running time (s)	0.04	0.11	0.41
LK running time (s)	0.06	0.20	0.77
Number of temperatures	117	143	171
Average SA_1 percentage excess	5.2	4.1	4.1
Average SA_1 excess after 2-opting	3.4	3.7	4.0
Average 2-Opt excess	4.5	4.8	4.9
Average 3-Opt excess	2.5	2.5	3.1
Average LK excess	1.5	1.7	2.0

scheme of Johnson et al. [1989]. The table reports averages over 10 runs for each 100-city test-bed instance and 5 runs for each of the larger test-bed instances. This is not enough to obtain tight confidence intervals, but enough to illustrate trends. (As with our other algorithms, there is a considerable variation between instances and between runs on the same instance when $N = 100$, although this shrinks as N increases.)

Note that, because the number of temperatures grows with N, the running time growth rate is actually worse than quadratic. Most of this growth takes place at the end of the annealing schedule, when very few moves are being accepted but the current solution is still not locally optimal. In this implementation, the process is not considered frozen until five consecutive temperatures have passed without a new champion solution and without the acceptance percentage going above 2%. As N grows, so does the proportion of time spent with the acceptance percentage below this threshold (while the tour continues to improve). Even so, the final tours are typically not 2-optimal, so this annealing implementation has a postprocessing phase in which 2-Opt is applied. Note that the percentage effect of this postprocessing on tour length is substantial for $N = 100$ but decreases significantly as N increases. The final tour quality degrades slightly as N increases, but this is also true for 2-Opt, 3-Opt, and Lin–Kernighan when $N \leqslant 1000$. Thus holding the ratio of temperature length to neighborhood size constant, as we have done here, appears to provide roughly a constant level of performance relative to those competitors, independent of N. Choosing a temperature length simply equal to the neighborhood size has not yielded impressive results, however. The average final percentage excesses, although better than those for 2-Opt, are not as good as those for 3-Opt.

In summary, this baseline implementation SA_1 produces worse tours than 3-Opt on average, while taking almost 300 times as long when $N = 100$ and over

7500 times as long when $N = 1000$. Better results can be obtained if one is willing to spend even more time annealing. Increasing running time by a factor of 10 (by increasing temperature length to $10\,N(N-1)$) reduces the average final percentage excess for $N = 100$ from 3.4% to 1.9%, significantly better than 3-Opt (but taking 3000 times as long). This new excess is still worse than the 1.5% of Lin–Kernighan, however. Increasing temperature length by another factor of 10 reduces the average final percentage excess for $N = 100$ to 1.3%, now better than Lin–Kernighan (but taking 20 000 times as long).

Such increases in temperature length are not a practical option as N gets large, given the quadratic growth in running time for SA_1; using a temperature length of $100N(N-1)$ for $N = 1000$ would yield a running time some 400 000 times that for LK, i.e., some 3.5 days versus 0.77 seconds. Moreover, the fact that simulated annealing can do better than LK if allowed to spend so much more time is not necessarily an argument on its behalf. Lin–Kernighan too can profit from extra time. We shall consider more sophisticated ways of using the extra time in Section 6, but even a very simple scheme can be effective, such as performing multiple independent runs of LK and taking the best solution found. For instance, simply taking the best of 100 different runs of LK (using independently chosen randomized greedy starts) yields an average percentage excess of 0.9% for $N = 100$, better than the best average reported above for SA_1 and still in significantly less time. (Moreover, these 100 runs take only 70 times as long as one run, since the initial neighbor-list construction only needs to be performed once, so it can be amortized across all 100 runs.)

If simulated annealing is to be useful for the TSP, we need to find ways to get the effect of longer temperature lengths while somehow simultaneously reducing the overall running time substantially. It is intuitively clear that speeding up the algorithm by simply reducing the temperature length (or using fewer temperatures) can only make the average tour length worse. Early follow-ups on Kirkpatrick et al. [1983] that in effect took this approach, such as Nahar, Sahni & Shragowitz [1985] and Golden & Skiscim [1986], bear this out. Nahar et al. restricted their runs to just 6 temperatures (obtained using a reduction factor of 0.90) and adjusted the temperature length to keep the overall time within a fixed low multiple of the Lin–Kernighan time. Golden & Skiscim used 25 temperatures (evenly rather than geometrically spaced) as well as an adaptive temperature length that varied from temperature to temperature but whose average value was not likely to grow as N^2. Insofar as comparisons can be made, both resulting algorithms appear to have found worse tours than those found by SA_1, and they remain significantly slower than the neighbor-list implementations of 3-Opt and Lin–Kernighan.

5.2 Key speedup techniques

Effective speedup ideas do exist, fortunately. Bonomi & Lutton [1984] described two of them in another early follow-up to the Kirkpatrick et al. [1983] paper.

Neighborhood pruning

The first (and more crucial) idea was to prune the neighborhood structure. Although there are $\Theta(N^2)$ possible 2-Opt moves that can be made from any tour, a 2-Opt move that introduces a long edge into a good tour will typically make things much worse, and hence is unlikely to be accepted. If we can find a way to exclude such moves a priori, we might be able to greatly speed up the algorithm without significant loss in performance. Bonomi & Lutton suggested the following approach, applicable to geometric instances. As in our implementation of 2-Opt itself, let us view a 2-Opt move as determined by a choice of a city t_1, one of its tour neighbors t_2, and a third city t_3 which is to replace t_1 as a tour neighbor of t_2. We restrict the choice of t_3 as follows. Identify the smallest square containing all the cities, and divide it up into m^2 grid cells for some integer m. Then consider for t_3 only those cities that are either in the same cell as t_2 or in a neighboring cell. For random geometric instances with uniformly distributed cities, one can reduce the expected number of choices for t_3 to $O(1)$ by choosing m to be $\Theta(\sqrt{N})$. This in turn reduces the expected number of neighbors of a given tour from $\Theta(N^2)$ to $\Theta(N)$. Bonomi & Lutton combined neighborhood pruning with the following additional key idea.

Low-temperature starts

The second idea was to abandon the high temperature portion of the annealing schedule and instead start with a relatively low temperature. To reduce the probability that the process might be immediately trapped in an unproductive region, they proposed using a tour construction heuristic to generate the starting tour (as in traditional local optimization). This idea of low-temperature starts was also proposed by Kirkpatrick [1984], who used it without neighborhood pruning but still got sufficient speedups to make longer temperature lengths possible and thus improve on the results obtainable by baseline implementations like SA_1. Using nearest neighbor starting tours on 400- and 900-city random geometric instances, he appears to have obtained average percentage excesses of roughly 2.1% and 2.4%, respectively, only slightly worse than those for Lin–Kernighan. Kirkpatrick used the rectilinear metric as opposed to the Euclidean metric, but average percentage excesses for 2-Opt, 3-Opt, and LK are seldom greatly affected by such a change in metric. He did not compute the Held–Karp lower bounds for his instances, so we have estimated his average percentage excess based on the average Held–Karp lower bound for instances of this type and size, as determined using the techniques of Johnson, McGeoch & Rothberg [1996].

Combining the two

Combining the two ideas of low-temperature starts and neighborhood pruning, as is done by Bonomi & Lutton [1984], should yield even better results. For

starting tours, Bonomi & Lutton [1984] used a simple heuristic that generates a tour by stringing together tours for the individual cells into one overall tour, with the tour for each cell connected to tours in adjoining cells. They concentrated on random Euclidean instances and used an annealing schedule consisting of 50 temperatures with an initial temperature of $L/\sqrt{N}$, where L is the side of square in which the cities are randomly placed. They used a reduction factor of 0.925. They do not specify their temperature length, but it apparently grew more slowly than their neighborhood size, which for random geometric instances can be expected to grow as $\Theta(N)$. This means that the ratio of their temperature length to neighborhood size decreased as N increased. Given the results summarized in Table 8.9, this suggests that their average percentage excess should have increased significantly as N increased, and this indeed seems to have been the case. Although for instances from 200 to 400 cities they claim to have beaten 2-Opt substantially, this is not true for the one large instance they treat, a 10 000-city random geometric instance. Based on the Held–Karp estimates of Johnson, McGeoch & Rothberg [1996] for such instances, the tour length they report is likely to be roughly 6.9% in excess of the Held–Karp bound, far worse than the 5.0% that neighbor-list 2-Opt averaged on our test bed, all by itself! Just how well their implementation performs on the smaller instances is difficult to judge, as they only report normalized differences between the annealing and 2-Opt tour lengths, and the quality of the latter depends heavily on the details of their (unspecified) implementation of 2-Opt.

To get a clearer idea of the advantages of low-temperature starts and neighborhood pruning for the TSP, we added them to our baseline implementation. An obvious drawback to the pruning scheme of Bonomi & Lutton [1984] is that its ability to substantially reduce the overall neighborhood size depends crucially on the fact that cities are uniformly distributed. We used a more robust scheme suggested by the implementation of neighbor-list 2-Opt: In terms of the notation we used to describe that implementation, we simply restrict t_3 to the nearest 20 neighbors of t_2. This produces at most $40N$ neighbors for a given tour. All N cities are candidate for t_1, both tour neighbors of t_1 are candidates for t_2, and t_4 is uniquely determined, given prior choices for t_1, t_2, and t_3. (Note that some of these $40N$ neighbors may actually represent the same tour.)

For improved performance, we augment the above static pruning rule for t_3 with an additional dynamic one. At the beginning of each temperature c we restrict the neighbor list for each city t as follows: let t' be the current tour neighbor of t that is farther away. When $t_2 = t$ at this temperature, candidates for t_3 will be restricted to those cities t'' on the neighbor list for t such that a 2-Opt move that increases tour length by $d(t,t'') - d(t,t')$ would be accepted with probability at least 0.01, i.e., those cities t'' for which $e^{-(d(t,t'') - d(t,t'))/c} \geqslant 0.01$. Assuming that the neighbor lists are sorted by increasing distance, this can be accomplished by simply maintaining a pointer to the last acceptable city on each list, with the pointers updated once per temperature. As to temperature length in this dynamic environment, we leave that fixed at some constant multiple α of the total lengths of the initial neighbor lists, i.e., at $\alpha 20N$ as opposed to $\alpha N(N-1)$ for

the unpruned neighborhood. This means the effective temperature length actually increases as the dynamic pruning starts shrinking neighbor lists. The increase is typically from the initial α times current total neighborhood size to four or five times that amount.

As to low-temperature starts, we follow Kirkpatrick [1984] in using nearest neighbor starting tours, and we follow Bonomi & Lutton [1984] in letting the initial temperature be proportional to $L/\sqrt{N}$ for random Euclidean instances, although we choose the value $1.5\,L/\sqrt{N}$ rather than the $L/\sqrt{N}$ of Bonomi & Lutton [1984]. This produces an initial acceptance rate of about 50%, and it allows the tour length initially to grow by about a factor of 2 from its starting value. For other types of instance, we take this 50% initial acceptance rate as our criterion and determine an appropriate starting temperature by trial and error, although in retrospect it appears that multiplying the length of the nearest neighbor tour by $(1.5/N)$ typically gives a reasonable value, at least for geometric instances. We retain the temperature reduction factor of 0.95 from our baseline implementation.

Results for baseline annealing, annealing with pruning, and annealing with both pruning and low-temperature starts are summarized in Table 8.10 for various values α. For comparison Table 8.10 also includes the average results for 2-Opt, 3-Opt, and Lin–Kernighan, as well as the averages for the best of multiple runs of each algorithm (1000 and 10 000 runs for 2-Opt and 3-Opt and 10 or 100 runs for LK). To estimate these 'best of' results, we performed 10 000 runs of each algorithm and assumed that the resulting distributions of tour lengths were the true probability distributions. This yields fairly good estimates for the best of 1000, although the best-of-10 000 estimates are somewhat suspect. Typically, however, the variance of the underlying distribution is sufficiently small that the best of 10 000 is not likely to be much better than the best of 1000, and our estimates are consistent with this.

Pruning not only yields significantly faster running times for $\alpha = 1$ (almost a factor of 40 for $N = 1000$), it also produces better tours! Whatever degradation in performance the truncated neighbor lists might impose, it appears to have been more than compensated by the advantage of getting higher effective temperature lengths at the end of the annealing schedule. Higher values of α, now feasible because of the speedups provided by pruning, yield further improvements in average tour quality. Increasing α from 1 to 10 reduces the excess over the Held–Karp bound by roughly 40%. Adding low-temperature starts to neighborhood pruning appears to have no negative effect on tour quality (and indeed seems to improve it), while cutting the running time by a growing factor. The speedup factor grows with N since here the number of temperatures remains relatively unchanged as N increases, whereas it increased with N when we did not use low-temperature starts.

In what follows, we shall use SA_2 to denote the algorithm obtained from our baseline implementation SA_1 by adding neighborhood pruning and low-temperature starts. Under SA_2 the time for an $\alpha = 10$ annealing run is substantially

Table 8.10 Results for variants of simulated annealing, with temperature length set to α times the total length of the initial neighbor lists, compared to results for one or more runs of 2-Opt, 3-Opt, and Lin–Kernighan random Euclidean instances

		Average percent excess			Running time in seconds		
Variant		10^2	$10^{2.5}$	10^3	10^2	$10^{2.5}$	10^3
SA_1 (baseline annealing)	$\alpha = 1$	3.4	3.7	4.0	12.40	188.00	3 170.00
SA_1 + pruning	$\alpha = 1$	2.7	3.2	3.8	3.20	18.00	81.00
SA_1 + pruning	$\alpha = 10$	1.7	1.9	2.2	32.00	155.00	758.00
SA_2 (pruning + low temp.)	$\alpha = 10$	1.6	1.8	2.0	14.30	50.30	229.00
SA_2	$\alpha = 40$	1.3	1.5	1.7	58.00	204.00	805.00
SA_2	$\alpha = 100$	1.1	1.3	1.6	141.00	655.00	1 910.00
2-Opt		4.5	4.8	4.9	0.03	0.09	0.34
Best of 1000 2-Opts		1.9	2.8	3.6	6.60	16.20	52.00
Best of 10 000 2-Opts		1.7	2.6	3.4	66.00	161.00	517.00
3-Opt		2.5	2.5	3.1	0.04	0.11	0.41
Best of 1000 3-Opts		1.0	1.3	2.1	11.30	33.00	104.00
Best of 10 000 3-Opts		0.9	1.2	1.9	113.00	326.00	1 040.00
Lin–Kernighan		1.5	1.7	2.0	0.06	0.20	0.77
Best of 100 LKs		0.9	1.0	1.4	4.10	14.50	48.00

less than that for performing 10 000 runs of 2-Opt, and the average tour quality is better. This illustrates the oft observed phenomenon that, given enough time, simulated annealing will outperform multiple runs of the corresponding local optimization algorithm, even on a time-equalized basis. For this value of α, however, annealing remains slower and worse than 1000 runs of 3-Opt. Increasing α to 40 puts annealing ahead of 3-Opt as well (on a time-equalized basis), at least for $N = 1000$. It also produces a better average tour than Lin–Kernighan, although it is strongly dominated by the best of 100 runs of LK, which take less than one-tenth of the time required by SA_2. Even increasing α to 100 doesn't enable SA_2 to catch up.

The data in Table 8.10 nevertheless suggests that SA_2 may be gaining on Lin–Kernighan as N increases. For $\alpha = 100$ the average excess for SA_2 is roughly 0.4% below Lin–Kernighan for all values of N in the table. Meanwhile, the variance of the LK results for a given instance declines as N increases, seriously lessening the value of performing multiple runs. For sufficiently large N, one might therefore expect a crossover. To test this hypothesis, we performed comparisons on one of the 10 000-city instances in our random geometric instance test bed. For $\alpha = 100$ the running time for SA_2 was roughly 14 hours per run, with an average excess of 1.57% over five runs, all but one of which had excesses of 1.56% or less. The best excess seen in 11 000 LK runs (which collectively took 24 hours) was 1.60%. This may not be a significant difference, but it does seem to lie in favor of SA_2. Moreover, the implementation of SA_2 does not use the same sophisticated data structures for tour representation as adopted in neighbor-list LK. They might have sped up the implementation by an additional factor of 2 [Fredman et al., 1995]. In this case, one run of SA_2 would be equivalent to 3000 runs of LK, for which the expected best excess is roughly 1.64%.

For certain structured geometric instances, SA_2 can catch up to Lin–Kernighan when N is much smaller, especially since LK often is significantly slower for structured geometric instances than for random geometric instances, as observed in Section 4. Thus for the 1000-city clustered instance `dsj1000` from `TSPLIB` depicted in Figure 8.5, fewer than 500 runs of LK can be made in the time required for one $\alpha = 100$ run of SA_2 (as opposed to over 2000 LK runs for a 1000-city random geometric instance), and the average excess for the best of 500 LK runs is 1.35%, compared to a 1.27% average excess for SA_2. But note that `dsj1000` is something of an outlier in `TSPLIB`, as it was specifically generated to be a hard instance for Lin–Kernighan. Most `TSPLIB` instances do not exhibit such strongly separated clusters of cities. More typical is the situation with the printed circuit board instance `pr1002`, also depicted in Figure 8.5, where the LK slowdown is much less severe. For this instance, 1000 runs of Lin–Kernighan can be performed in the time required by a single run of SA_2 with $\alpha = 100$, and yield an expected best excess of excess of 1.57% versus excess of 1.89% for an average SA_2 run.

Nevertheless, SA_2 does seem to be capable of catching Lin–Kernighan on geometric instances as the number of cities (or the clustering) increases. For

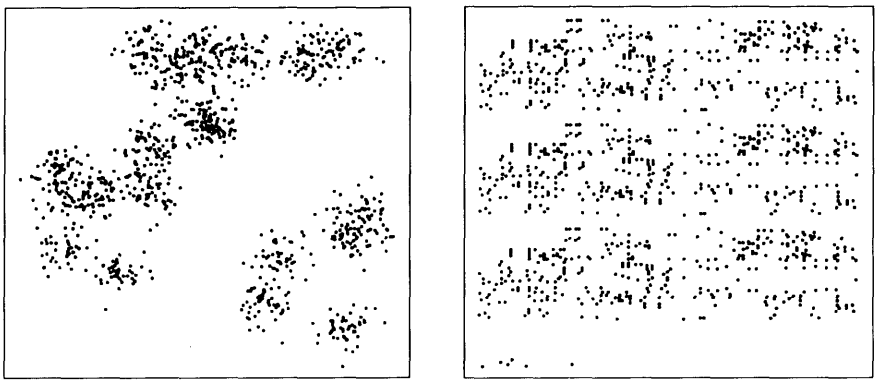

Figure 8.5 TSPLIB instances: (left) dsj1000 (right) pr1002

Table 8.11 Results for SA_2 on random distance matrices

	Average percent excess			Running time in seconds		
Algorithm	10^2	$10^{2.5}$	10^3	10^2	$10^{2.5}$	10^3
SA_2, $\alpha = 100$	12.0	37.0	99.0	101.00	447.12	2 250.0
2-Opt	40.0	51.0	70.0	0.01	0.12	1.0
Best of 1000 2-Opts	23.0	38.0	61.0	6.80	17.40	70.0
Best of 10 000 2-Opts	21.0	36.0	59.0	68.00	173.00	690.0
3-Opt	10.2	20.0	33.0	0.02	0.16	1.2
Best of 1000 3-Opts	2.8	13.0	28.0	14.00	48.00	241.0
Best of 1 0000 3-Opts	2.3	12.0	27.0	139.00	481.00	2 400.0
Lin Kernighan	1.4	2.5	3.6	0.05	0.32	2.0

random distance matrix instances, ones that in effect have no structural complexity at all, something quite different happens, as illustrated in Table 8.11. When $N = 100$ SA_2 with $\alpha = 100$ again appears to be better than 2-Opt, even on a time-equalized basis; but when $N = 1000$ it is worse on average than just a *single* run of 2-Opt, with an average excess of 99% versus 70% for 2-Opt. How can this be?

Recall from Section 3.2 that for random distance matrix instances the performance of 2-Opt depends strongly on the method chosen for generating starting tours. If the greedy algorithm is used, we have an average excess of 70% when $N = 1000$, as reported in Table 8.4. If we use nearest neighbor starts, however, like SA_2, the average excess increases to 96%. And if we start with a random tour – which the SA_2 tour may resemble after a few temperatures during which over half of all proposed moves are accepted – the average excess for 2-Opt balloons to 290%. This seems to be the reason SA_2 has so much trouble with

random distance matrices, since other experiments imply that replacing NN starts by random starts in SA_2 does not significantly affect final tour quality. If we view SA_2 simply as a method for improving random-start 2-Opt, it is stunningly successful. It reduces the average percentage excess from 290% to 99%, almost a factor of 3, while the best of 10 000 runs of random-start 2-Opt still has an average excess of 240%. Unfortunately, the true competition is greedy-start 2-Opt, and here SA_2 is at a disadvantage because the randomization inherent in its operation prevents it from exploiting a good starting tour when dealing with instances of this type. This flaw becomes more damaging as N increases, and it is fatal when $N = 1000$. (Starting at still lower temperatures, so as to preserve more of the starting tour, does not improve things much. SA_2 now behaves more and more like 2-Opt itself, and so does not get much advantage from the annealing process.)

If we consider 3-Opt and Lin–Kernighan, SA_2 is outclassed on random distance matrices even for smaller values of N and even when we ignore any running time considerations. Even allowing SA_2 a further factor of 100 in running time is not enough for it to catch Lin–Kernighan. Setting $\alpha = 10\,000$ reduces SA_2's average excess for $N = 100$ only to 3.6%, still much worse than the average of 1.4% for a single run of LK.

5.3 Other potential improvements

In the previous section we saw how two key speedup ideas sufficed to bring simulated annealing to the brink of competitiveness with Lin–Kernighan, at least for certain types of instance. In this section, we consider four additional ideas that have been proposed for pushing annealing to the front of the race: adaptive scheduling, using a permutation-based method for generating random moves, expanding the neighborhood structure to include 3-Opt moves, and exploiting parallelism.

Adaptive scheduling

Both SA_1 and SA_2 use cooling schedules in which the number of trials is the same for each temperature and temperatures are always reduced by the same factor. Although this sort of schedule was proposed by Kirkpatrick et al. [1983], their paper also suggests a more adaptive approach. According to the analogy with physical annealing that they propose, one should spend enough time to reach *equilibrium* at each temperature before cooling further. The term *equilibrium* was not defined, however, and it is doubtful that its technical meaning in terms of Markov chains was intended. Instead, Kirkpatrick et al. seemed to have something more physical in mind, emphasizing an analogy to *specific heat*, which they measured by the variance of solution values encountered at a given temperature. The higher the variance, the longer it presumably takes to reach equilibrium, so the longer one should spend at the temperature, or the slower one should lower the temperature.

Various ways have been proposed to implement such adaptive schedules. At the simplest end, Golden & Skiscim [1986] proposed performing a number of trials at each temperature that was a varying multiple of a fixed, small *epoch length L*. If at the end of a sequence of L trials the current tour length was more than a fixed bound ε away from the length at the beginning, another L trials were performed at the same temperature. Otherwise the temperature was lowered. At the more complicated end, several researchers have followed the suggestions of Kirkpatrick et al. [1983] more literally [Huang, Romeo & Sangiovanni–Vincentelli, 1986; Aarts & Van Laarhoven, 1985a; Aarts, Korst & Van Laarhoven, 1988; Lam & Delosme, 1988a; Lam, 1988]. In these implementations, detailed statistics are kept on the tour lengths encountered at the current temperature, so that a precise estimate of the variance can be computed. This estimate is then used in a complicated formula (differing from approach to approach) to determine the next temperature, thus controlling the cooling rate.

None of the published studies of these adaptive cooling schedules provide enough information for us to evaluate them fully. It appears possible that when properly tuned they might yield something like a factor of 2 speedup for a given average tour quality (by allowing us to spend less time at unproductive temperatures), although it is not clear whether this remains true when we use low-temperature starts. Nor is it clear how far the schedules that result from these adaptive processes differ from the nonadaptive schedules of SA_1 and SA_2. A detailed study that compares these approaches to each other and to nonadaptive schedules on a wide spectrum of instances would be welcome.

A simple idea that we have evaluated is the use of 'cutoffs', originally proposed by Kirkpatrick [1982]. Kirkpatrick suggested that, as soon as the number of accepted moves had reached a large figure like 10% of the planned trials for a given temperature, one might as well declare equilibrium and lower the temperature at once. This can have a significant effect on running times even for low-temperature starts, given that we have defined a 'low temperature start' to mean starting with a 50% accept rate. Based on limited testing, it appears that adding such cutoffs to SA_2 yields speedups of 10–25% without measurable effect on average tour length.

Permutation-based move generation

One way in which SA_1 and SA_2 are adaptive is in their choice of when to stop the annealing process and switch over to a final deterministic 2-Opt phase. As we have already observed, a significant portion of the overall time spent is at low temperatures while this adaptive decision is being pondered and few moves are being accepted. Theoretical results about annealing suggest that it is worthwhile to continue annealing in this situation, rather than to switch to 2-Opt, but it would still seem that the time could be more effectively used. At such low temperatures we typically have to try many moves before we find one that has a realistic chance of being accepted. In such a situation, the standard move-selection rule used by simulated annealing, random selection with replacement, seems inherently less efficient as a search heuristic than random selection *without* replacement. Even at

higher temperatures it might be desirable to have a fixed bound on how long one can go without examining a given potential move.

This issue was first addressed by Johnson et al. [1989], who discussed it in the context of graph partitioning. In that problem one chose a random move by choosing a random vertex of the graph being partitioned. Rather than simply sampling vertices without replacement, Johnson et al. considered the following alternative. At the beginning of each sequence of N trials, where N is the number of vertices in the graph, a random permutation of the vertices was constructed. Then the vertices were considered in sequence, in the order determined by the permutation, after which a new random permutation was generated. Thus one gets the advantage of sampling without replacement while enjoying a high degree of randomization. Based on limited tests, this appeared to be a good idea, although the result was surprisingly not so much a reduction in running time as an improvement in final solution value.

We can adapt this idea to the TSP as follows. At the beginning of each temperature, we construct a list of triples (i, j, a), where i is a city, j is a city on i's current neighborhood list (which itself can change from temperature to temperature when we are using dynamic pruning), and $a \in \{0, 1\}$. This triple represents the move in which i is t_2, j is t_3, and t_1 is the tour predecessor (successor) of i if $a = 0$ ($a = 1$). We then randomly permute this list and consider the corresponding moves according to their order in the permutation. When the last one has been considered, we construct a new random permutation and repeat until the total number of trials allowed for this temperature has been exhausted.

Based on limited tests of a version of SA_2 augmented to include this permutation-based randomization, the augmentation appears to offer significant advantages. We do not obtain the improved solution quality observed for graph partitioning, but we do obtain substantial reductions in running time without any noticeable deterioration of solution quality. For our 1000-city random Euclidean instances, this running time reduction ranges from 20% to 35%. Of this, perhaps half is due to more rapid termination of the annealing process (the total number of temperatures considered is reduced by about 10%). The other half comes from the fact that this way of generating moves can be significantly more efficient on a per-move basis. This arises because of the different ways in which topologically illegal moves are handled under the two schemes.

In our ordinary scheme, t_2's two tour neighbors are very likely to be on t_2's neighbor list, so there is always a chance they will be picked when we make a random choice for t_3. If this happens, we pick again and there is a chance we will pick one of them again. (To avoid getting in an endless loop of fruitless choices, we declare the trial an a priori failure if the current neighbor list for our t_2 candidate does not contain at least three elements.) Thus the average number of calls to the random number generator is more than one per move under the original scheme, whereas it is precisely one under the new permutation-based scheme (made at the time the permutation is constructed).

It would thus appear that a permutation-based randomization offers a worthwhile improvement, one that is relatively straightforward to incorporate. More-

over, permutation-based randomization combines well with the idea of cutoffs, mentioned in the previous subsection, to obtain even greater speedups. In our limited tests, the speedup of this combined approach over raw SA_2 was typically by a factor of 2 or more, all at no significant cost in terms of tour quality. These tests will be summarized in Section 5.4 (Table 8.14), where we examine a still faster alternative to standard annealing.

Although the direct advantage of these ideas is the reduction of running time, this reduction can be traded for improved tour quality. For a given total amount of running time, we can now double the temperature length, and as we have already seen, increases in temperature length are likely to produce better average tour quality.

The ideas of adaptive scheduling, cutoffs, and permutation-based randomization are all generic, in that they are not specifically tied to the TSP as an application area. Indeed, some of the proposals mentioned above originated in other domains. The second proposal we consider in this section, also due to Kirkpatrick, is more problem-specific, although it too might be relevant to other problem domains, at least by analogy.

Including 3-Opt moves

Given that 2-Opt is not nearly as good a local optimization algorithm as 3-Opt, it is natural to wonder whether we are losing something by restricting ourselves just to 2-Opt moves in our implementations of simulated annealing. With this as motivation, Kirkpatrick [1984] proposed an alternative TSP neighborhood structure based on 3-Opt moves. Since he was using unpruned neighborhoods, Kirkpatrick could not afford to consider all possible 3-Opt moves and the resulting $\Theta(N^3)$ neighborhood size. He thus restricted attention to a particular easy-to-generate subclass of 3-Opt moves similar to the subclass used in Or-Opt algorithm [Or, 1976] described in Section 3.5.

Or-Opt restricts its attention to 3-Opt moves in which one of the three segments into which the tour is initially broken is of length 3 or less. Kirkpatrick expanded this to include moves in which one of the segments is of length 10 or less. One can randomly pick this segment (for which there are $10N$ possibilities) and then randomly decide how to reinsert it into the tour (for which there are roughly $2N$ possibilities, depending on the neighboring cities between which the segment is inserted and on whether it is inserted in its original order or is reversed). This yields a total of roughly $20N^2$ possibilities, larger than the unpruned 2-Opt neighborhood size that Kirkpatrick used, but only by a constant factor. For 100-city random geometric instances, using this restricted 3-Opt neighborhood and a factor 8.5 increase in running time, he reports obtaining a 1.2% improvement in average tour length over 2-Opt annealing.

It is not clear, however, whether this was the best way to spend the extra time. As we have observed, increasing the running time for ordinary 2-Opt annealing (without changing the neighborhood structure) will typically by itself lower the average percentage excess. To find the value of 3-Opt moves for simulated

annealing more fully, we augmented our implementation SA_2 to include 3-Opt moves. The resulting implementation, which we shall denote by SA_3, was then extensively tested.

Given what we had learned from our experience with normal 2-Opt annealing and with 2-Opt and 3-Opt themselves, SA_3 does not take precisely the same approach as Kirkpatrick. First, rather than replacing the 2-Opt neighborhood structure with one based solely on 3-Opt moves, we use a neighborhood structure that includes both sorts of moves (just as the neighbor-list implementation of 3-Opt itself looks for 2-Opt as well as 3-Opt moves). We do this by spending half our time at each temperature using a 3-Opt neighborhood and half using our standard pruned 2-Opt neighborhood. Second, we don't require that our 3-Opt moves have a short segment, but instead prune the 3-Opt neighborhood structure analogously to the way in which we pruned the 2-Opt neighborhood structure. To be specific, we use our bounded-length neighbor lists to restrict the possibilities for both t_5 and t_3.

Here is a high-level description of how a 'random' 3-Opt move is chosen. We first randomly choose t_1 and one of its tour neighbors t_2. Next t_3 is randomly chosen from the current (pruned) neighbor list for t_2. We then randomly choose t_4 (both neighbors of t_3 are now possible, whereas under 2-Opt the choice was forced), and randomly choose t_5 from the current neighbor list for t_4. (Note that, for topological reasons, some members of the neighbor list may be ineligible to serve as t_5. In the search for a topologically legal candidate, we allow ourselves five random choices before giving up.) Finally, there will be either one or two possibilities for t_6, and in the latter case we choose randomly. For initial neighbor lists of length 20, this yields a total of at most $3200N$ neighbors and typically far fewer because of topological considerations.

Table 8.12 summarizes some of the results obtained for SA_3 on our standard test beds. For these results we used pruned neighborhoods based on neighbor lists of length 20 and performed runs with $\alpha 20N$ moves per temperature. For the values of α tested (40 and 100) this means that only a small fraction of the 3-Opt moves that survive our pruning will actually be investigated. Permutation-based move generation would thus have been more expensive than the standard move-generation method, so we used the standard method. We did not use cutoffs for the results reported here, although experiments indicate that, as with SA_2, they would yield a 15–30% speedup without significant loss of tour quality. For simplicity, we spend the first half of each temperature ($\alpha 10N$ moves) using the 2-Opt neighborhood and the second half using the 3-Opt neighborhood. This appears to yield similar results to those we would obtain by alternating between neighborhoods or randomly choosing a neighborhood at each move.

Table 8.12 shows that the replacement of half of our 2-Opt moves at each temperature by 3-Opt moves does seem to yield significantly improved tour lengths. Moreover, although the replacement also increases running time, this appears to be a more effective way to use the extra time than simply to perform 2-Opt annealing with a slightly larger value of α. Indeed, for random Euclidean instances with $N > 100$, SA_3 with $\alpha = 40$ is both better and faster than SA_2 with $\alpha = 100$. For the one 10 000-city instance we tested, we did not perform enough

Table 8.12 Results for simulated annealing with and without 3-Opt moves

	Average percent excess			Running time in seconds		
Algorithm	10^2	$10^{2.5}$	10^3	10^2	$10^{2.5}$	10^3
SA_2 $\alpha = 40$	1.3	1.5	1.7	58	204	805
SA_3 $\alpha = 40$	1.2	1.2	1.5	81	269	1 150
SA_2 $\alpha = 100$	1.1	1.3	1.6	141	655	1 910
SA_3 $\alpha = 100$	1.1	1.2	1.3	206	674	2 700
Best of 1000 LKs	0.9	0.9	1.3	40	144	478
	Random distance matrices					
SA_2 $\alpha = 100$	12.0	37.0	99.0	101	447	2 250
SA_3 $\alpha = 40$	5.0	8.0	23.0	61	294	1 520
SA_3 $\alpha = 100$	4.0	7.0	18.0	147	881	3 670
One LK	1.4	2.5	3.5	0	0	2

runs to determine which produced better tours, both algorithms producing excesses in the range 1.5–1.6%, but some relative speed information is as follows: SA_2 with $\alpha = 100$ took 14 hours (6.5 with cutoffs and permutation-based move generation) and SA_3 with $\alpha = 40$ took 8 hours (5.5 with cutoffs).

The advantage of SA_3 with $\alpha = 40$ over SA_2 with $\alpha = 100$ is even more dramatic for random distance matrices, e.g., cutting the average percentage excess by over a factor of 4 when $N = 1000$. For real-world instances from `TSPLIB`, the advantage of SA_3 typically resembles than for random Euclidean instances, although results for individual instances may vary. For instance, on `dsj1000` no improvement is seen, whereas on pr1002 using SA_3 with $\alpha = 100$ drops the average excess (over 20 runs) from 1.89% to 1.70%. Running times increase by 10–20%.

In summary the improvements obtainable by incorporating 3-Opt moves into the neighborhood structure are significant but, as comparisons to Lin–Kernighan indicate, unlikely to make simulated annealing competitive on a time-equalized basis with LK, except in those situations where SA_2 was already fairly close.

Exploiting parallelism

In Section 3.4 we discussed two basic schemes for exploiting parallelism in local search for the TSP, instance partitioning and parallel move generation. The discussion of the best-of-k-runs approach in this section suggests a third scheme, performing multiple runs in parallel. In the annealing domain, none of these parallelization schemes has been studied extensively. The two-path tour-partitioning scheme of Allwright & Carpenter [1989], mentioned in Section 3.4, was

originally proposed in the context of annealing, but the authors did not present any significant computational results. The easiest to implement, however, would be the multiple parallel-run approach.

Simulated annealing, like the other algorithms we have discussed in this chapter, produces tour lengths with a high variance, especially for small values of N. Thus a best-of-k-runs approach might be as effective for simulated annealing as we have seen it to be for 2-Opt, 3-Opt, and Lin–Kernighan. This may be true even if we don't have parallel processors. It might well be that performing k runs with $\alpha = x$ and taking the best solution found is a better approach than performing one run with $\alpha = kx$.

Some idea of the relative merits of the two approaches can be gleaned from Table 8.13, where we have computed estimates for the best of one, two and four runs for the random Euclidean instances in our test bed and selected values of α under both SA_2 and SA_3. These estimates are based on the same runs used in computing the averages given in earlier tables, and much of this data is based on only five runs for the given α and neighborhood structure. Thus we cannot accurately estimate results for larger values of k, and even the values for $k = 4$ are suspect, although they can still help to reveal trends. We give the data to two decimal places of accuracy mainly to prevent the obliteration by rounding effects of possibly significant

Table 8.13 Best-of-k-runs for SA_2 and SA_3

			Average percent excess			Running time in seconds		
Algorithm		k	10^2	$10^{2.5}$	10^3	10^2	$10^{2.5}$	10^3
			Random Euclidean matrices					
SA_2	$\alpha = 10$	1	1.55	1.80	2.18	14	50	184
SA_2	$\alpha = 10$	2	1.31	1.54	2.07	29	100	372
SA_2	$\alpha = 10$	4	1.15	1.33	1.98	57	201	747
SA_2	$\alpha = 40$	1	1.28	1.51	1.66	58	201	807
SA_2	$\alpha = 40$	2	1.10	1.35	1.56	116	405	1 610
SA_2	$\alpha = 40$	4	1.00	1.23	1.50	232	814	3 220
SA_2	$\alpha = 100$	1	1.13	1.32	1.61	144	655	1 930
SA_2	$\alpha = 100$	2	1.02	1.18	1.53	285	1 310	3 840
SA_2	$\alpha = 100$	4	0.96	1.08	1.46	566	2 620	7 660
SA_3	$\alpha = 40$	1	1.19	1.24	1.48	81	271	1 090
SA_3	$\alpha = 40$	2	1.04	1.10	1.37	162	539	2 250
SA_3	$\alpha = 40$	4	0.96	1.00	1.28	323	1 080	4 570
SA_3	$\alpha = 100$	1	1.08	1.21	1.32	204	666	2 700
SA_3	$\alpha = 100$	2	0.97	1.07	1.25	410	1 340	5 400
SA_3	$\alpha = 100$	4	0.93	0.98	1.18	822	2 700	10 800
Best of 100 LKs			0.91	1.00	1.41	4	14	48
Best of 1 000 LKs			0.91	0.89	1.29	41	144	478
Best of 10 000 LKs			0.91	0.82	1,23	406	1 440	4 780

differences of 0.1%. Times given are those for running the k iterations in sequence. To evaluate parallel performance, simply use the $k = 1$ time for all values of k. (In this simpleminded scheme there is essentially no parallel overhead, assuming each processor has enough memory to run the algorithm). The $k = 1$ running times in some cases differ slightly from those reported in earlier tables, as we are here reporting the median rather than the mean.

Multiple runs lead to significantly better tours under both algorithms, although the amount of the gain declines as average tour quality improves and as N increases. The first effect is most marked with the 100-city random Euclidean instances, where improvement obtainable by taking the best of four runs declines from 0.40% to 0.15% as one goes from the weakest combination of algorithm and parameter setting in the table (SA_2 with $\alpha = 10$) to the strongest (SA_3 with $\alpha = 100$). One reason for this is that the room available for improvement shrinks in the latter case to just 0.18%. The best possible average excess is the 0.91% obtained by the best of 100 LK runs. (For random Euclidean instances this small, LK will typically find an optimal solution if given 100 tries.) The fact that the value of multiple runs also declines as N increases is consistent with the already observed fact that, for most algorithms, the variance of the tour lengths produced declines as N grows. Indeed, if we go out as far as $N = 10\,000$, the improvement obtainable by taking the best of four runs for SA_3 with $\alpha = 40$ drops to just 0.05% from the 0.20% figure in the table for $N = 1000$.

The reported results also suggest that, if one plans to perform the multiple runs in parallel, the incremental reduction in average tour length obtained by adding the kth processor declines as k increases. Typically there is a bigger reduction in average tour length for going from one to two processors than for going from two to four, suggesting that even the value of doubling the number of processors may decline as k increases. To get a better feel for this effect, we increased the number of runs of SA_3 with $\alpha = 40$ on our test-bed instances with $N = 316$ from 5 to 25, so that plausible estimates for the expected best of up to 16 runs could be obtained. For k increasing by factors of 2 from 1 to 16, these new estimates for the expected best excess are 1.27%, 1.11%, 1.00%, 0.92%, and 0.86%; each successive improvement is smaller than its predecessor.

As to whether performing multiple runs on a single processor is better than increasing α, the evidence from Table 8.13 is mixed. For $N < 1000$ and SA_2, four runs at $\alpha = 10$ seems to be a significantly better use of time than one run at $\alpha = 40$, and two runs at $\alpha = 40$ seem to be at least comparable to one at $\alpha = 100$. Similarly, for SA_3, two runs at $\alpha = 40$ are possibly slightly better than one run at $\alpha = 100$. For $N = 1000$, however, only one of these comparisons comes out in favor of multiple runs, and indeed one would expect the best-of-k-runs approach to lose out as N increases further, again because of the corresponding decline in variance.

Finally, for $N < 1000$ the best-of-k-runs approach does not come close to making annealing competitive with LK on a time-equalized basis (assuming the two approaches are allowed to use the same number of processors). For $N = 1000$, however, using multiple runs of SA_3 with $\alpha = 100$ can bring it into parity with LK. The best of two annealing runs is essentially just as good as the best of 10 000 LK

runs, and it can be obtained in essentially the same time. Results for random distance matrices are similar, although no parity with LK seems possible, and the advantage of larger α's over multiple runs seems to hold even for $N = 316$.

5.4 Threshold-based variants on annealing

We conclude our treatment of simulated annealing and the TSP by considering several variants that dispense with the coin flipping used to decide whether moves are accepted; instead they simply test whether the resulting tour length is less than a precomputed threshold. These are variants on the *threshold algorithms* described in Chapter 4.

Threshold Accepting

The threshold algorithm that is most like standard simulated annealing is the *Threshold Accepting* algorithm of Dueck & Scheuer [1990]. In this algorithm the temperature schedule of annealing is replaced by a threshold schedule, and a proposed move is accepted so long as it does not increase the current tour length by more than the current threshold. Assuming a threshold schedule and a temperature schedule have both yielded the same total number of trials, the threshold schedule will presumably have a running time advantage in that it will save one call to a random number generator and one exponentiation per trial. But note that our baseline annealing implementation and its augmentations already save most of the time needed for exponentiation by using a lookup table, so the running time savings might not be major. Moreover, one would need to have a schedule that not only yielded the same number of trials (or fewer) but also the same number of accepted moves (or fewer), and there might be a tour-quality penalty for using thresholds under such restrictions.

Using pruned neighbor lists, Dueck & Scheuer claim they are able to get better solutions than simulated annealing for a given number of trials, based on a comparison of their results for instance `pcb442` from `TSPLIB` to the simulated annealing results of Rossier, Troyon & Liebling [1986]. Rossier, Troyon & Liebling use neighborhood pruning but it is the nonrobust geometric version of Bonomi & Lutton [1984] rather than pruning based on nearest neighbors, and it is not clear whether low-temperature starts are used. If we compare Dueck & Scheuer's results to those for SA_2, their advantage is less clear. Their schedules were designed to produce a target number of trials, and their best result, an average tour length of 51.36, was obtained using 4 000 000 trials. If one sets $\alpha = 8$ in our implementation, the average tour length and number of trials over 20 runs are 51.33 and 4 020 000, pretty much a dead heat.

It is dangerous to generalize from just one instance, however. To expand the range of comparison, we thus modified SA_2 to do Threshold Accepting and we tested it on the 1000-city random Euclidean instances in our test bed. Dueck & Scheuer do not say much about how to choose an appropriate threshold schedule for a given instance class, but a natural choice would be one that minics a temperature

schedule in standard annealing: for some appropriately chosen $\beta > 0$ we simply convert each temperature T into a threshold $\theta = \beta T$. For our 1000-city random geometric cities and $\alpha = 10$, taking $\beta = 1.0$ yields the same initial acceptance ratio as standard annealing (given Greedy starts for both). The acceptance ratio for Threshold Accepting drops more rapidly, however, so there are 25% fewer acceptances per temperature or threshold as well as 8% fewer temperatures or thresholds (because of earlier termination). The overall running time savings for thresholds is about 18%, due mostly to this reduced number of trials and acceptances rather than to saved exponentiation time. And, unfortunately, the average excess increases from 2.02% to 2.68%. Choosing $\beta = 1.5$ equalizes the running times, but does not improve the average tour quality. This suggests that, barring the existence of much more effective schedules, Threshold Accepting offers no significant advantage over standard annealing and may indeed make thing worse.

Deterministic Threshold Accepting

Threshold acceptance removes randomization from the process of move acceptance. A second algorithm proposed by Dueck & Scheuer [1990] removes it from the process of move generation as well. In their *Deterministic Threshold Accepting* algorithm (DTA), they simply cycle through all possibilities in a fixed order, once per threshold. Like the permutation-based move-generation process discussed above, this insures that no move will be overlooked because of inopportune random choices, and it saves the many calls to the random number generator that ordinary Threshold Accepting made to generate candidate moves. Dueck & Scheuer propose to combine this approach with pruned neighbor lists of length 20 or less and to go through the move set once per threshold. According to Dueck [1993], the best ordering is to have the choice of candidate for t_2 made in an inner loop, with the location for t_3 on t_2's neighbor list already determined by an outer loop that runs over all possible locations, from farthest to nearest. City t_1 is always taken to be the tour successor of t_2. For a fixed number of thresholds (Dueck & Scheuer suggest 100), this yields relatively rapid running times, i.e., close to those we would obtain using SA_2 with $\alpha = 1$.

Unfortunately, it also seems to yield comparable tours, i.e., significantly worse than a *single* run of Lin–Kernighan can provide in substantially less time. (Recall that we needed values of α larger than 10 to get better average tours than LK.) Data reported by Dueck [1993] for the test instances `pcb442` and `att532` from `TSPLIB` suggests that the best of 100 runs (from random starting tours) is often worse than an average LK run for both instances. One can get better results by using more closely spaced thresholds, and we have tested this in our own implementation of DTA. However, we were unable to find a threshold schedule that yields results as good as SA_2 with $\alpha = 10$, even when as much time was used. Indeed, we could not get DTA to perform even as well as the original Threshold Accepting algorithm, which for our 1000-city random Euclidean instances produced tours averaging some 0.6% worse than those for SA_2. The gap between deterministic and standard Threshold Accepting could be partially closed by adding some randomization

to the deterministic algorithm. For instance, if we randomly permuted the cities each time the threshold was changed, we typically reduced the average excess by 0.2–0.4% at no significant running time penalty. Nevertheless, the basic approach cannot yet be called promising.

Water level algorithms

Dueck [1993] also proposed two other variants on simulated annealing, the *Great Deluge Algorithm* and *Record-to-Record Travel*, both of which dispense with the need for a schedule as well as the need for randomized move acceptance. The analogy motivating both is best understood if we temporarily assume we are dealing with a maximization problem, not a minimization problem like the TSP. We assume the search space is a hilly terrain, currently being drenched by a continuing downpour of rain. As the rain continues to fall, the water level rises, gradually isolating the higher land elevations as islands in a surrounding sea. Our current solution randomly moves around the countryside, subject only to the constraint that it does not step in the water. Eventually the rising water will force it to the top of some hill or mountain, and this is the local optimum to which the process converges.

In the Great Deluge Algorithm (GDA) we link the rises of the water level to the acceptance of moves. Each time a move is accepted (does not lead underwater), the water level rises by some fixed increment Δ. (The initial water level might simply be the value of the initial random solution.) In Record-to-Record Travel (RTR) the water level rises each time a new champion solution is found, with the level maintained at some fixed distance δ below the current best solution value. Both these approaches were adapted to the TSP under the 2-Opt neighborhood structure by Dueck [1993], concentrating on deterministic variants in which move generation was performed as in Deterministic Threshold Accepting. (For a mimimization problem, such as the TSP, the water level must now decrease rather than increase, so perhaps we should view ourselves as fish in the time of a drought.) RTR and GDA produced results comparable to DTA and thus were not impressive.

Dueck's [1993] tests were again limited to just two instances, however, and to settings of Δ and δ that led to relatively fast running times. Based on our own experiments, allowing additional running time and limited randomization in the move-generation process, as described above for DTA, makes the situation much more competitive, at least in the case of GDA. In our implementation we adapt an idea of Dueck and let Δ be the maximum of a fixed fraction a of the current tour length and a fixed fraction b of the distance between the current tour length and the water level. In an attempt to adapt the algorithm automatically to instances of different sizes, we allow a and b to vary with N, setting $a = 1/(r \times 10\,000N)$ and $b = 1/(r \times N)$ where r is an additional parameter we can use to adjust the running time, similar to α in simulated annealing. Larger values of r yield smaller steps sizes and hence longer running times (and presumably better tours). In analogy with the low-temperature starts of SA_2, we generate our starting tours using Nearest Neighbor and set the initial water level to be twice the initial tour length. Dueck's results suggest that we can safely restrict ourselves to neighbor lists of length 10,

Table 8.14 Comparison of simulated annealing and the Great Deluge Algorithm

		Average percent excess			Running time in seconds		
Instance	α, r	SA_2	SA_2^+	GDA	SA_2	SA_2^+	GDA
				Random Euclidean instances			
$N = 316$	10, 4	1.80	1.83	1.81	50	23	17
	40, 16	1.52	1.57	1.61	232	97	65
$N = 1000$	10, 2	2.02	2.14	1.91	228	98	78
	40, 8	1.66	1.68	1.70	805	357	283
$N = 10{,}000$	40, 2	1.77	1.79	1.64	25 300	10 200	6 200
	100, 5	1.57	1.53	1.51	49 700	23 600	16 200
				TSPLIB instances			
pcb442	10, 2	1.60	1.74	1.57	62	39	17
	40, 8	1.38	1.43	1.20	281	109	47
att532	10, 2	2.12	2.18	2.23	77	38	16
	40, 8	1.84	1.83	1.84	303	151	62
dsj1000	10, 2	1.83	2.13	1.96	216	90	69
	40, 8	1.61	1.68	1.49	907	434	276
pr1002	10, 2	2.33	2.25	2.08	198	87	80
	40, 8	1.90	1.90	1.89	784	327	339
				Random distance matrices			
$N = 316$	40, 2	46	44	42	182	107	21
	100, 6	37	37	39	447	264	62

something one cannot do for normal annealing without significantly degrading the output tours, and we do this. Our first observation is that for such lists, allowing GDA to permute the cities randomly before each pass through the set of moves, as described above, typically yields an improvement in average tour length of some 0.3% over that obtained with purely deterministic move generation.

As to comparisons with simulated annealing, Table 8.14 presents results for selected instances and instance classes (the results for 10 000-city random Euclidean instances were based on the single instance we have been tracking throughout this section). These results suggest that GDA is well matched against SA_2, even if we allow SA_2 to use permutation-based move generation and cutoffs, a variant we denote by SA_2^+. For each instance and instance class covered in Table 8.14 and for each choice of α, we were able to find a value of r such that the tours found by GDA were roughly as good as the tours found by SA_2 and SA_2^+ (themselves comparable with the SA_2 tours slightly better). Moreover, GDA usually produced its tours in significantly less time than SA_2^+, which itself was roughly two times faster than SA_2. Note also that for each instance and instance class, increasing r (and hence running

time) yields an improvement in GDA's tours roughly equivalent to the improvement obtained with SA_2 when α is increased by the same factor.

It thus appears that the Great Deluge Algorithm is well worth further study. The Record-to-Record Travel algorithm may also perform well, although intuitively this seems somewhat less likely, and we have not tested it ourselves.

Multicanonical Annealing

Multicanonical Annealing, our final variant, also appears to have advantages over the standard version. It has been successfully applied to random Euclidean instances with as many as 40 000 cities. This algorithm, due to Lee & Choi [1994], has a structure rather like record-to-record travel (RTR) in the previous subsection; we maintain a threshold that is typically a fixed distance δ above the best tour length seen so far, and we accept no move that produces a tour longer than this threshold. However, this is just the beginning, as the approach also incorporates a variant on the 'rejectionless annealing' idea of Greene & Supowit [1986].

As with RTR, we use the pruned 2-Opt neighborhood; specifically, we consider only the 20 nearest neighbors of a chosen t_2 as candidates for t_3. Now, however, instead of just picking a random such neighbor as t_3, or considering each in turn, we consider all of them in parallel. Given our choice for t_2, we let t_1 be its tour successor and compute for each t_3 candidate the length of the tour obtained by performing the induced 2-Opt move. Each possible tour length has a weight associated with it, the weight being 0 if the tour length exceeds the threshold [Lee, 1995]. The successor of the current tour is then chosen randomly from this set of derived tours augmented by the current tour, with probabilities proportional to their weights. The choice of one from many is the essence of rejectionless annealing, but in Greene & Supowit [1986] the current tour is not included in the options, and the weights are computed differently and are temperature dependent.

Here the weight function is designed so as to increase the probability of acceptance of the more rarely seen solution values (suitably discretized). According to the theory behind this approach [e.g., Lee, 1993], the appropriate choice for the weight of the (rounded) tour length L is $1/\Omega(L)$, where $\Omega(L)$ is the probability density for length L. If we could assume everything were independent, this would make each possible value of L that is less than the threshold equally likely to occur as the next tour length. Assuming that the more extreme tour lengths are the rarer ones (and that extremely bad lengths are ruled out by the threshold constraint), this scheme will cause the better of two moves to be chosen with the higher probability, as in ordinary simulated annealing. This new approach might conceivably be faster, however, because the 'rejectionless' way in which the choices are made is potentially less biased toward retaining the current tour.

Of course, we do not typically know $\Omega(L)$, so the algorithm can only attempt to approximate it. The approximation is based on a continually updated histogram of the lengths of the chosen tours, with log-linear extrapolations made for tour lengths shorter than any previously seen [Lee & Choi, 1994]. To facilitate the maintenance of histograms, the threshold is only lowered after every MN moves (typically

$M = 25$), and at that time it is set to the current tour length or δ plus the best tour length seen so far, whichever is greater. Here δ is fixed for a given instance size but shrinks as a proportion of tour length as N increases.

The results reported by Lee & Choi [1994] suggest that, at least for smaller values of N, their implementation of Multicanonical Annealing attains tour quality comparable to SA_2 with $\alpha = 40$, interpreting the reported data in light of the estimates for expected Held–Karp bounds by Johnson, McGeoch & Rothberg [1996]. For $N = 10\,000$, however, the reported tour quality for Multicanonical Annealing implies a 2.0% excess, compared to 1.7% for 2-Opt Annealing. Lee & Choi do not generally report running times, so it is difficult to evaluate how the multicanonical approach competes on that basis. The one time they do report is 7 CPU-hours on an IBM 340 h workstation for 10 000-city instances; compare this to the 6 hours for 2-Opt Annealing on the same instance using the presumably faster SGI Challenge. This suggests that, on the same machine, 2-Opt Annealing's 1.7% excess might take significantly longer to obtain than Multicanonical Annealing's 2.0%, leaving open the question as to which approach would be better once running times have been equalized – a question which joins the many we have already left open.

Clearly there are many possible directions that might lead to improved performance for annealing or a close variant, ideas which might be applicable in other domains. As we warned at the beginning of this section, these questions are mostly academic in the case of the TSP. We have already seen how the much faster Lin–Kernighan algorithm dominates annealing when running times are limited. In the next section we shall encounter approaches that, in the context of the TSP, are much more effective than simulated annealing at using any extra time that might be available.

6 GENETIC ALGORITHMS AND ITERATED LIN–KERNIGHAN

The use of genetic algorithm as an approach to optimization can be traced back at least to the 1970s. See Goldberg [1989] and Chapter 6 for some of the history. The best adaptations of this approach to the TSP follow the basic schema presented in Figure 8.6, where each performance of the loop consisting of Steps 3.1 through 3.5 can be viewed as the processing of a single *generation* in the evolutionary process. Note that the operations on different solutions can be performed in parallel if desired, so this is sometimes called the parallel genetic algorithm. As with the schema for simulated annealing in Section 5, the schema leaves several operations and definitions unspecified. A specific adaptation of the schema to the TSP needs to specify k and k', the methods for generating starting solutions (tours), the local optimization algorithm **A**, the mating strategy, the nature of the crossover and mutation operators, the selection strategy, and the criterion for convergence.

The schema in Figure 8.6 is not what was meant by a 'genetic' algorithm in such early references as Holland [1975]. In particular, the application of local optimization to the individual solutions in Steps 2 and 3.4 could be viewed as an almost

1. Generate a *population* of k starting solutions $\boldsymbol{S} = \{S_1, \ldots, S_k\}$
2. Apply a given local optimization algorithm **A** to each solution S in $\boldsymbol{S}$, letting the resulting locally optimal solution replace S in $\boldsymbol{S}$.
3. While not yet *converged* do the following:
   - 3.1 Select k' distinct subsets of $\boldsymbol{S}$ of size 1 or 2 as *parents* (the *mating* strategy).
   - 3.2 For each 1-element subset, perform a randomized *mutation* operation to obtain a new solution.
   - 3.3 For each 2-element subset, perform a (possibly randomized) *crossover* operation to obtain a new solution that reflects aspects of both parents.
   - 3.4 Apply local optimization algorithm **A** to each of the k' solutions produced in Step 3.3, and let $\boldsymbol{S}'$ be the set of resulting solutions.
   - 3.5 Using a *selection strategy*, choose k *survivors* from $\boldsymbol{S} \cup \boldsymbol{S}'$, and replace the contents of $\boldsymbol{S}$ by these survivors.
4. Return the best solution in $\boldsymbol{S}$.

Figure 8.6 General schema for a genetic optimization algorithm

heretical addition: In the context of the original biological motivation for the genetic approach, it embodies the discredited Lamarckian principle that learned traits can be inherited. Nonetheless, such local optimization steps appear to be essential if competitive results are to be obtained for the TSP, and we shall restrict our attention in what follows to genetic algorithms that use them. This is not to say that the non-Lamarckian approach has been abandoned by researchers. For an interesting recent example in which the genes correspond to schema for geometrically partitioning the cities, see Valenzuela & Jones [1994].

Even without the local optimization steps of Figure 8.6, a genetic algorithm can properly be classified as a variant on local search. Implicit is a neighborhood structure in which the neighbors of a solution are those solutions that can be reached by a single mutation or mating operation. With the local optimization steps, the schema can also be viewed simply as a variant on an algorithmic approach we discussed extensively in the previous section the best-of-k-runs approach to local optimization. Here, instead of independent random starts, we use the genetically motivated operations to construct what we hope will be better starting tours, tours that incorporate knowledge we have obtained from previous runs. The latter way of viewing the tour turns out to be more productive.

We begin in Section 6.1 by covering the history of the schema in Figure 8.6 and its most influential adaptations. Section 6.2 then presents results on our standard test bed for the most cost-effective adaptation to date, the *Iterated Lin–Kernighan* algorithm. Section 6.3 samples some recent adaptations that take the basic schema in more complicated directions.

6.1 Filling in the schema

Brady's algorithm and its successors

The first TSP algorithm embodying the schema of Figure 8.6 appears to be that of Brady [1985]. Brady used 2-Opt for local optimization and restricted himself to matings (i.e., all offspring had two parents). For his mating strategy, he identified the parents by doing a random matching of the elements of **S** (thus implying $k' = k/2$). In this crossover operation he examined the parent tours until he found a pair of common subpaths, one in each parent tour, that contained precisely the same set of cities, but in a different order. The child was then obtained by removing the longer of the two subpaths from the tour that contained it and replacing it by its shorter counterpart.

In his experiments with this approach, Brady first observed that increasing k could yield improved results. For a 64-city random Euclidean instance, he obtained significantly better results in a given (sequential) amount of time with $k = 12$ than with $k = 2$. The $k = 12$ results were also marginally better than he obtained by spending the same amount of time performing multiple-run 2-Opt from random starting tours. This advantage may, however, have been due to the fact that, when started from the fairly good tours produced by the mating operation, his implementation of 2-Opt was two or three times faster than when it was started from a random tour. Even with the overhead of the mating procedure, Brady's genetic algorithm could perform twice as many 2-Opt runs in a given amount of time. This same sort of speedup could have been obtained simply by using the randomized Greedy algorithm to generate the starting tours for 2-Opt. And as observed in Section 3.2, Greedy starts yield significantly better final tours than totally random starts.

Brady's results nevertheless suggested that his version of the genetic approach had potential, as there were obvious ways in which it could be improved. First, the particular crossover operation he chose was severely limited. For instances with 64 cities he was usually able to find common subpaths of nontrivial length (8–32 cities), but this would clearly become less likely if N were to increase. For instances of 100–200 cities Suh & Van Gucht [1987a, 1987b] replaced Brady's crossover with what they called the *heuristic* crossover from Grefenstette et al. [1985]. Here one constructs the child of two parents by a variant of Nearest Neighbor, restricted to the edges in the union of the two parents. The construction works roughly as follows. We begin by picking a random starting city. Inductively, if city c is currently the last city in the current partial tour, the next city is the closest unvisited city that is a tour neighbor of c in one of the parent tours. If no such city exists, a random unvisited city is chosen. Suh & Van Gucht combined this crossover with a truncated version of 2-Opt (one that performed a limited number of moves and did not guarantee local optimality). With $k = 100$ they appear to have found tours comparable to those reported for 3-Opt in Section 3.2, although their studies were restricted to a few instances, all with $N \leqslant 200$. Moreover, their running times, even for their parallel implementation [Suh &

Van Gucht, 1987b], appear to have been far in excess of the 0.1 second or less required by neighbor-list 3-Opt for instances this small.

The Mühlenbein et al. algorithm

A major leap in genetic algorithm performance came with the work of Mühlenbein, Gorges-Schleuter & Krämer [1988]. Their algorithm was designed from the start for parallel implementation, and it introduced a more sophisticated mating strategy together with what appears to be an even better crossover. One tour was assigned to each processor, and every time through the loop each processor would mate its tour (the receiver) with some other processor's tour the (the donor). The donor was chosen randomly from a collection of tours consisting of the four tours on neighboring processors (their initial parallel machine used a grid to interconnect its processors) together with the best tour overall (if not a neighbor), with a bias imposed in favor of the shorter tours. The crossover operation itself proceeded by randomly choosing a subpath of length between 10 and $N/2$ from the donor and then extending it to a full tour by adding successive cities much as in the *heuristic* crossover. The extension was biased toward edges in the receiver rather than to shorter edges; it worked as follows. Let city c be the last city in the current path. If c has a tour neighbor in the receiver tour that has not yet been visited, add such a city to the end of the path. Otherwise if c has an unvisited tour neighbor in the donor tour, add such a city. If neither possibility holds, add the first as-yet-unvisited city in the receiver tour.

Mühlenbein et al. also introduced an idea for speeding up local optimization on offspring. For each offspring tour they identified all the subpaths of four or more cities that were present in both parents and *locked* all the edges joining the cities in such paths, forbidding any 2-Opt move that broke such an edge. As described by Mühlenbein et al. [1988], their local optimization algorithm performed Or-Opt as well as 2-Opt moves and ran until full local optimality was attained, rather than truncating the search as did Suh & Van Gucht. For their most impressive results, however, they ended up dropping the Or-Opt moves and simply used a truncated 2-Opt, sped up using the above locking trick. This allowed them to process more generations in a given amount of time, which apparently made up for the loss of local optimization firepower.

The results reported by Mühlenbein et al. [1988] were for a 16-processor Encore system, and the biggest problem considered was an early version of the `TSPLIB` instance `pcb442` (in which distances were computed to a precision one decimal place fewer than currently obtainable). For this instance, using full 2-Opting and Or-Opting, the best solution length found was 1% shorter than an average 3-Opt tour and only 0.1% longer than an average Lin–Kernighan tour. Follow-up experiments using their fast truncated 2-Opt and a network of 64 transputers improved significantly on this. Here they concentrated on `TSPLIB` instance `att532` and obtained an average excess over optimal of 0.19% [Gorges-Schleuter, 1991], which is significantly better than the average of 0.94% for Lin–Kernighan. Even better results were reported by Mühlenbein & Kindermann

[1988], but they were based on a corrupted copy of the instance [Mühlenbein, 1995].

Each run took 3 hours on the 64-processor network, compared with 2 seconds for an LK run on our SGI Challenge. The transputers in the network were early models, however, and probably not even 1/64 as fast as the SGI Challenge. Thus the times for Mühlenbein et al. probably translate to between 1 and 3 sequential hours on the SGI Challenge. This isn't quite enough to make the algorithm competitive with Lin–Kernighan, for which 500 independent runs take 10 minutes and yield an expected best excess over optimal of 0.15%. It is, however, in the same ballpark as can be accomplished with our best simulated annealing variant. Four runs of the annealing algorithm SA_3 in Section 5 with cutoffs and $\alpha = 100$ together take roughly an hour, with an expected best excess of 0.18% based on a suite of 100 such runs.

The results of Mühlenbein et al. suggest that the genetic approach may be at least as serious a competitor for multiple-run Lin–Kernighan as is simulated annealing. Subsequent experiments with variants of the Muhlenbein et al. algorithm yielded more detailed information. Gorges-Schleuter [1989] confirmed Brady's observation that larger population sizes lead to better results in a given amount of sequential time, at least up to her 64-processor limit; see also Mühlenbein & Kindermann [1989]. She also performed detailed studies of various mating and selection strategies, showing that they could significantly influence the quality of the final tours. More important for what follows, however, is the question of how the choice of local optimization algorithm affects the results.

Jog, Suh & Van Gucht [1990] showed that, at least for their 70-processor implementation, the less 2-Opt was truncated, the better were the solutions produced (albeit at a cost of increased running time). Braun [1990] went back to the original Mühlenbein et al. idea of using full 2-Opt and Or-Opt and made it more practical by using neighbor-list implementations, apparently not investigated by earlier researchers. This allowed him to have the benefits of more powerful local optimization without compromising the number of generations he could perform in a reasonable length of time. For the 431-city instance in `TSPLIB`, he reports finding optimal tours on almost half his runs, each of which took about 35 minutes on a Sun workstation. This substantially outperforms even multiple-run Lin–Kernighan. Braun did not specify the model workstation used, but one can guess that it was probably at least 10 times slower than the SGI Challenge. Even if LK is allowed the full 35 minutes, which yields about 1200 runs, the expected best would be 0.3% above optimal. Moreover, this estimate is based on a suite of 10 000 runs, and not one of them found an optimal tour.

Ulder et al. [1991] considered using Lin–Kernighan itself for the local optimization phase, comparing it to an approach using 2-Opt. Their results are not directly comparable to the earlier results because they used a common bound on the overall (sequential) running times for both variants, a bound considerably smaller than Mühlenbein et al. would have used. Nevertheless, although the 2-Opt genetic algorithm degraded to a 2.99% excess over optimal on `att532`, the LK version averaged only 0.17% over, with comparable results for other

TSPLIB instances of similar size. Because the running times were equal, the LK implementation could process significantly fewer generations than the 2-Opt implementation, but in this case stronger local optimization more than made up for fewer generations.

Martin, Otto & Felten's algorithm and Iterated Lin–Kernighan

The above studies suggest that, for a given amount of (sequential) time, the performance of a genetic algorithm can be improved by using a more powerful local optimization algorithm, assuming it is efficiently implemented. Results of Brady and Mühlenbein et al. suggest that performance can also be improved by increasing the population size. Which is more important? In the context of the TSP, it now appears that the power of the algorithm is by far the more important factor. This was first made apparent by Martin, Otto & Felten [1991, 1992], who designed a high-performing 'genetic' algorithm with a population size of one! In biological terms this might be called a *parthenogenetic* algorithm.

Matings in parthenogenetic algorithms are no longer possible, of course, and one must restrict attention to mutations, about which we have said very little. A mutation subjects a solution to some form of randomized alteration that makes localized changes to the solution while leaving most of its structure intact, like the small changes in DNA structure that cause biological mutations. In an ordinary genetic algorithm, mutations are one way of adding additional diversity to the population. In the Martin–Otto–Felten algorithm they are the only way to keep the process going.

The mutation used by Martin et al. is the *double-bridge* 4-Opt move described in Section 3.5, chosen because it normally cannot be found (or directly undone) by 3-Opt or Lin–Kernighan, as explained in the original Lin–Kernighan paper [1973]. Martin, Otto & Felten choose their mutations randomly from a restricted set of double-bridge moves in which no added edge exceeds a given length bound, typically a small multiple of the average nearest neighbor distance. Selection, which in this case means only resolving the question of whether to keep the current tour or replace it with its locally opted offspring, is done as in simulated annealing with a small fixed temperature c. That is, if the offspring is shorter, it is selected, otherwise it is accepted with probability $e^{-\Delta/c}$, where Δ is the amount by which its length exceeds the length of its parent. Because of this randomized acceptance criterion, Martin et al. originally referred to their algorithm as the *Large-Step Markov Chain* algorithm. The shorter and more descriptive term *Chained Local Optimization* has since been introduced to describe the general procedure of which this algorithm is a special case [Martin & Otto, 1996].

For local optimization Martin et al. primarily used a fast implementation of 3-Opt using neighbor lists and the gain criterion, along with the equivalent of the *don't-look* bits mentioned in Section 3.3. The don't-look bits can be exploited even more successfully in this context. Recall that if city c has its don't-look bit on, we do not consider it as a candidate for t_1, something we can implement simply and efficiently by keeping the cities whose bits are off in an *active* queue. When we

start the local optimization process for an offspring, the active queue needs to contain only the eight cities that were endpoints of the edges broken by the preceding double-bridge mutation. As usual, we add a city to the queue when one of its tour neighbors changes as the result of an improving move, but typically there will not be many such additions. This procedure is something like the Mühlenbein et al. idea of only allowing moves to break edges not shared with both parents, although it is a bit more flexible.

The results for this algorithm were quite promising. It found optimal solutions to the 318-city TSPLIB problem lin318 in an hour on a SPARCstation 1 (perhaps 4–6 minutes on our SGI Challenge. For att532 it could get within 0.07% of optimal in 15 hours (1–1.5 hours on our SGI Challenge). In limited experiments with a version of Lin–Kernighan as the local optimizer, the algorithm typically found optimal solutions for this instance in 3 hours (12–18 minutes on our SGI Challenge). These are clearly much better results for this instance than any we have discussed so far.

A more thorough study of the value of Lin–Kernighan in this context was reported by Johnson [1990]. Inspired by preprints of Martin et al. [1991], Johnson studied a simplified variant of the Martin–Otto–Felten algorithm that he called *Iterated Lin–Kernighan* (ILK), where an *iteration* corresponds to a generation in genetic algorithm terms. Johnson's simplifications consisted of removing the length restriction on the 4-Opt moves, so that a mutation was simply a random double-bridge move, and removing the randomness from the acceptance criterion, with the locally opted offspring being rejected if it was longer than its parent. Unbeknownst to Johnson, the term *iterated* had already been reserved for just this sort of algorithm by Baum [1986a, 1986b], who proposed an *iterated descent* algorithm for the TSP with 2-Opt as the local optimization algorithm and the role of the 4-Opt more played by a random 2-Opt move. The key distinction between *chained* and *iterated* local optimization is that iterated optimization dispenses with the randomized acceptance criterion.

An early version of the Lin–Kernighan implementation discussed in Section 4.2 was used, but with all cities initially in the active queue at the beginning of each iteration. Johnson reported finding optimal solutions for lin318, pcb442, att532, gr666, pr1002, and pr2392, which were all the large TSPLIB instances for which optimal solutions were then known. Not all runs yielded optimal solutions of course. In going from lin318 to pr2392, the times per run, translated to our SGI Challenge, increased from 11 minutes to 5.5 hours, whereas the fraction of runs leading to optimal solutions dropped from 60% to 10%. However, even when not optimal, the solutions found were typically far better than multiplerun LK could find in the same number of iterations, and the running time was far less.

It is now widely believed that the Martin–Otto–Felten approach, in particular the ILK variant, is the most cost-effective way to improve on Lin–Kernighan, at least until one reaches stratospheric running times, an eventuality we shall discuss briefly in Section 6.3. ILK was used as a primary upper-bounding tool in

the optimization techniques of Applegate et al. [1995] that have set the last three records for the largest TSP instance solved to optimality (3038, 4461, and 7397-city instances, respectively). Johnson, McGeoch & Rothberg [1996] have used month-long runs of ILK to confirm that the Held–Karp lower bound remains with 0.75% of the optimal tour length for random Euclidean instances with as many as 100 000 cities. ILK also holds promise for more practical uses. In the next section we shall summarize results for ILK used in what might be called 'production mode'.

Before concluding this section, however, we should point out that no thorough study has yet been performed comparing ILK to an equally well-implemented version of the original Martin–Otto–Felten approach. Recent results of Martin & Otto [1995] suggest that the two approaches are actually quite well-matched on an iteration-for-iteration basis.

6.2 Production-mode Iterated Lin–Kernighan: experimental results

Most of the work discussed in the previous section, including the early research on the Martin–Otto–Felten algorithm and Iterated Lin–Kernighan, is difficult to extrapolate to other instances. Experiments were typically performed on just a few specific instances, and no attempt was made to determine how key parameters such as the number of iterations should be scaled to reflect instance type or size. For ILK it turns out that a very simple rule gives reasonably predictable results, and in this section we shall summarize the experiments of Johnson et al. [1997b] that confirm this. They also illustrate the trade-offs between running time and solution quality that can be made by adjusting the number of iterations.

In the version of ILK discussed here, we adopt the Martin et al. idea of starting each iteration with only eight cities in the queue of candidates for t_1, i.e., the eight cities which had a tour neighbor changed as a result of the just-completed double-bridge mutation. We also follow the suggestion of Applegate, Chvátal & Cook [1990] that the depth of all the LK searches be bounded at 50. Otherwise, the full power of Lin–Kernighan as described in Section 4.2 is applied. Let ILK (m) denote the algorithm in which ILK is run for m iterations. For our test bed of random Euclidean instances, Table 8.15 reports the average percentage excess over the Held–Karp lower bound and the running times for ILK ($N/10$), ILK ($N/10^{0.5}$) and ILK (N), contrasting them with the expected results of performing an equivalent number of LK runs from independent random Greedy starts and taking the best solution found, computed as described in Section 5.2.

Somewhat better solutions would have resulted had we started each ILK iteration with all cities in the active queue and left the LK search depth unbounded, as in Johnson [1990]. This would have come at the cost of substantial increases in running time, however, and would not have been cost-effective on a time-equalized basis. Moreover, although we do not find optimal solutions for the solved TSPLIB instances as often as did Johnson [1990], we still find them on

Table 8.15 Comparison of iterated Lin–Kernighan with independent LK runs

	10^2	$10^{2.5}$	10^3	$10^{3.5}$	10^4	$10^{4.5}$	10^5
	Average percent excess over the Held–Karp lower bound						
Independent iterations							
1	1.52	1.68	2.01	1.89	1.96	1.91	1.95
$N/10$	0.99	1.10	1.41	1.62	1.71		
$N/10^{0.5}$	0.92	1.00	1.35	1.59	1.68		
N	0.91	0.93	1.29	1.57	1.65		
ILK iterations							
$N/10$	1.06	1.08	1.25	1.21	1.26	1.25	1.31
$N/10^{0.5}$	0.96	0.90	0.99	1.01	1.04	1.04	1.08
N	0.92	0.79	0.91	0.88	0.89	0.91	
	Running time in seconds on a 150 MHz SGI Challenge						
Independent iterations							
1	0.06	0.2	0.8	3	10	40	150
$N/10$	0.42	4.7	48.1	554	7 250		
$N/10^{0.5}$	1.31	14.5	151.3	1 750	22 900		
N	4.07	45.6	478.1	5 540	72 400		
ILK iterations							
$N/10$	0.14	0.9	5.1	27	189	1 330	10 200
$N/10^{0.5}$	0.34	2.4	13.6	76	524	3 810	30 700
N	0.96	6.5	39.7	219	1 570	11 500	

over half the runs of ILK (N) for lin318 and pcb442 and on about one third of the runs for att532.

The averages in Table 8.15 are based on 10 runs per instance up to 10 000 cities, 5 for 31 623 cities, and 2 for 100 000 cities. As with our earlier tables of this sort, the last digit of precision in our percentage excess figures is not itself statistically significant, but it helps to identify those situations where figures differ by at least 0.1%, a gap that may be meaningful. Note that for 100 cities the independent run strategy appears to produce tour quality as good if not better than ILK, at least for the numbers of iterations considered. As N increases, however, ILK rapidly pulls away, even for $m = N/10$ iterations. The independent run results gradually approach those for a single run, whereas the ILK percentage excesses, like those for single-run LK itself, stay roughly constant as N increases from 1000 to 100 000, or at least grow exceedingly slowly. As to the relative improvements brought by more iterations, the first $m = N/10$ iterations yield a 0.6–0.7% improvement on single-run LK. This essentially halves the excess above the expected optimal tour length, which itself is 0.60–0.80% above the Held–Karp

bound based on the estimates of Johnson, McGeoch & Rothberg [1996]. Increasing m by a factor of $10^{0.5}$ yields another 0.2% improvement, and increasing it by a second factor of $10^{0.5}$ gains us a further 0.1%. This composite factor of 10 has again cut the distance above optimal by a factor of 2 or more, and it leaves us no more than 0.3% above the predicted optimal.

The running times reported in Table 8.15 also tell an interesting story. Since instance sizes go up by factors of $10^{0.5}$ in each successive column, a quadratic time algorithm should see its running times increase by a factor of 10 from column to column. Confirming what was already observed in Section 4.2, the results in the table clearly indicate that single-run Lin–Kernighan is subquadratic. On the other hand, even though preprocessing time can be amortized when performing multiple runs, the overall time to perform cN independent runs of LK for fixed c grows worse than quadratically. Surprisingly, ILK (cN) is itself subquadratic for each fixed value of c. This means that the average running time per iteration in an ILK run grows sublinearly, at least within the range of N covered by Table 8.15. In point of fact, there is a linear-time component to an iteration, since all cities are touched in performing each double-bridge mutation. The running time for this operation has a very low constant of proportionality, however, and for $N \leqslant 100\,000$ the dominant time in an iteration remains that for performing Lin–Kernighan. This can grow sublinearly because no matter what N is, we only start with eight cities on our active queue, and often we can stop without looking far beyond them.

Table 8.16 presents analogous data for selected `TSPLIB` instances, concentrating on the results for single-run LK and the various ILK (m). Instances `lin318`, `gr431`, `att532`, and `gr666` all have nonstandard metrics, so they were read into the computer as distance matrices, yielding larger times for neighbor-list

Table 8.16 Results for Iterated Lin–Kernighan on TSPLIB instances

	Average percent excess					Running time in seconds			
	1	$N/10$	$N/10^{0.5}$	N	OPT	1	$N/10$	$N/10^{0.5}$	N
`lin318`	1.61	0.55	0.41	0.32	0.28	1	2	5	13
`gr431`	2.72	1.41	1.02	0.89	0.69	3	9	19	57
`pcb442`	1.36	0.94	0.72	0.60	0.55	1	3	8	25
`att532`	1.91	1.20	1.08	1.03	0.97	2	5	12	29
`gr666`	2.38	1.20	0.84	0.74	0.64	3	16	38	95
`dsj1000`	3.08	1.31	0.87	0.83	0.61	6	57	164	515
`pr1002`	2.61	1.62	1.27	1.13	0.89	2	10	29	74
`pr2392`	2.85	1.68	1.62	1.42	1.22	2	20	54	148
`cb3038`	2.04	1.31	1.13	1.00	0.81	2	30	86	240
`fl3795`	8.41	6.72	4.78	4.79	< 1.04	51	1 010	3 440	7 930
`fnl4461`	1.66	1.06	0.86	0.76	0.55	3	54	164	510
`pla7397`	2.19	0.96	0.73	0.69	0.58	19	960	3 060	12 900

construction but smaller times for subsequent distance computations. Where the optimal solution is known, we present its percentage excess above Held–Karp as well, so that closeness to optimality can be judged. Averages are over 10 or more runs for each instance.

Note first that running times typically exceed those for random Euclidean instances of comparable sized by factors of 2 or more. Tour quality is also slightly worse, although the general trend of results is the same as it was for random Euclidean instances. ILK ($N/10$) again typically reduces LK's distance to optimal by a factor of 2 or more. ILK (N) often comes within 0.2% of optimal and usually is no worse than 0.4% away. The one exception is instance fl3795, which we include (even though it has not yet been solved optimally) because it is the TSPLIB instance that gives our algorithms the most trouble. Note that LK is more than 8% off the Held–Karp bound for fl3795 and ILK (N) is still roughly 4.8% away. Instance fl3795 is depicted in Figure 8.7 and is even more pathologically clustered than instance dsj1000 (see Figure 8.5). In many places the cities of fl3795 are so close together as to appear as line segments and solid rectangles. This instance is not totally out of reach to our techniques, however. If we replace the standard 20-nearest-neighbor lists of our Lin–Kernighan implementation with the 40-quadrant-neighbor scheme described in Section 3.3, LK's average excess drops to 3.9%, and ILK ($N/10$)'s average excess drops to 1.23%, at the cost of increasing running time by factors of 3 and 10 respectively. Table 8.16 gives an upper bound of 1.04% on the optimal excess for this instance, a bound attained by the best of 10 ILK ($N/3$) runs with 40-quadrant-neighbor lists.

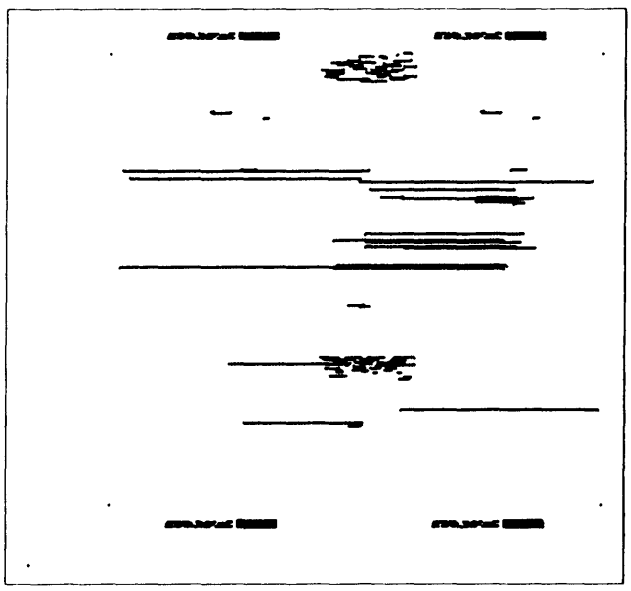

Figure 8.7 TSPLIB instance fl3795

Table 8.17 Results for Iterated Lin–Kernighan on random distance matrices

	Average percent excess			Running time in seconds		
N	1	N	$10N$	1	N	$10N$
100	1.42	0.30	0.30	0.1	1	12
316	2.48	0.45	0.12	0.3	15	130
1 000	3.56	1.09	0.55	2.0	81	730
3 162	4.60	2.29	1.37	13.3	616	4 660
10 000	5.82	3.68	2.64	131.0	5 700	43 500
31 623	6.94	5.83	-	1 600.0	74 000	

Experiments with larger TSPLIB instances reveal the same sort of performance as shown in Table 8.16, albeit with larger running times. On the largest, pla85900, ILK ($N/10$) takes roughly 100 times longer than LK, but it lowers LK's 2.17% average excess to 0.96%.

As a final illustration of the power of Iterated Lin–Kernighan, we also tested it on our random distance matrix test bed. Recall that these were instances for which even Lin–Kernighan's performance degrades as instance size increases. As Table 8.17 illustrates, ILK suffers the same fate, although it does provide improvements over LK if we give it enough iterations, in this case N and $10N$ rather than $N/10$, $N/10^{0.5}$, and N. But note how the improvement that ILK (cN) gives over LK for a fixed c declines as N increases. For instance, ILK (N) cuts the percentage excess for LK by a factor of almost 5 for $N = 100$, but only by about 15% for $N = 31\,623$. For $N = 100$, ILK (N) found the optimal solution on every run for each of the two instances in our test bed, which explains why ILK ($10N$) yields no improvement for $N = 100$. The optimal solutions have not yet been determined for larger instances in our test bed, but results reported by Johnson, McGeoch & Rothberg [1996] suggest that, for random distance matrices, the gap between the Held–Karp bound and the optimal tour length averages less than 0.01% for $N \geqslant 1000$. Thus the fact that ILK (cN) provides less improvement over LK as N increases does not seem to be because there is less room for improvement, but rather because ILK is just less effective for instances of this type when N grows large.

Nevertheless, for real-world instances at least, we can conclude that production-mode ILK is a reasonably robust way of obtaining significant improvements over Lin–Kernighan, assuming one is prepared to spend the extra amounts of time required. Within certain ranges, this extra time remains quite feasible. For geometric instances (random and real-world) of up to 1000 cities, ILK (N) typically gets within 0.2–0.4% of optimal in minutes and for instances of up to 20 000 or more cities, ILK ($N/10$) typically gets within 0.5–1.0% of optimal in under an hour.

Table 8.18 Iterated 3-Opt versus Lin–Kernighan and Iterated Lin–Kernighan

Algorithm	10^2	$10^{2.5}$	10^3	$10^{3.5}$	10^4	$10^{4.5}$
	Average percent excess over the Held–Karp lower bound					
LK	1.52	1.68	2.01	1.89	1.96	1.91
I-3Opt($N/10^{0.5}$)	1.85	1.81	2.09	2.12	2.14	2.12
I-3Opt(N)	1.57	1.49	1.73	1.72	1.77	1.76
ILK($N/10$)	1.13	1.12	1.20	1.22	1.24	1.25
	Running time in seconds on a 150 MHz SGI Challenge					
LK	0.06	0.20	0.8	2.5	10	39
I-3Opt($N/10^{0.5}$)	0.07	0.31	2.1	13.8	115	1 340
I-3Opt(N)	0.18	0.79	5.7	39.1	350	4 190
ILK($N/10$)	0.14	0.88	5.5	35.4	292	2 480

As a final question related to ILK, one might wonder whether the iterating process is equally powerful at speeding up other local optimization algorithms. Recall that the original Martin–Otto–Felten algorithm used 3-Opt rather than LK as its local search engine. Given that 3-Opt (and the iterating process) is much easier to implement than is Lin–Kernighan, one might wonder whether iterated 3-Opt might be a worthwhile option for those who do not have the time to implement Lin–Kernighan. We tested this hypothesis on our random Euclidean test bed, and the results are summarized in Table 8.18, where I-3Opt(k) denotes the algorithm that performs k iterations of Iterated 3-Opt from a Greedy starting tour.

Note that although I-3 Opt($N/10^{0.5}$) does not quite equal LK in tour quality, I-3Opt(N) beats it by 0.15–0.19% for $N > 100$, so I-3Opt(N) might well be a reasonable choice for those preferring a running time penalty to a programming time penalty. That running time penalty grows quickly, however, reaching a factor of 100 by $N = 31\,623$. If one does have an LK implementation, one can find substantially better tours in less time using ILK ($N/10$).

6.3 Further variants

The success of Iterated Lin–Kernighan supports the idea that the real contribution of the genetic approach is to provide an effective method for adaptively generating better and better starting tours for its local optimization subroutine. Given such a view, several researchers have explored other possibilities for adaptive starting tour generation. In this section, we sample some of these.

Within the realm of ILK variants, several alternative mutations have been proposed, typically involving significantly greater change to the current champion tour than a simple 4-Opt move. Hong, Kahng & Moon [1995] consider the most straightforward generalization: going from 4-Opt double-bridge moves to

random k-Opt moves for fixed values of $k > 4$. Based on experiments with `lin318`, `att532`, and an 800-city random distance matrix, and using a less powerful version of Lin–Kernighan than Johnson et al., they conclude that larger values of k can yield better results. The distinctions are tiny, however, and the statistical noise in the data appears to be high. Moreover, using higher values of k increases running time per iteration, because one must put $2k$ cities on the active queue at the beginning of each iteration. When this time is factored in, the conclusions become even murkier. But the method is certainly worth a more extensive study.

Fiechter [1994] proposes a mutation that does more damage by performing more 4-Opt moves. Instead of creating his next starting tour by performing just one such move, he proposes performing a sequence of from $N/50$ to $N/150$ of them, guiding the choice by randomized heuristics operating in tabu search mode, where the tabu list contains the cities that most recently had their tour neighbors changed by one of the 4-Opt moves. His results are not comparable to those of Section 6.2, since he used 2-Opt for his local optimization algorithm rather than Lin–Kernighan (and not even full 2-Opt but a parallel version based on tour partitioning). Given the weaker form of local optimization, the results Fiechter reports for random Euclidean instances are impressive. For $N = 10\,000$ he needs only 30 iterations to obtain average excesses that are better than those reported for 3-Opt in Section 3.2, whereas iterated 2-Opt does not catch 3-Opt on such instances even with 10 000 iterations. Each of Fiechter's iterations takes significantly more time than one of iterated 2-Opt, because of the high cost of generating his mutations, but it appears that it is time well spent. It might be interesting to see how this approach would perform with Lin–Kernighan as the local optimizer.

Dam & Zachariasen [1994] and Zachariasen & Dam [1996] have studied variants in which the mutations are performed via a tabu search in which neighbors are generated using relaxed versions of LK search or their own flower transition (described in Section 3.5), with only a constant number of neighbors being examined in each step so as to speed the process along. Local optimization is then performed using strictly downhill versions of either move, although the search continues to be guided using tabu lists and a limited application of the best-seen choice rule, as opposed to the first-seen rule of Lin–Kernighan. Viewed as a tabu search algorithm, this complex approach far outclasses the simple approaches described in Section 4.1, as well as standard Lin–Kernighan. It does not appear to be quite as good as ILK, but a hybrid of the Dam–Zachariasen mutations with a more standard Lin–Kernighan local search routine would be worth further exploration. Moreover, there are indications in their results that the flower transition may on occasion be more effective than LK search, and this is well worth studying in the context of standard neighbor-list Lin–Kernighan and ILK.

Two other intriguing mutation schemes have been proposed by Codenotti et al. [1993, 1996]. The first is restricted to geometric instances. Suppose T is the current champion tour. Randomly perturb the city coordinates by small

amounts, obtaining a new instance I'. It is rare that T will be locally optimal with respect to I', so we can perform local optimization on it with respect to I', obtaining a new tour T'. And it is rare that this new tour T' will be locally optimal in I, so it becomes our new 'official' starting tour. Codenotti et al. also propose a nongeometric scheme, in which I' is obtained by deleting a random subset of say $N/200$ cities. The tour T' is then converted to a legal tour for the original instance by inserting each deleted city into the tour immediately after one of its 20 nearest neighbors (randomly chosen). For these schemes, Codenotti et al. report significantly better improvements on a per-iteration basis than for their version of ILK. The iterations in the new schemes are more expensive, however; although the local optimization in the transformed instance can be performed by 2-Opt, this is because 2-Opt is a nontrivial operation compared to simply generating a random double-bridge move. Moreover, each invocation of local search normally begins with far more cities in the active queue than the eight needed by ILK, since far more than four edges are likely to have changed. Consequently, Codenotti et al. do not run any of their implementations long enough to obtain tour qualities as good as those obtained by ILK ($N/10$), and their results do not give a clear picture of all the running time trade-offs involved. Nevertheless, these mutation schemes also seem well worth further study.

Gu [1994] has proposed a variant on ILK somewhat similar to the first Codenotti scheme in that the distance metric is altered between iterations. Here, however, the true metric is not introduced until the final iteration. Suppose that all distances have been normalized to lie between 0 and 1. Let d^* be the average intercity distance and α the number of iterations desired. During iteration i, the distance between cities c and c' is taken to be $d^* + (d(c,c') - d^*)^{\alpha - i + 1}$ if $d(c,c') > d^*$ and $d^* - (d^* - d(c,c'))^{\alpha - i + 1}$ otherwise. Note that, on the final iteration, distances attain their true values, although initially they are all very similar and therefore the search space is 'smoother'. The starting tour for each iteration is simply the final tour from the previous iteration. Gu [1994] only tests this approach on random distance matrices with $N \leqslant 100$ and concentrates on 2-Opt and Or-Opt as his local optimizers. He obtains significant improvements over simply performing the local optimizers by themselves. For $N = 100$ his version using 2-Opt appears to find slightly better tours than we report for 3-Opt on such instances. As we have seen, however, results obtained using random distance matrices (and using such small instances) may be highly misleading, so the case remains to be made for this 'search space smoothing' approach to the TSP.

As to genetic algorithms with population sizes exceeding 1, there has been a continuing flow of proposals for new 2-parent crossovers, e.g., see Oliver, Smith & Holland [1987], Whitley, Starkweather & Fuquay [1989], Starkweather et al. [1991], Homaifar, Guan & Liepins [1993], and Bui & Moon [1994]. More interesting is a new development in nontraditional parenting: using m-parent crossovers for $m > 2$. Boese, Kahng & Muddu [1994] propose taking the union of all the edges present in the current population of tours, assigning a weight to each edge based on the quality of the tours that contain it, then using a randomized heuristic to generate starting tours based on those weights.

Unfortunately, none of these new two-or-more-parent proposals has yet been studied extensively enough to determine how competitive it might be with ILK. There is one multiparent proposal that *has* been shown to find better tours than ILK, however, if one is willing to spend the extra time. The running time penalty is substantial (and unavoidable) since this new crossover is embedded in a genetic scheme whose local search algorithm is not LK but ILK itself. The new proposal is based on the observation that, in typical instances, good tours share many edges. This has been experimentally confirmed for 2-Opt and 3-Opt by many authors, including Kirkpatrick & Toulouse [1995], Mühlenbein [1991a], and Boese, Kahng & Muddu [1994]. It seems to be even more true for Lin–Kernighan and Iterated Lin–Kernighan. In fact, as observed by Applegate et al. [1994], the graph consisting of the union of edges in a small collection of ILK tours is often sufficiently sparse and well-structured that one needn't settle for the heuristic approach of Boese et al. in generating the next starting tour. One can instead simply compute the *optimal* tour among those tours that use only edges from the graph.

The key observation is that these graphs typically have bounded *branch width*, as defined by Robertson & Seymour [1991]. Even though determining branch width is NP-hard, low branch-width decompositions for such graphs can typically be constructed using appropriate heuristics [Cook & Seymour, 1993], and once one has such a decomposition, a dynamic programming algorithm can find the optimal tour in time linear in N, albeit exponential in the width of the decomposition [Applegate et al., 1994]. For this approach to work, we must start with a population of very good parent tours, typically obtained by taking the best tours from a large collection of runs of ILK(m) with $m \gg N$, a very time-consuming approach. We can then 'mate' various subsets of this population and apply ILK to the derived tours, which although optimal for the set of edges in their parents, need not be optimal with respect to the entire instance and may therefore be improvable. The resulting tours are then added to the population, and we can try again.

This approach was a key step in finding and verifying the optimal tour lengths for `fnl4461` and `pla7397`, the two largest `TSPLIB` instances solved to date. On `pla7397` the best tours found by ILK were still 0.01% above what turned out finally to be the optimal length, and the branch-width approach reduced the gap to 0.001%. Thus the practical value of the branch-width-approach is probably nil, except to those attempting to set new TSP records. Within that context, however, it seems the best way to improve on ILK, so it deserves a place of honor at one end of the time-tour quality trade-off curve for TSP heuristics. More details on the approach should soon be forthcoming from Applegate et al.

In the meantime, the more straightforward approach of simply performing many independent long runs of ILK and taking the best has much to recommend it, assuming the time is available and differences of less than 0.02% are important. Although in our experiments an average run of ILK (N) was typically better than the best of 10 ILK($N/10$) runs, the analogous statement does not hold for

substantially longer runs. For example, Johnson, McGeoch & Rothberg [1996] were able to find optimal tours quite frequently for 1000-city random Euclidean instances by taking the best of 20 ILK (10 000) runs. The alternative of performing a single run of ILK (200 000) would probably not have worked as well, given that the majority of the ILK (10 000) runs had stopped improving by the 5000th iteration, and given that they typically averaged around 0.01% above the best-of-20 result.

7 NEURAL NETWORK ALGORITHMS

This section must of necessity come as something of an anticlimax to what has gone before, since none of the TSP algorithms so far proposed under the general rubric of 'neural net algorithms' has proved competitive with more classical approaches. In particular, none can produce tours even as good as those we have reported for single-run 3-Opt, and most take substantially more time, at least on sequential machines. This section is also somewhat redundant, given Potvin's excellent recent survey on the topic [1993]. We shall thus only briefly sketch the major ways in which researchers have attempted to adapt the neural net approach to the TSP, referring the reader to Chapter 7 for precise details of how neural algorithms work and to Potvin [1993] for a more complete and detailed survey of the variations on these themes. Since some of the studies of neural net algorithms have based their comparisons on a faulty understanding of expected optimal tour lengths and the true potential of classical algorithms, one of our emphases will also be on putting such reported results into more accurate perspective.

The set of neural net algorithms for the TSP can currently be divided into two main classes. The first consists of generally applicable algorithms in which the neurons are organized according to some formulation of the TSP as an integer program. The second is restricted to geometric instances and consists of algorithms in which the neurons can be viewed as points in space seeking out cities with which to identify. We shall cover each class in turn.

7.1 Neural networks based on integer programs

The first application of a neural net approach to the TSP was due to Hopfield & Tank [1985]. Their approach was based on the integer programming formulation of the TSP depicted in Figure 8.8. Here $x_{ik} = 1$ is taken to mean that city c_i is the kth city in the tour, in which case the sum being minimized is the tour length. The first constraint says that each position contains precisely one city and the second says that each city is in precisely one position.

Hopfield & Tank's algorithm attempted to find feasible solution to this integer program by viewing the x_{ij} as *neurons* that could take on arbitrary values in the interval [0, 1]. The neurons were connected up with an inhibitory network that tried simultaneously to impose the constraints and to lower the cost of a surrogate *energy* function. Local optima for this energy function were found using a local optimization algorithm where individual neurons changed state so as to

$$\text{Minimize} \sum_{i=1}^{N} \sum_{j=1}^{N} d(c_i, c_j) \cdot \left(x_{i,N} \cdot x_{j,1} + \sum_{k=1}^{N-1} x_{i,k} \cdot x_{j,k+1} \right)$$

$$\text{Subject to} \sum_{i=1}^{N} x_{ik} = 1, \quad 1 \le k \le N$$

$$\text{and} \sum_{k=1}^{N} x_{ik} = 1, \quad 1 \le i \le N$$

$$\text{and} \quad x_{ik} \in \{0, 1\}, \quad 1 \le i, k \le N$$

Figure 8.8 An integer programming formulation for the TSP

lower their contribution to the total energy. Multiple random starts were allowed.

In the context of the other algorithms we have discussed in this chapter, Hopfield & Tank's results were not at all promising. Although they regularly found optimal tours for 10-city instances, they often failed even to converge to feasible solutions when $N = 30$, and the best solution they ever found on such an instance was still more than 17% above optimal. Furthermore, their approach is very sensitive to the connection weights, and even results of the above quality were difficult for other researchers to reproduce [Wilson & Pawley, 1988]. To round things off, the computational requirements of the approach are exorbitant: N^2 neurons are required, each connected to all the others. A fully parallel implementation would thus require $\Theta(N^4)$ hardware, whereas a sequential simulation would have running time $\Omega(N^4)$.

Subsequently, researchers have tried many techniques to salvage this basic scheme. These include modifying the energy function as for example in Brandt et al. [1988], using simulated annealing to perform a discretized version of the local optimization as in the *Boltzmann machine* approach of Aarts & Korst [1989b], and replacing the N neurons representing $X_{i,k}$, $1 \leqslant k \leqslant N$ by a single N-dimensional *Potts neuron* and using *mean field annealing*, as in the approaches of Peterson & Söderberg [1989] and Van den Bout & Miller [1989] (see Chapter 7). Unfortunately, although some of these approaches led to feasible solutions for as many as 200 cities, the tour length obtained seem to have been 8% or more above the Held–Karp bound and hence easily dominated by the (much faster) 2-Opt algorithm of Section 3. Moreover, although several of these approaches succeeded in reducing the number of interconnections needed in the corresponding network from $\Theta(N^4)$ to $\Theta(N^3)$, their computational requirements still remain far too great for instances of reasonable size.

A key factor holding all these variants back is the integer programming formulation of Figure 8.8, which requires that a large portion of a neural net's work be spent merely getting a feasible solution. Researchers trying to find optimal solutions to the TSP via polyhedral techniques have relied on a much

$$\text{Minimize} \sum_{i=1}^{N} \sum_{j=1+1}^{N} d(c_i, c_j) \cdot x_{i,j}$$

$$\text{Subject to} \sum_{i=k \text{ or } j=k} x_{ij} = 2, \quad 1 \le k \le N$$

$$\text{and} \sum_{|S \cap \{i,j\}|=1} x_{ij} \ge 2, \text{ for all } S \subseteq \{1, 2, \ldots, N\} \text{ with } 1 \le |S| < N,$$

$$\text{and } x_{ij} \in \{0, 1\}, \; 1 \le i \le j \le N$$

Figure 8.9 A better integer programming formulation for the TSP

more effective formulation, as depicted in Figure 8.9. Here the variables x_{ij}, $1 \leqslant i < j \leqslant N$, correspond to potential tour edges, and we take $x_{ij} = 1$ to imply that the tour contains an edge between cities c_i and c_j. (The Held–Karp lower bound on the optimal tour length is in fact the solution to the linear programming relaxation of this formulation in which the integrality constraint is replaced by $0 \leqslant x_{ij} \leqslant 1$.)

This formulation has its own drawbacks. In particular, there are an exponential number of the middle *subtour elimination* constraints, one for each of the possible proper subsets S. Thus neural net algorithms based on the Figure 8.9 formulation normally begin simply by ignoring these constraints. These algorithms typically use only $\Theta(N^3)$ hardware or sequential time, a significant improvement over the $\Theta(N^4)$ of the original Hopfield–Tank approach, although still impractical for large instances. Dropping the subtour constraints means that any collection of disjoint cycles is a feasible solution, not just Hamiltonian circuits through all the cities. (It also means that the minimum cost solution can be computed exactly and efficiently by the classic *b-matching* techniques of Edmonds & Johnson [1970], although neural net algorithms have generally ignored this fact.) The problem thus becomes that of turning a disjoint collection of cycles into a tour.

Aarts & Korst [1989b], in a Boltzmann machine implementation, use extra neurons to insure that locally optimal solutions correspond to connected graphs. Joppe, Cardon & Bioch [1990] propose something similar, with a second level of neurons that can trigger changes to the external inputs for the first level. Neither approach seems to have been tried on instances with more than 30 cities, however. A more effective (if less neural) approach appears to invoke the classical algorithms of Karp [1977]. This was proposed and implemented by Xu & Tsai [1991], who were able to handle up to 150-city instances. Before switching to Karp's patching heuristic, Xu & Tsai perform five successive neural net runs, each adding penalty terms to the energy function so as to inhibit the creation of the subtours seen in the previous rounds. This in effect implements some of the subtour constraints from the Figure 8.9 formulation, and it succeeds in reducing the number of cycles that eventually need to be patched together, at least for the still relatively small instances they consider.

Unfortunately, the results are not impressive except perhaps in the context of other neural net algorithms. Xu & Tsai's main proposal thus concerns a further hybridization with classical algorithmic techniques: to use the tours they generate as starting tours for 2-Opt and Lin–Kernighan. They claim that one run of 2-Opt (Lin–Kernighan) using their method of starting tour generation yields better tours on average than the best of 25(20) random starts of the corresponding local optimization algorithm. But as shown in Section 3.2, random starts are already significantly outperformed by Nearest Neighbor and Greedy starts in this context. If one converts their data to a form comparable with this chapter, it appears that for 100-city random Euclidean instances they do not on average obtain better tours than those for an average Greedy-start 2-Opt run, much less the best of 25. Moreover, when they use their version of Lin–Kernighan as postprocessor, their average tours do not even beat the average we report for a single 3-Opt run. This suggests an error in their implementation of Lin–Kernighan.

Nevertheless, for neural net algorithms based on the integer programming formulations of Figures 8.8 and 8.9, it appears that hybrids with classical techniques are the only way to go if one wants to obtain reasonably good tours. Given the substantial computational requirements of these approaches and the difficulties that many of them have with even maintaining feasibility as N grows, hybrid approaches also appear to be the only way to go if one wants to handle large instances. With this as motivation, Foo & Szu [1989] propose a hybrid with the geometric partitioning schemes mentioned in Section 3.4. Unfortunately, it seems that once a hybrid algorithm has been constructed, one can always obtain still better performance both in time and tour quality by simply removing the neural component entirely and replacing it with single-run 3-Opt.

In the next section we will consider approaches that, for geometric instances, run considerably more quickly than the approaches based on integer programming formulations, and hence can be directly applied to much larger instances.

7.2 Geometric neural networks

In a typical geometric neural network the neurons can be viewed as a set of $M \geqslant N$ points in the plane, initially positioned as the vertices of a regular M-gon in the middle of the instance, as shown on the left of Figure 8.10. The goal is to iteratively move these vertices toward cities, thus deforming the M-gon as on the right of Figure 8.10. This process continues until the M-gon looks like a tour, with each city matched with one of the M-gon's vertices, and each unmatched vertex lying on the straight line between the matched vertices immediately before and after it on the M-gon. There are two basic variants on this theme, the *elastic net* of Durbin & Willshaw [1987] and the *self-organizing map* derived from ideas of Kohonen [1988b].

Elastic nets

In the Durbin & Willshaw elastic net approach, each iteration of the algorithm simultaneously updates the positions of all the vertices (and hence can be

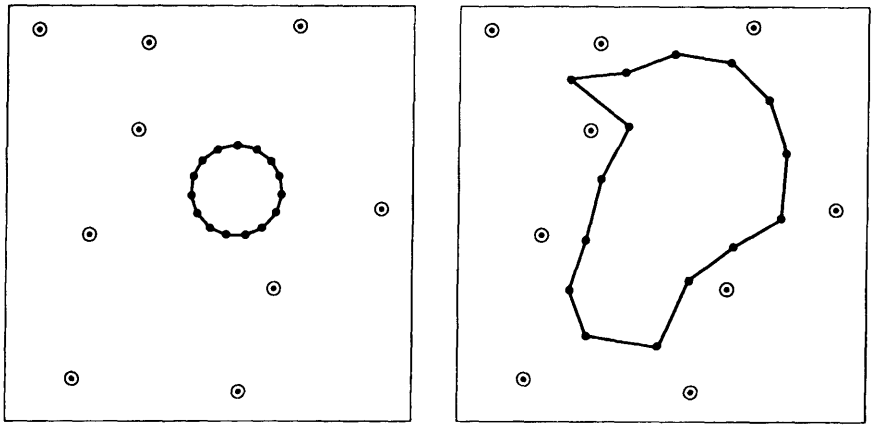

Figure 8.10 A geometry-based neural net at the start and in the middle of its execution

executed in parallel if desired). In an iteration each vertex independently determines its new location depending on its current positions, the locations of the cities, the positions of the two neighboring M-gon vertices, and a control parameter K that is reduced after each iteration. The motion of the vertices is governed by two forces. The first pulls the vertices toward the cities, with the relative attraction of the nearer cities growing stronger as K is reduced. This is what drives the vertices to match up with particular cities. Ties are avoided by choosing M to be a sufficiently large multiple of N and by taking advantages of side effects of the second force. This second, unparameterized force pulls each vertex toward the current locations of its two neighboring vertices in the M-gon. It thus acts to keep the perimeter of the M-gon short. By appropriately balancing these two forces, one can insure that each city is eventually matched to a vertex of the M-gon and that no vertex is matched to more than one city [Durbin, Szeliski & Yuille, 1989; Simmen, 1991].

As described by Durbin & Willshaw, a sequential implementation of the elastic net algorithm would take $\Theta(N)$ time per vertex per iteration. In their case this meant an overall time of $\Omega(N^3)$ since the number of iterations they typically performed was a large multiple of N. They were thus constrained as to the largest instance they could test, and they considered no instance with more than 100 cities. Furthermore, their only comparisons were with a simulated annealing algorithm of unknown quality. Nevertheless, since they concentrated on random Euclidean instances, we can get some idea of how their performance stacks up. For 50-city random Euclidean instances, the elastic net approach averaged 3% worse than the best simulated annealing solutions they saw. This is clearly worse than the 3-Opt results reported in Section 3.2, where the average excess over the Held–Karp lower bound was less than 3%. For the single 100-city instance they tested, the elastic net algorithm did better than their simulated annealing implementation, although probably still worse than 3-Opt.

Indeed, subsequent experiments suggest that when $N > 100$ the elastic net approach does not even outperform 2-Opt, which typically gets within 5% of the Held–Karp bound for random Euclidean instances. Peterson [1990a] compared the elastic net approach to various others on 200-city instances and obtained results that averaged 6% worse than those found by a version of the Mühlenbein et al. [1988] genetic algorithm described in Section 6. Vakhutinsky & Golden [1996] used a hierarchical scheme to speed up the process (and slightly improve tour quality) and were able to handle 500-city random Euclidean instances, but they appear to have found tours that were more than 10% above the Held–Karp bound.

Boeres, de Carvalho & Barbosa [1992] sped up the elastic net approach in much the same way that Bonomi & Lutton [1984] sped up simulated annealing, using a partition of the until square into a grid of smaller cells to help narrow their searches. In particular, when computing the force imposed on a vertex by the cities, they restricted attention to just those cities in nearby cells. This enabled them to handle 1000-city random Euclidean instances, and they claim there is no significant loss in tour quality. Their tours, however, are 5–10% worse than those found by their implementation of Lin–Kernighan, and therefore they are worse than 2-Opt could have provided, given that 2-Opt is typically only 3% or so behind Lin–Kernighan on random Euclidean instances. Moreover, 2-Opt and even Lin–Kernighan would have been substantially faster: for 1000 cities the Boeres et al. running time is roughly 15 000 seconds on a SPARCstation, versus an average of less than 1 second for Lin–Kernighan on our SGI Challenge, a machine that is at most 15 times faster. In their paper, Boeres et al. claim significant speed advantages for elastic nets over Lin–Kernighan, but this comparison is based on an implementation of Lin–Kernighan that appears to have a worse than quadratic running time growth rate, as opposed to the decidedly subquadratic running time reported for Lin–Kernighan in Section 4.2.

Self-organizing maps

The *self-organizing map* approach is a variant on the elastic net approach that has handled significantly larger instances. This variant was inspired by the *competitive neural nets* of Kohonen [1988b]. Here the competition is between the vertices of the M-gon. At each step we choose a random city c and determine that vertex v which is closest to it. The location of v is then moved toward that of c, with v's neighbors on the M-gon also moving in that direction, but to a lesser extent, and *their* neighbors moving to a still lesser extent, etc. The strength of the effects on neighboring vertices will normally decline in later stages of the algorithm.

We choose the cities according to random permutations, rather than making an independent choice at each step, so we can speak of *rounds* of the cities. These are different from the 'iterations' of the elastic net algorithm. Whereas all updates in an iteration of the elastic net algorithm can at least conceptually be computed and performed in parallel, the updates in a round of the self-organizing map approach must be performed sequentially because there is a real possibility that

a single vertex may be the closest vertex to two different cities. Some variants of the self-organizing map approach respond to such a double win by duplicating the winning vertex. The new vertex is inserted into the (now) $(M + 1)$-gon between the old vertex and one of its neighbors, and this new vertex is declared the champion for the current city. Often this vertex addition approach is accompanied by a vertex deletion strategy, whereby vertices that fail to win any competitions in successive rounds get deleted.

Experiments with self-organizing map algorithms have generally led to the conclusion that, although they can be made to converge faster than elastic net algorithms, they tend to produce slightly worse tours [Angéniol, Vaubois & Le Texier, 1988; Burke & Damany, 1992]. Fort [1988] applied the approach (without vertex duplication) to 400-city instances and got within 3% of Stein's [1977] estimate that for random Euclidean instances the expected optimal tour length is approximately $0.765\sqrt{N}$. As we said in Section 2.3, however, Stein's formula is a serious overestimate. For instances of this size, 3-Opt typically finds tours that are *better* than Stein's estimate, and 2-Opt less than 2% above.

Fritzke & Wilke [1991] applied a version of the algorithm with vertex duplication to instances from `TSPLIB` up to `pr2392`. In order to handle these larger instances, they restricted their nearest vertex searches to a constant number of candidates, thus speeding the algorithms up from $\Omega(N^2)$ to approximately linear time per round. The restriction on candidates was accomplished in the first round by starting with $M = 3$ (an allowable starting value since vertex duplication will eventually increase M to at least N). In subsequent rounds the candidates for the nearest vertex to city c were restricted to the previous nearest vertex plus the k vertices nearest to it on the M-gon. Fritzke and Wilke's algorithm apparently needed only a bounded number of rounds to converge, thus yielding linear time overall. The constant of proportionality was high, however, yielding a time for `pr2392` of about 1000 seconds on a SPARCstation 2, compared to 1.2 seconds for neighbor-list 3-Opt on our SGI Challenge machine, no more than 10 times faster. As to tour quality, Fritzke & Wilke were some 10% above optimal and hence significantly worse even than 2-Opt, as they themselves observe. They claim a speed advantage over 2-Opt, but this is because they were using an $\Omega(N^2)$ implementation rather than one of the much more efficient implementations discussed in Section 3.

Favata & Walker [1991] were able to handle instances with as many as 10 000 cities, using a different variant. Here there was no duplication of vertices; if a vertex was chosen by more than one city in a given round, this was taken to mean that those cities should be consecutive in the tour, but that their relative order in the tour didn't matter. One could thus at any point in the algorithm generate a tour by traversing the cities in the order in which they were matched to M-gon vertices, randomly ordering those cities matched to any given vertex. Favata & Walker took advantage of this by stopping the process as soon as their tour's length was within 10% of the length they obtained using a simulated annealing algorithm (and hence still far worse than could be obtained using 2-Opt). Even so, their running time for a 1000-city random Euclidean instance

was still more than 20 times that for neighbor-list 2-Opt (after adjusting for the relative speeds of their VAX 3600 and our SGI Challenge). They do not report times for their 10 000-city instance. Aarts & Stehouwer [1993] apply a similar self-organizing map approach to instances from TSPLIB of up to 11 849 cities. Although they apparently perform far more rounds than Favata & Walker, their tour quality still loses to 2-Opt by a margin that grows with instance size. The average percentage excess over Held–Karp is 17.4% on the 11 849-city instance, compared to 6% for 2-Opt.

As of this writing, the largest instance to which a self-organizing map algorithm has been applied is a 30 000-city random Euclidean instance handled by Amin [1994]. Amin's algorithm uses dynamic duplication and elimination of vertices and yields tour quality in the same ballpark as reported by the earlier authors: 4% worse than the Stein bound, which at $N = 30\,000$ is itself more than 7% above the average Held–Karp lower bound. Unfortunately, Amin's algorithm has a running time that grows quadratically, so it must use long running times as opposed to algorithmic efficiency to handle instances this large. The 30 000-city instance required 44 workstation hours [Arthur, 1994], compared to less than 20 seconds for 2-Opt or 3-Opt on our SGI Challenge.

From all this we can only conclude that the geometric neural net approach, despite its ability to handle larger instances, has no more practical value than the approaches based on integer programming formulations. If the large body of research into refining these algorithms is to have any practical consequences, it will most likely have to be in other domains, where the lessons learned from the TSP might bear more useful fruit.

8 CONCLUSION

In this chapter we have surveyed a wide variety of approaches to the TSP. The best ones are based on local search in one form or another. Assuming one has enough time to run something more sophisticated than a simple tour construction heuristic, one's first choice would probably be an efficient implementation of one of the classic local optimization algorithms 2-Opt, 3-Opt, and Lin–Kernighan, as discussed in Sections 3 and 4. Lin–Kernighan comes within 1.5% of optimal for random Euclidean instances with as many as a million cities, and it is capable of doing almost as well for the real-world instances in TSPLIB. Within the running time bounds of these algorithms, no tabu search, simulated annealing, genetic, or neural net algorithm has yet been developed that provides comparably good tours. If shorter tours are wanted and significantly more time is available, both simulated annealing and genetic algorithms can for many instances find better tours than could be found in the same time by performing multiple independent runs of Lin–Kernighan. These approaches also have the advantage that one can use them without the detailed coding required to implement Lin–Kernighan. If one does have an implementation of Lin–Kernighan, the best choice would seem to be the 'production-mode' Iterated Lin–Kernighan algorithm of Section 6.2. This algorithm can be implemented so that its running time

grows subquadratically for $N \leqslant 100\,000$, and it can typically reduce the excess above optimal to 0.2–0.5% or less. If time is no object, solutions within 0.1% or less of optimal can be found by performing several longer runs of ILK, and still better solutions can be obtained by genetic algorithms based on ILK and the branch-width mating scheme of Applegate et al. [1994].

These empirical results all contrast sharply with what is known theoretically about local search for the TSP, since most of the theoretical TSP results are worst-case results, or are far from tight upper bounds on average-case performance. The one domain where the predictions of theory do seem to hold up is the mathematically interesting but application-free realm of random distance matrices. Here all the heuristics we studied produce tours whose percentage excess over the optimal tour length grows with N, although for Lin–Kernighan the average excess is still less than 7% for $N \leqslant 30\,000$.

What lessons do these results provide for other applications? First, one must recognize that the TSP is something of an outlier among NP-hard optimization problems as far as the quality of heuristics go. Most other problems do not have a classical algorithm with the speed and effectiveness of the Lin–Kernighan algorithm, thus leaving much more room for alternative approaches to shine. Nevertheless, careful attention to data structures and algorithmic trade-offs will probably still be necessary to obtain good performance. For many TSP heuristics, running times can be reduced by a factor of N or more by making appropriate implementation choices, and this is just as true for simulated annealing and neural nets as it is for 3-Opt and Lin–Kernighan. Studying the ways in which this was done for the TSP may well be a good source of ideas about how best to accomplish it in other domains.

ACKNOWLEDGMENTS

The authors are grateful to many people for their help in preparing this chapter. A long list of people helped us to track down references and where relevant to clarify technical issues related to their own work. For help in this way, we thank Emile Aarts, Shara Amin, David Applegate, Jon Bentley, Kenneth Boese, Thang Bui, Bruno Codenotti, Bill Cook, Hal Gabow, Michel Gendreau, Fred Glover, Bruce Golden, Jun Gu, Andrew Kahng, Scott Kirkpatrick, John Knox, Jooyoung Lee, Jan van Leeuwen, Tom Liebling, Olivier Martin, Byung-Ro Moon, Heinz Mühlenbein, David Neto, Carsten Peterson, Chris Potts, Giovanni Resta, Andre Rohe, Wei-tek Tsai, Christine Valenzuela, Steef van de Velde, Marco Verhoeven, Dominique de Werra, Xin Xu, Mihalis Yannakakis, and Martin Zachariasen. Our apologies to anyone inadvertently left out. We also thank Emile Aarts, Kenneth Boese, Marco Dorigo, Fred Glover, Howard Karloff, Jan Karel Lenstra, Christine Valenzuela, and Martin Zachariasen for their insightful comments on preliminary versions of this chapter.

Local Search in Combinatorial Optimization
Edited by E. Aarts and J. K. Lenstra
© 1997 John Wiley & Sons Ltd.

9

Vehicle routing: modern heuristics

Michel Gendreau, Gilbert Laporte, Jean-Yves Potvin
Université de Montréal, Montréal

1 INTRODUCTION 311
2 SIMULATED ANNEALING 312
2.1 The Alfa, Heragu and Chen algorithm 313
2.2 Osman's simulated annealing algorithm 313
3 TABU SEARCH 316
3.1 Two early tabu search algorithms 316
3.2 Osman's tabu search algorithm 316
3.3 TABUROUTE 318
3.4 Taillard's algorithm 319
3.5 Computational comparison 320
3.6 Tabu search algorithms for the VRP with time windows and capacity constraints 320
4 GENETIC ALGORITHMS 323
4.1 Hybridization with a greedy heuristic 326
4.2 GIDEON 327
4.3 GENEROUS 328
5 NEURAL NETWORKS 330
5.1 Self-organizing maps 330
5.2 The hierarchical deformable net 333
6 CONCLUSION 335
ACKNOWLEDGMENTS 336

1 INTRODUCTION

The *vehicle routing problem* (VRP) holds a central place in distribution management. It can be simply stated as the problem of determining optimal delivery or collection routes through a set of locations, subject to some constraints. More formally, the VRP is defined on a directed graph $G=(V, A)$, where $V=\{v_0,\ldots,v_n\}$ is a vertex set and $A=\{(v_i, v_j): v_i, v_j \in V, i \neq j\}$ is an arc set. Vertex v_0 represents a depot at which is based a fleet of m identical vehicles. The value of m is either *fixed a priori*, or *free*, i.e., bounded above by a constant $\bar{m} \leqslant n$. All remaining vertices represent cities or customers. A nonnegative distance matrix $D=(d_{ij})$ is defined on A. Depending on the context, d_{ij} can be interpreted as a travel cost or as a travel time. If $d_{ij}=d_{ji}$ for all (v_i, v_j), the problem is said to be symmetric and

arcs are then replaced by undirected edges. The VRP consists of determining m vehicle routes of minimal total cost, each starting and ending at the depot, so that every customer is visited exactly once, subject to a number of side constraints: (1) in the *capacitated* VRP a nonnegative weight q_i is associated with each vertex and the total weight of any route may not exceed the vehicle capacity Q; (2) in the *distance or time constrained* VRP, d_{ij} represents a travel time and each customer has a nonnegative service time δ_i; then the total duration of any route may not exceed a preset bound L; (3) in the VRP *with time windows* each city v_i must be visited within a given time interval $[a_i, b_i]$, but usually vehicles are allowed to wait at a location if they arrive before the beginning of the time window. There exist several versions of this problem; the objective can be defined in terms of distance or time (waiting time is included in the second case, but not in the first); another common objective is to minimize the number of vehicles used in the solution. These objectives may be combined or they may be optimized in a hierarchical fashion.

Several formulations and algorithms have been proposed for the VRP: see the surveys of Bodin et al. [1983], Christofides [1985], Laporte & Nobert [1987], Laporte [1992], Desrosiers et al. [1995], and Fisher [1995]. Several interesting variants are described by Assad [1988].

The VRP is an NP-hard combinatorial optimization problem. Several optimization algorithms have been proposed [e.g., Cornuéjols & Harche, 1993; Christofides, Hadjiconstantinou & Mingozzi, 1993; Araque et al., 1994; and Fisher, 1995], but they can rarely solve instances in excess of 50 vertices. Most researchers have concentrated on the development of approximation algorithms. Many of the latest developments have occurred in the area of local search algorithms such as *simulated annealing, tabu search, genetic algorithms,* and *neural networks.* Expository articles on these methods can be found in Chapters 4 to 7 of this book, Pirlot [1992], and Reeves [1993a]. These approximation algorithms explore the solution space to find a good solution. Both simulated annealing and tabu search start from an initial solution x_1, and move at each iteration t from x_t to a solution x_{t+1} in the neighborhood $N(x_t)$ of x_t, until a stopping condition is satisfied. If $f(x)$ denotes the cost of x then $f(x_{t+1})$ is not necessarily less than $f(x_t)$, so care must be taken to avoid cycling. Genetic algorithms examine at each step a population of solutions; each population is derived from the preceding one by combining its best elements and discarding the worst. Neural networks embody a learning heuristic that gradually adjusts a set of weights until an acceptable solution is reached. The rules governing the search differ in each case and must be tailored to the shape of the problem at hand. A fair amount of creativity and experimentation is also required. Our purpose is to survey recent applications of local search algorithms to the VRP.

2 SIMULATED ANNEALING

At iteration t of simulated annealing, a solution x is drawn randomly from $N(x_t)$. If $f(x) \leqslant f(x_t)$, then x_{t+1} is set equal to x; otherwise

$$x_{t+1} := \begin{cases} x & \text{with probability } p_t, \\ x_t, & \text{with probability } 1 - p_t, \end{cases}$$

where p_t is usually a decreasing function of t and of $f(x) - f(x_t)$. It is common to define p_t as

$$p_t = \exp(-[f(x) - f(x_t)]/c_t), \tag{1}$$

where c_t denotes the *temperature* at iteration t. The rule employed to define c_t is called a *cooling schedule*. Typically, c_t is a decreasing step function of t: c_t is initially set equal to a given value $c_1 > 0$ and multiplied by a factor α $(0 < \alpha < 1)$ after every T iterations, so the probability of accepting a worse solution should decrease with time. Three common stopping criteria are: (1) the value f^* of the incumbent x^* has not decreased by at least π_1% for at least k_1 consecutive cycles of T iterations; (2) the number of accepted moves has been less than π_2% of T for k_2 consecutive cycles of T iterations; (3) k_3 cycles of T iterations have been executed.

We are aware of two applications of simulated annealing to the VRP: Alfa, Heragu & Chen [1991] study the capacitated version of the VRP, whereas Osman [1993] considers maximum route durations as well.

2.1 The Alfa, Heragu and Chen algorithm

The method described by Alfa, Heragu & Chen [1991] can be viewed as a *route first–cluster second* algorithm of the type proposed by Beasley [1983] and Haimovich & Rinnooy Kan [1985]. A giant tour is first constructed without considering weights and vehicle capacity, and then partitioned into segments of consecutive vertices whose total weight does not exceed Q. It is implicitly assumed that the number of vehicles is not fixed a priori. Alfa, Heragu & Chen consider symmetric problems. They start with the initial tour $(v_0, v_1, \ldots v_n, v_0)$. At any iteration t, three edges are randomly selected and removed from the tour, which is then tentatively reconnected in eight possible ways, as in the classical 3-opt algorithm [Lin, 1965]. These eight reconstructed tours constitute the neighborhood of the current solution x_t; each is evaluated and the tour x having the least cost is identified. The authors use the third stopping rule defined above.

The algorithm was tested on three instances taken from the literature ($n = 30$, 50, 75) and the results were rather disappointing. In each case, the best solution produced by the algorithm was significantly worse than the previously best known solution. This may be due in part to a bad adjustment of some parameters or to the neighborhood structure. It is more likely, however, that the use of a route first–cluster second strategy is mainly to blame for this poor performance: although this strategy is expected to perform well in an asymptotic sense [Haimovich & Rinnooy Kan, 1985], it has never been shown to be competitive on small or medium-size problems.

2.2 Osman's simulated annealing algorithm

The simulated annealing implementation proposed by Osman [1993] is substantially more involved in several respects: (1) it uses a better starting solution; (2)

some parameters of the algorithms are adjusted in a trial phase; (3) richer solution neighborhoods are explored; (4) the cooling schedule is more sophisticated. In order to describe the algorithm, it is useful to begin by examining the neighborhood structure. The algorithm uses a so-called *λ-interchange generation mechanism* in which two routes p and q are first selected, together with two subsets of customers S_p and S_q, one from each route, satisfying $|S_p| \leqslant \lambda$ and $|S_q| \leqslant \lambda$. The operation consists of swapping the customers of S_p with those of S_q as long as this is feasible. Either S_p or S_q can be empty, so this family of operations includes simply *shifting* customers from one route to another. As the number of combinations of route pairs and choices of S_p and S_q is usually large, this procedure is implemented with $\lambda = 1$ or 2, and in the most efficient versions of the algorithm, the search stops as soon as a solution improvement is discovered (when this does not happen, the whole neighborhood must be explored).

The algorithm that was implemented was tested on symmetric problems with m free. It is summarized as follows.

Phase 1: Descent algorithm

- Step 1, *initial solution.* Generate an initial solution using the Clarke–Wright algorithm [1964].
- Step 2, *descent.* Search the solution space using the λ-interchange scheme. Implement an improvement as soon as it is identified. Stop whenever an entire neighborhood exploration yields no improvement.

Phase 2: Simulated annealing search

- Step 1, *initial solution.* Use as a starting solution the incumbent obtained at the end of Phase 1, or a solution produced by the Clarke–Wright algorithm.
- Step 2, *parameter initialization.* Perform a complete neighborhood search using the λ-interchange generation mechanism without, however, implementing any move. Record $\Delta_{\max}$ and $\Delta_{\min}$, the largest and smallest absolute changes in the objective function, and compute β, the number of feasible (potential) exchanges. Set $c_1 := \Delta_{\max}$, $\delta := 0$, $k := 1$, $k_3 := 3$, $t := 1$, $t^* := 1$ (this is the iteration at which the best known solution has been identified within the current cycle). Let x_1 be the current solution and $x^* := x_1$.
- Step 3, *next solution.* Explore the neighborhood of x_t using the λ-interchange scheme. As soon as a solution x with $f(x) < f(x_t)$ is encountered, set $x_{t+1} := x$; if $f(x) < f(x^*)$, set $x^* := x$ and $c^* := c_k$. If a whole exploration yields no better solution than x_t, let x be the best solution encountered in the neighborhood of x_t and set

$$x_{t+1} := \begin{cases} x & \text{with probability } p_t, \\ x_t, & \text{with probability } 1 - p_t, \end{cases}$$

where p_t is defined by equation (1). If $x_{t+1} := x_t$, set $\delta := 1$.

- Step 4, *temperature update.*

  Occasional increment rule: If $\delta = 1$, set $c_{t+1} := \max\{c_t/2, c^*\}$, $\delta := 0$, $k := k + 1$.

  Normal decrement rule: If $\delta = 0$, set $c_{t+1} := c_t/[(n\beta + n\sqrt{t})\,\Delta_{\max}\,\Delta_{\min}]$.

  Set $t := t + 1$.
- Step 5, *termination test.* If $k = k_3$ then stop. Otherwise, go to Step 3.

An original feature of this implementation is the rule used to generate the cooling schedule. Instead of the standard decreasing step function or a continuously decreasing function, here temperature decreases monotonically as long as the current solution is modified. Whenever $x_{t+1} = x_t$ the current temperature is either halved or replaced by the temperature at which the incumbent was identified. This scheme is in fact a generalization of the rule proposed by Connolly [1990] in which only one temperature reset is made. Osman [1991, 1993] shows that when applied to the VRP, to the *capacitated clustering problem*, and to the *generalized assignment problem*, this cooling schedule outperforms the monotonic cooling schedules usually found in the simulated annealing literature.

The algorithm was implemented with $\lambda = 1$, using the best Phase 1 solution as a starting point for Phase 2. Tests were carried out on 26 instances. In Table 9.1, we report the values obtained on the 14 benchmark problems described by Christofides, Mingozzi & Toth [1979]. Computational results indicate that this

Table 9.1 Computational results for Osman's algorithm

Problem number	n	Type[a]	f^*	Best known solution value	Time[b]
1	50	C	528	524.61[c,d,e]	167.4
2	75	C	838.62	835.26[c]	6 434.3
3	100	C	829.18	826.14[c,d]	9 334.0
4	150	C	1 058	1 028.42[c]	5 012.3
5	199	C	1 378	1 298.79[c]	2 318.1
6	50	C,D	555.43	555.43[c,d,e]	3 410.2
7	75	C,D	909.68	909.68[c,d,e]	626.5
8	100	C,D	866.75	865.94[c,d]	957.2
9	150	C,D	1 164.12	1 162.55[c,d]	84 301.2
10	199	C,D	1 417.85	1 397.94[c]	5 708.0
11	120	C	1 176	1 042.11[c,d]	315.8
12	100	C	826	819.56[c,d]	632.0
13	120	C,D	1 545.98	1 541.14[c]	7 622.5
14	100	C,D	890	866.37[c,d,e]	305.2

[a] C = capacity restrictions; D = distance restrictions.
[b] Seconds on a VAX 8600 computer.
[c] Taillard [1993a].
[d] Gendreau, Hertz & Laporte [1994].
[e] Osman [1993].

simulated annealing implementation produces good quality results, but does not as a rule perform as well as the best available approximation algorithms. Osman notes that this simulated annealing algorithm is not very robust. In some cases (e.g., Problem 14), the descent algorithm with $\lambda = 2$ yields a better solution.

3 TABU SEARCH

Tabu search examines sequences of solutions as in simulated annealing, but the next move is made to the best neighbor of the current solution x_t. To avoid cycling, solutions that were recently examined are forbidden, or *tabu*, for a number of iterations. To alleviate time and memory requirements, it is customary to record an attribute of tabu solutions rather than the solutions themselves. The basic tabu search mechanism can be enhanced by several computational features such as diversification and intensification strategies, as described in Chapter 5 of this book.

In recent years tabu search has been applied to the VRP by a number of authors. Willard [1989], Pureza & França [1991], Osman [1993], Gendreau, Hertz & Laporte [1994] and Taillard [1993a] consider VRPs with capacity and time restrictions. Garcia [1993], Garcia, Potvin & Rousseau [1994], and Potvin et al. [1996] have applied tabu search to the VRP with time windows and capacity restrictions.

3.1 Two early tabu search algorithms

One of the first attempts to apply tabu search to the VRP is due to Willard [1989]. Here the solution is first transformed into a giant tour by replication of the depot, and neighborhoods are defined as all feasible solutions that can be reached from the current solution using 2-opt or 3-opt exchanges. The next solution is determined by the best nontabu move. The author presents limited computational experience with this approach. On three of the Christofides, Mingozzi & Toth [1979] benchmark problems, the proposed algorithm does not appear to be competitive with most known approximation algorithms.

Pureza & França [1991] define the neighbors of a solution by moving a vertex to a different route, or by swapping vertices between two routes while preserving feasibility. As in Willard, the best nontabu feasible move is selected at each iteration. As shown in Table 9.2, computational results obtained through this approach are somewhat better than those of Willard, but again they are not nearly as good as those produced with more sophisticated tabu search implementations.

3.2 Osman's tabu search algorithm

Osman [1993] again defines neighborhoods using the λ-interchange generation mechanism, with $\lambda = 2$. This includes a combination of 2-opt moves, vertex reassignments to different routes, and vertex interchanges between two routes. In one version of the algorithm, called *best-admissible*, the whole neighborhood is

Table 9.2 Computational comparisons of various tabu search algorithms: best known solutions are indicated by an asterisk

			Willard	Pureza & França		Osman BA[c]		Osman FBA[c]		TABUROUTE Standard[f]		TABUROUTE Best[h]	Taillard
Problem	n	Type[a]	f^*	f^*	Time[b]	f^*[d]	Time[c]	f^*[d]	Time[c]	f^*	Time[g]	f^*	f^*
1	50	C	588	536	973	524.61*	67.2	524.61*	114.0	524.61*	360	524.61*	524.61*
2	75	C	893	842	155	844	70.8	844	50.3	835.77	3 228	835.32	835.26*
3	100	C	906	851	1 796	835	675.0	838	1 543.0	829.45	1 104	826.14*	826.14*
4	150	C	–	1 081	7 351	1 052	3 075.0	1 044.35	3 560.0	1 036.16	3 528	1 031.07	1 028.42*
5	199	C	–	–	–	1 354	1 972.7	1 334.55	3 246.0	1 322.65	5 454	1 311.35	1 298.79*
6	50	C,D	–	560	1 557	555.43*	140.2	555.43*	173.0	555.43*	810	555.43*	555.43*
7	75	C,D	–	929	2 772	913	203.0	911	1 056.7	913.23	3 276	909.68*	909.68*
8	100	C,D	–	887	7 245	866.75	1 200.0	878	2 998.0	865.94*	1 536	865.94*	865.94*
9	150	C,D	–	1 227	19 376	1 188	2 443.6	1 184	4 755.8	1 177.76	4 260	1 162.89	1 162.55*
10	199	C,D	–	–	–	1 422	3 310.1	1 441	4 561.0	1 418.51	5 988	1 040.75	1 397.94*
11	120	C	–	1 049	1 776	1 042.11*	1 398.4	1 043	1 445.4	1 073.47	1 332	1 042.11*	1 042.11*
12	100	C	–	829	3 527	819.59	407.5	819.59	892.2	819.56*	960	819.56*	819.56*
13	120	C,D	–	1 631	3 501	1 547	1 343.0	1 547	2 834.0	1 573.81	3 570	1 545.93	1 541.14*
14	100	C,D	–	866	6 166	866.37*	5 579.0	866.37*	1 175.9	866.37*	3 942	866.37	866.37*

[a] C = capacity restrictions; D = distance restrictions.
[b] Seconds on an MC68020 microcomputer.
[c] BA = best-admissible strategy; FBA = first-best admissible strategy.
[d] Values obtained with real distances; integer values correspond to a truncated f^*.
[e] Seconds on a VAX 8600 computer.
[f] Algorithm using a single set of parameters.
[g] Seconds on a Silicon Graphics workstation, 36 MHz, 5.7 Mflops.
[h] Best solution value over several runs of the algorithms, with various sets of parameters.

explored and the best nontabu feasible move is selected; in the other version, *first-best-admissible*, the first admissible improving move is selected if one exists; otherwise the best admissible move is implemented. Results reported in Table 9.2 indicate that these two tabu search implementations produce excellent results, but they can still be improved in most cases.

3.3 TABUROUTE

Compared to the previous tabu search implementations, the TABUROUTE algorithm of Gendreau, Hertz & Laporte [1994] is rather involved and contains several innovative features. The neighborhood structure is defined by all solutions that can be reached from the current solution by removing a vertex from its current route and inserting it into another route containing one of its nearest neighbors using GENI, a *generalized insertion* procedure developed by Gendreau, Hertz & Laporte [1992] for the *traveling salesman problem* (TSP). This may of course eliminate an existing route or create a new one. The GENI procedure consists of inserting a vertex between two of its p closest neighbors on a route while performing a local reoptimization of the route.

This short description of GENI is valid for symmetric problems. To perform a generalized insertion, consider a route $(v_0, v_1, \ldots, v_t, v_0)$. Define the p-neighborhood of vertex v as follows: when v is not on the route, $N_p(v)$ is the set of the p vertices already on the route closest to v if $p \leqslant t-1$, or it is the set of all vertices on the route if $p \geqslant t$; when v is on the route, $N_p(v)$ is the set of the p vertices already on the route closest to v if $p \leqslant t$, or it is the set of all vertices on the route if $p \geqslant t+1$. Also denote by P_{ij} the set of vertices on the path from v_i to v_j for a given orientation of the route. For any given vertex v not on the route, two types of insertion are defined.

- Type I. Select $v_r, v_s \in N_p(v)$ and $v_k \in N_p(v_{r+1}) \cap P_{sr} \setminus \{v_r, v_s\}$. Delete arcs (v_r, v_{r+1}), (v_s, v_{s+1}) and (v_k, v_{k+1}); insert arcs (v_r, v), (v, v_s), (v_{r+1}, v_k) and (v_{s+1}, v_{k+1}). Paths $(v_{r+1}, \ldots, v_s)$ and $(v_{s+1}, \ldots, v_k)$ are reversed.
- Type II. Select $v_r, v_s \in N_p(v)$, $v_k \in N_p(v_{r+1}) \cap P_{sr} \setminus \{v_s, v_{s+1}\}$ and $v_l \in N_p(v_{s+1}) \cap P_{rs} \setminus \{v_r, v_{r+1}\}$. Delete arcs (v_r, v_{r+1}), (v_{l-1}, v_l), (v_s, v_{s+1}) and (v_{k-1}, v_k); insert arcs (v_r, v), (v, v_s), (v_l, v_{s+1}), (v_{k-1}, v_{l-1}) and (v_{r+1}, v_k). Paths $(v_{r+1}, \ldots, v_{l-1})$ and $(v_l, \ldots, v_s)$ are reversed.

To detemine the best move, it is necessary to compute the cost of the route corresponding to each type of insertion, to each orientation, and to each possible choice of v_r, v_s, v_k, v_l. In TABUROUTE, each generalized insertion is executed with $p = 5$.

A second important feature of TABUROUTE is that the search process examines solutions that may be infeasible with respect to the capacity or maximum route length constraints. More precisely, the objective function contains two penalty terms, one measuring overcapacity, the other measuring overduration, each weighted by a self-adjusting parameter. Every 10 iterations each parameter may be divided by 2 if all 10 previous solutions were feasible, or multiplied by 2 if they were all infeasible. This way of proceeding produces a mix

of feasible and infeasible solution and lessens the likelihood of being trapped in a local minimum. At various points during the search process, TABUROUTE reoptimizes the route in which a vertex has just been inserted. This is achieved by using the *unstringing & stringing* TSP postoptimization routine also developed by Gendreau, Hertz & Laporte [1992].

TABUROUTE does not actually use a tabu list, but random tabu tags. Whenever a vertex is moved from route r to route s at iteration t, its reinsertion into route r is forbidden until iteration $t + \theta$, where θ is an integer randomly drawn from an interval $[\underline{\theta}, \overline{\theta}]$ equal to [5, 10] in the current implementation. The idea of using random tabu durations was originally proposed by Taillard [1991] in relation to the *quadratic assignment problem* and is useful to prevent cycling. The range [5, 10] is consistent with the values proposed by Glover & Laguna [1993] for 'simple dynamic tabu term rules'.

Yet another feature of TABUROUTE is the use of a *diversification* strategy (see Chapter 5). This is achieved by penalizing vertices that have been moved frequently in order to increase the probability of considering slow-moving vertices. To achieve this, the objective function is artificially increased by adding to it a term proportional to the absolute frequency of movement of the vertex v currently being considered. Finally, TABUROUTE uses *false starts*. Initially, several solutions are generated and a limited search is carried out on each of them. The best identified solution is then selected as a starting point for the main search.

We now provide a short description of TABUROUTE. Readers are referred to the original Gendreau, Hertz & Laporte [1994] article for a detailed discussion of the parameter choices. In what follows, W is the set of vertices considered as candidates for reinsertion into another route at each iteration, $q \leqslant |W|$ is the number of these vertices for which a tentative reinsertion is actually made, and k is the number of iterations without improvement.

- Step 1, *initialization*. Generate $\lfloor \sqrt{n}/2 \rfloor$ initial solutions and perform tabu search with $W = V \backslash \{v_0\}$, $q = 5m$ and $k = 50$. This value of q ensures that the probability of selecting one vertex from each route is at least 90%.
- Step 2, *solution improvement*. Starting with the best solution observed in Step 1, perform tabu search with $W = V \backslash \{v_0\}$, $q = 5m$ and $k = 50n$.
- Step 3, *intensification*. Starting with the best solution observed in Step 2, perform tabu search with $k = 50$. Here W is the set of the $\lfloor |V|/2 \rfloor$ vertices that have been most often moved in Steps 1 and 2, and $q = |W|$.

As can be seen from Table 9.2, TABUROUTE produces high-quality results and often yields a best known solution.

3.4 Taillard's algorithm

The Taillard [1993a] tabu search implementation contains some of the features of the Gendreau, Hertz & Laporte [1994] algorithm, namely random tabu durations and diversification. It defines neighborhoods using the λ-interchange

generation mechanism described by Osman [1991, 1993], with $\lambda = 1$. Rather than executing the insertions with GENI, the algorithm uses standard insertions, thus enabling each insertion to be carried out in less time, and feasibility is always maintained. Every so often, individual routes are reoptimized using the optimization algorithm of Volgenant & Jonker [1983].

A novel feature of Taillard's algorithm is the decomposition of the main problems into subproblems. In planar problems these subproblems are obtained by initially partitioning vertices into sectors centered at the depot, and into concentric regions within each sector. Each subproblem can be solved independently, but periodic moves of vertices to adjacent sectors are necessary. This makes sense when the depot is centered and vertices are uniformly distributed in the plane. For nonplanar problems, and for planar problems not possessing these properties, the author suggests a different partitioning method based on the computation of shortest spanning arborescences rooted at the depot. This decomposition method is particularly well suited for parallel implementation because subproblems can then be distributed among the various processors. As will now be shown, the combination of these strategies yields excellent computational results.

3.5 Computational comparison

In order to compare the various tabu search algorithms just described, we report the best solution values obtained using each algorithm on the 14 benchmark problems described by Christofides, Mingozzi & Toth [1979]. As noted by Gendreau, Hertz & Laporte [1994], care must be taken when comparing these results: (1) some algorithms (Osman, TABUROUTE, Taillard) work with real d_{ij}'s whereas the truncation or rounding rule is unspecified in the case of Willard and of Pureza & França; (2) computation times are difficult to compare and are not available for the Willard and Taillard results; Taillard uses parallel computing; (3) TABUROUTE-Best gives the best results over a number of runs, with several parameters; since the number of runs was not recorded, it is meaningless to report computing times in this case. Table 9.2 reports the solution values obtained with the various algorithms, as well as their computation times. The best solution values known to have been obtained with real d_{ij}'s are indicated by asterisks.

Computational results indicate that the best known solutions are always found using Taillard's algorithm, followed closely by TABUROUTE. The solution value reported for Problem 1 ($f^* = 524.61$) is known to be optimal [Christofides, Hadjiconstantinou & Mingozzi, 1993]. Note that the full solutions obtained by Osman's algorithm and by TABUROUTE-Best are reported by Osman [1993] and by Gendreau, Hertz & Laporte [1994], respectively.

3.6 Tabu search algorithms for the VRP with time windows and capacity constraints

Some authors have recently developed tabu search algorithms for the VRP with time windows and capacity constraints. There are fewer available results in this

area than in the case of the VRP with capacity or distance restrictions only, but early results indicate this line of research to be promising.

Potvin et al. [1996] describe a tabu search algorithm applicable to VRPs with time windows. Their primary objective is to minimize the number of vehicles, and their secondary objective is to minimize the total solution time. The neighborhood structure is defined by using two types of edge-exchange operations: Or-opt moves and 2-opt* moves. Or-exchanges consist of moving a string of one, two, or three consecutive vertices into a different portion of the route. These were first proposed by Or [1976] and are fully described in Chapter 10 of this book. The 2-opt* procedure is appropriate in contexts such as the VRP with time windows in which it is critical to preserve the route orientation when performing an exchange. To perform a 2-opt* exchange, consider two oriented routes. Delete (v_i, v_j) from the first route and (v_k, v_l) from the second route; introduce arcs (v_i, v_l) and (v_k, v_j). As the number of Or-opt and 2-opt* exchanges is potentially large, the authors restrict the set of admissible exchanges as follows. For Or-opt moves, let v_i be the first vertex in the string of vertices to be moved, and v_j be the vertex after which v_i will be reinserted; then v_i must be one of the p closest neighbors of v_i. Similarly, in a 2-opt* move, v_l must be one of the q closest neighbors of v_i. Here p and q are input parameters. The authors only consider feasible moves. Their tabu search algorithm can now be summarized as follows.

- Step 1, *initialization.* Generate an initial feasible solution using the I1 insertion heuristic described by Solomon [1987].

Repeat Steps 2 through 5 until k_1 consecutive iterations have been performed without improvement, or until k_2 iterations have been performed in total.

- Step 2, *vehicle reduction.* Attempt to reduce the number of vehicles by eliminating routes with three customers or less, using Or-opt moves. Consider all customers v_i in such routes, and attempt to move into another route a string of at most three customers starting with v_i. Any such move must be feasible and must satisfy the neighborhood condition. If the objective function cannot be improved through this process, go to Step 3. Otherwise, implement the best move and repeat this step.
- Step 3, *time reduction.* Select r customers v_i at random and evaluate all 2-opt* moves that are feasible and satisfy the neighborhood condition. Implement the best nontabu move and declare the reverse move tabu for five iterations. Repeat this step until no improvement has been observed for k_3 consecutive iterations.
- Step 4, *vehicle reduction.* Starting with the best solution identified in Step 3, attempt again to reduce the number of vehicles by applying Step 2.
- Step 5, *time reduction.* Attempt to reduce the total time by proceeding as in Step 3, except that Or-opt is used instead of 2-opt*, with strings of 1, 2, or 3 vertices starting with vertex v_i. A string of vertices may be moved into the same route or into another route. This step is repeated for k_4 consecutive iterations without improvement.

The final solution is the best solution identified so far.

Table 9.3 Computational results for the Potvin et al. tabu search algorithm

		Solomon		Potvin et al.							
		I1		TABU1[c]			TABU2[d]			TABU-Best[e]	
Problem type	Number of instances	m[a]	f^*[b]	m^*[a]	f^*[b]	Time[f]	m^*[c]	f^*[b]	Time[f]	m^*[a]	f^*[b]
$R1$	12	13.6	2 695.5	12.9	2 412.0	291	12.8	2 408.3	593	12.5	2 390.7
$R2$	11	3.3	2 578.1	3.2	2 213.4	346	3.2	2 196.6	671	3.1	2 185.7
$C1$	9	10.0	10 104.2	10.0	9 872.1	201	10.0	9 870.9	392	10.0	9 850.2
$C2$	8	3.1	9 921.4	3.0	9 615.8	200	3.0	9 611.2	441	3.0	9 594.6
$RC1$	8	13.5	2 775.0	12.8	2 519.4	221	12.8	2 513.3	478	12.6	2 507.6
$RC2$	8	3.9	2 955.4	3.5	2 516.4	305	3.5	2 498.1	660	3.4	2 459.8

[a] Best number of vehicles.
[b] Best route duration.
[c] Results with one tabu search application.
[d] Results with two tabu search applications.
[e] Best results with several tabu search applications, using various sets of parameters.
[f] Seconds on a SPARC 10 workstation.

In their implementation, the authors use $p = q = 35$, $r = 15$, $k_1 = 2500$, $k_2 = 7500$, $k_3 = k_4 = 25$. Several other sets of parameters were tested by Garcia [1993] and by Garcia, Potvin & Rousseau [1994], who implemented a parallel version of the algorithm in the spirit of Taillard [1993a], but no noticeable improvements were observed.

The algorithm just described was tested on Solomon's problems [1987]. These are made up of several sets. Problems $R1$ and $R2$ are *random* problems in which vertices are randomly generated in the plane according to a uniform distribution. In problems $C1$ and $C2$, customers are clustered. Problems $RC1$ and $RC2$ contain a mix of randomly distributed and clustered customers. In each instance there are 100 customers and the distance matrix satisfies the triangle inequality. The main computational results are summarized in Table 9.3. All reported results are average values over the number of instances. With respect to Solomon's solutions, which were used as a starting point (see Step 1), the algorithm of Potvin et al. [1996] improves route durations by average percentages varying between 14.4% and 33.2%, even when only one pass is executed. The optimal solutions were known in six cases (two random instances and four clustered instances). For the four clustered instances the values of m^* and f^* found by the heuristic were optimal; for the two random instances the value of m^* was one unit above the optimum and the value of f^* was within 2.7% of the optimum.

In closing this section, it is worth mentioning, that tabu search algorithms have been successfully applied to the solution of complex VRPs arising in practice. A first example is described by Semet & Taillard [1993]. This problem is concerned with the distribution of goods to 45 stores in Switzerland. Potential savings of 10–15% with respect to the current solution were reported. In a second example, Rochat & Semet [1994] solved a pet food and flour distribution problem to approximately 100 farms and stores. The difficulty in this case was the determination of feasible solutions which had become exceedingly difficult or impossible to obtain in practice. Using tabu search, the authors generated feasible solutions in all cases and also achieved average savings as large as 16.6%.

4 GENETIC ALGORITHMS

Evolution algorithms were originally randomized search techniques aimed at simulating the natural evolution of populations of asexual species [Fogel, Owens & Walsh, 1966]. Holland [1975] extended this model by allowing the recombination of pieces of information taken from two parents to create new offspring. These algorithms were called *genetic algorithms*, and the recombination operator proved to be a fundamental ingredient in the success of this search technique.

Genetic algorithms operate on *populations* of solutions encoded as *chromosomes*, typically bitstrings. To describe a genetic algorithm, consider a population $X^t = \{x_1^t, \ldots, x_N^t\}$, on which is defined a *fitness* function f. It is standard to maximize f, but as we are interested in solving VRPs, we will assume f is to be *minimized*. At iteration t, X^t is transformed into X^{t+1} by producing new solutions (*offspring*) from old ones (*parents*), and replacing these parents with their

offspring. This is accomplished so as to propagate the characteristics of good solutions from one generation to the next by biasing the selection process towards good solutions. Minor random mutations are applied to offspring in order to ensure a good diversity of solutions in the population. It is hoped that the process will evolve towards better solutions. Here follows a simple description of a genetic algorithm where k_1 is the number of generations, and $k_2 \leqslant N/2$ is the number of selections per generation.

At each of iteration $t = 1, \ldots, k_1$, apply k_2 times Steps 1 through 3, then apply Step 4.

- Step 1, *selection*. Randomly select two parents from X^t with a probability inversely related to the value of f.
- Step 2, *one-point crossover*. Create two offspring from the two parents by swapping a bitstring located after a randomly selected *cutpoint*, as shown in this example:

Parent 1	1	0	1	0	0
Parent 2	**0**	**0**	**1**	**1**	**1**
Offspring 1	1	0	**1**	**1**	**1**
Offspring 2	**0**	**0**	1	0	0

  Here, the cutpoint is selected after the second bit.

- Step 3, *mutation*. Apply a random mutation to each offspring by flipping bits with a small probability.
- Step 4, *new population*. Obtain X^{t+1} from X^t by removing the $2k_2$ worst solutions in X^t and replace them with the $2k_2$ offspring just created.

Historically, genetic algorithms were designed to solve numerical optimization problems rather than combinatorial optimization problems. Consequently, difficulties quickly arise when this problem-solving methodology is applied to routing problems like the TSP or the VRP. First, encoding a solution as a bitstring is not natural for these problems and more sophisticated representation schemes must be used to encode routes. A natural solution to this problem is to use a path representation. Two chromosomes of this type are shown below. They encode two different routes on vertices 1 to 8, namely (2, 3, 5, 6, 4, 1, 7, 8) and (1, 4, 2, 3, 6, 5, 8, 7):

Chromosome 1	2	3	5	6	4	1	7	8
Chromosome 2	1	4	2	3	6	5	8	7

The departure from the bitstring looks appropriate at first glance, given that the path representation is a natural way to encode routes. However, it is not clear

whether some nice theoretical results derived for the bitstring representation, like the schema theorem of Holland [1975], generalize to this new representation, and there is no underlying theory to guarantee that the genetic algorithm performs an intelligent search of the solution space.

Another problem is that infeasible offspring are likely to be created by the one-point crossover operator. For example, if this operator is applied to the parent chromosomes 1 and 2 below, the following two offspring are created:

Parent 1	2	3	5	6	4	1	7	8
Parent 2	**1**	**4**	**2**	**3**	**6**	**5**	**8**	**7**
Offspring 1	2	3	**2**	**3**	**6**	**5**	**8**	**7**
Offspring 2	**1**	**4**	5	6	4	1	7	8

Neither of two offspring corresponds to a permutation of the vertices. The difficulty is that routing problems are pure sequencing problems: all chromosomes carry the same values, and differ only in the ordering of these values. Hence, specialized permutation operators must be developed in order to generate offspring. For example, a permutation operator known as the cycle crossover has been developed for the TSP [Oliver, Smith & Holland, 1987]. It generates feasible offspring by identifying a subset of vertices that occupy the same subset of positions in the two parent chromosomes. In the following example, the subset of vertices $\{3, 4, 6\}$ both occupy the subset of positions $\{2, 4, 5\}$ on parent chromosomes 1 and 2. Consequently, these vertices can be exchanged to generate two new feasible offspring. Several different permutation operators like this one have been designed for the TSP. A survey is provided by Potvin [1996].

Parent 1	2	3	5	6	4	1	7	8
Parent 2	**1**	**4**	**2**	**3**	**6**	**5**	**8**	**7**
Offspring 1	2	**4**	5	**3**	**6**	1	7	8
Offspring 2	**1**	3	**2**	6	4	**5**	**8**	**7**

The VRP with capacity constraints, distance constraints, or time windows is more difficult to encode on a chromosome than the TSP. First, a solution is now composed of multiple routes. Second, side constraints must be satisfied. Fitness functions with additional terms to penalize infeasible solutions, and repair operators to transform infeasible solutions into feasible ones have been proposed to address these problems [Richardson et al., 1989; Siedlecki & Sklansky, 1989; Michalewicz & Janikow, 1991]. However, these studies consider specific application domains quite remote from the VRP. Other research avenues have been explored for vehicle routing to overcome the encoding and constraint satisfaction problems. We describe these approaches in the next section.

The literature on genetic algorithms for VRPs is scarce, and the first papers on the subject appeared only in 1991. We now describe the three known approaches to this problem, considering in each case the VRP with capacity and time window constraints.

4.1 Hybridization with a greedy heuristic

The work of Blanton & Wainwright [1992, 1993] is representative of a well-known problem-solving approach to combinatorial optimization problems with side constraints. It is based on the hybridization of a genetic algorithm with a greedy insertion heuristic. Under this scheme, a chromosome is a sequence of vertices and its fitness is the objective value of the solution produced by the greedy heuristic. That is, the fitness of the sequence relates to the quality of the solution produced by the greedy heuristic. The greedy heuristic typically considers the vertices one by one in the sequence and generates a feasible solution to the problem via an iterative insertion procedure. Accordingly, the genetic algorithm is only concerned with the search for good sequences of vertices; it leaves the side constraints to the greedy heuristic. The genetic algorithm developed by these authors to solve the VRP with time windows is based on the GENITOR package [Whitley, 1989]. The authors describe two new permutation operators whose distinctive feature is the use of a global precedence relationship among the vertices to guide the recombination phase.

Since the problem involves time windows, it is preferable to insert vertex i before vertex j during the greedy insertion phase, if the time window at vertex i occurs before the time window at vertex j. Accordingly, the authors define a global precedence vector, where the vertices are sorted in increasing order of their time windows. This vector is then used to guide crossover and to push the customers with early time windows towards the front of the offspring. Two types of crossover operators are defined, but we describe Merge Cross # 1 (MX1) in greater detail.

First, the vertices on both chromosomes are considered in sequence, from the first position to the last. Here the first position on parent chromosomes 1 and 2 is occupied by vertices 2 and 1, respectively:

Precedence vector	**1**	**5**	**3**	**6**	**2**	**8**	**4**	**7**
Parent 1	2	3	5	6	4	1	7	8
Parent 2	<u>**1**</u>	<u>**4**</u>	<u>**2**</u>	<u>**3**</u>	<u>**6**</u>	<u>**5**</u>	<u>**8**</u>	<u>**7**</u>

Vertex 1 on parent 2 has precedence over vertex 2 on parent 1 (since vertex 1 is located before vertex 2 in the precedence vector), and is selected to occupy the first position on the offspring. Furthermore, vertices 2 and 1 are swapped on parent 1, in order to guarantee the creation of feasible offspring. At this point, the situation is as follows:

Parent 1	1	3	5	6	4	2	7	8
Parent 2	**1**	**4**	**2**	**3**	**6**	**5**	**8**	**7**
Offspring	1	X	X	X	X	X	X	X

The vertices at the second position are then compared; in this case, vertex 3 on parent 1 has precedence over vertex 4, and is selected to occupy the second position on the offspring. Vertices 3 and 4 are then swapped on parent chromosome 2. When all positions have been examined, the following offspring is created:

Offspring	1	3	5	6	4	2	8	7

Blanton & Wainwright [1993] also describe a second crossover operator MX2 based on similar ideas, as well as a simple mutation operator that randomly swaps the positions of two vertices on a single chromosome. The fitness of each chromosome is evaluated by the greedy heuristic, using the sequence provided by the chromosome. This heuristic takes the first m vertices of the sequence to initialize m routes. The remaining vertices are then considered in turn and inserted in a feasible position so as to minimize the extra distance traveled. Since the number of routes is fixed, it may not be possible to insert each vertex. Hence, solutions with unrouted vertices are accepted and the corresponding sequences are maintained in the population. However, these chromosomes carry a much worse fitness value than the feasible ones. The genetic algorithm is typically run several times with an increasing number of routes until feasible solutions emerge.

The algorithm was applied to four different problems of size 15, 30, 75, and 99. The first three problems are grid problems for which time windows and demands were randomly generated. The 99-vertex problem is a real problem provided by a retail distribution company. Comparisons are made between MX1, MX2, the edge recombination crossover [Whitley, Starkweather & Fuquay, 1989], the cycle crossover [Oliver, Smith & Holland, 1987], and the partially mapped crossover [Goldberg & Lingle, 1985]. On the three largest problems, MX1 and MX2 found feasible solutions with fewer routes than the other tested operators. However, no comparisons are provided with approximation algorithms taken from the operations research literature.

4.2 GIDEON

GIDEON is a genetic algorithm for VRPs with time windows and capacity constraints, based on a *cluster first–route second* strategy [Thangiah, Nygard & Juell, 1991; Thangiah, 1993; Thangiah & Gubbi, 1993]. The genetic algorithm is only applied during the clustering phase: a procedure called 'genetic sectoring' partitions the vertices into sector or clusters centered at the depot, as in the sweep algorithm [Gillett & Miller, 1974].

Each chromosome contains a set of binary numbers encoding the angles defining the sectors. Since the genetic algorithm manipulates bitstrings, the classical one-point crossover and bit mutation operators are used to generate new angles. As the population of angles evolves through the generations, the algorithm eventually converges to a set of angles associated with a good partitioning. The quality of the clusters and, consequently, the fitness of each chromosome, is determined by building a route within each cluster using a cheapest insertion heuristic. Since the genetic sectoring process is based on geographical considerations only, there is no guarantee that the sequencing phase will generate a feasible solution relative to the capacity and time window constraints. In contrast to the Blanton & Wainwright [1993] algorithm, infeasible vertices are inserted into the routes, but penalties for overload or time window violations are introduced into the fitness function. At the end, local improvement heuristics are applied to the best solution produced by the genetic search. The first improvement heuristic moves strings of one or two vertices from one route to another. The second heuristic swaps strings of one or two vertices between two different routes. In both cases the fitness function with penalties is used in order to encourage of a good feasible solution.

4.3 GENEROUS

GENEROUS is the genetic routing system of Potvin & Bengio [1996] for VRPs with time windows. This algorithm avoids the difficulties related to the encoding of a solution into a chromosome by applying the crossover and mutation operators on the solutions themselves. In this algorithm a new solution is created from two parent solutions 1 and 2 by linking the first customers on a route of parent 1 to the last customers on a route of parent 2, as in the 2-opt* exchange heuristic. The new route replaces the old one in parent solution 1. A second offspring solution can be created by inverting the roles of the parent solutions. Since the crossover operator involves two routes in two different solutions, the new solution is rarely a permutation of the vertices. Some vertices are typically duplicated and are found in two different routes, whereas other vertices are left out. Accordingly, a *repair operator* is applied to produce a feasible solution. First, duplicates are eliminated; this step is not particularly difficult since feasible routes remain feasible when customers are removed. Second, each unrouted vertex is added to the new solution by using a cheapest insertion criterion. If a vertex cannot feasibly be inserted, the offspring is then discarded. New parent solutions are selected for crossover until a full population of feasible solutions is created. In Solomon's test set about 50% of the offspring are infeasible. Consequently, about $2N$ offspring solutions must be created in order to produce a new population of size N.

Although this algorithm is computationally expensive, it has produced very good solutions on Solomon's test problems; these results are reported in Table 9.4, and comparisons are provided with GIDEON [Thangiah, 1993]. Note that the first objective is to minimize the number of routes in both cases. However, the

Table 9.4 Computational results of GIDEON and GENEROUS

		GIDEON			GENEROUS			Best tabu search solution value[e]
Problem type	Number of instances	m^*[a]	$f^*(f^*)$[b]	Time[c]	m^*[a]	$f^*(f^*)$[b]	Time[d]	f^*[f]
$R1$	12	12.8	1 298.8	99.9	12.6	1 296.8	932.0	
			(2 528.5)			(2 407.4)		2 390.7
$R2$	11	3.2	1 124.7	183.2	3.1	1 167.1	1 431.6	
			(2 514.4)			(2 216.7)		2 185.7
$C1$	9	10.0	892.1	89.6	10.0	838.0	870.0	
			(9 972.4)			(9 838.0)		9 850.2
$C2$	8	3.0	749.1	142.3	3.0	596.9	1 371.6	
			(9 789.4)			(9 596.9)		9 594.6
$RC1$	8	12.5	1 472.2	110.0	12.1	1 446.2	937.0	
			(2 651.1)			(2 509.9)		2 507.6
$RC2$	8	3.4	1.411.0	159.4	3.4	1 360.6	1 456.4	
			(2 725.8)			(2 461.6)		2 459.8

[a] Best number of vehicles.
[b] Best distance (best route duration).
[c] Seconds on a SOLBOURNE 5/802 computer.
[d] Seconds on a Silicon Graphics workstation.
[e] Potvin et al. [1996].
[f] Best route duration.

secondary objective differs. GIDEON minimizes the total distance, but GENEROUS minimizes the total route time. GENEROUS is computationally more expensive than GIDEON as it handles a population of solutions rather than a population of sector angles. In particular, local improvement heuristics are massively used within GENEROUS in the form of mutation operators. This is not the case for GIDEON; only the best routes produced by the genetic sectoring process are modified by local improvement procedures. On Solomon's problems the additional computational requirements of GENEROUS translate into the saving of seven additional routes.

5 NEURAL NETWORKS

Neural networks are learning devices composed of simple processing units or neurons that are richly interconnected like the neural cells in the brain. The connections are weighted by numerical values that encode the knowledge of the network. Starting from random weights, a neural network incrementally adjusts these values via a learning algorithm so as to improve its ability to perform a particular task. Neural networks were designed for learning tasks, not for solving combinatorial optimization problems. However, different neural network models have recently been applied to the TSP. They include the Hopfield–Tank model [Hopfield & Tank, 1985], the elastic net [Durbin & Willshaw, 1987], and the self-organizing map [Kohonen, 1988a], but so far results have not been competitive [Potvin, 1993]. Among the TSP models, the self-organizing map was recently applied to the VRP by two authors [El Ghaziri, 1991, 1993; Matsuyama, 1991]. The work of El Ghaziri is used in the following to illustrate this problem-solving methodology. Although the self-organizing map can handle spatial relationships among the vertices to find good tours on pure geographic problems, it is not easy to modify its design to take additional side constraints into account. Accordingly, the literature on the VRP is very limited and, to the best of our knowledge, neural networks have not yet been applied to the VRP with time windows.

5.1 Self-organizing maps

Self-organizing maps are instances of so-called competitive neural network models [Kohonen, 1988a]. They 'self-organize' by gradually adjusting the weights on their connections. A self-organizing map with two input units and three output units is depicted in Figure 9.1. Topological relationships are established among the output units and, in this example, the units are linked together to form a ring. In Figure 9.1 T_{11} and T_{21} denote the weights on the connections from the two input units I_1 and I_2 to output unit 1, and $T_1 = (T_{11}, T_{21})$ is the weight vector of output unit 1.

Assuming an ordered set of n input vectors in p dimensions and a self-organizing map with p input units and m output units on a ring, the algorithm for adjusting the weights is summarized as follows.

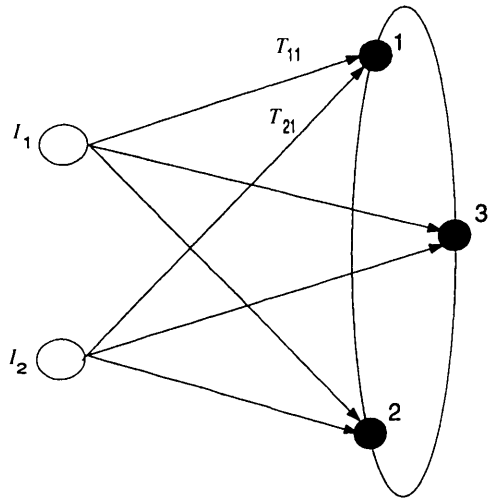

Figure 9.1 A self-organizing map with two input units and a ring of three output units

- Step 1, *initialization.* Set the connection weights to some initial values.
- Step 2, *competition.* Consider the next input vector I. If all input vectors have been considered, start again from the first input vector. Compute the output value o_j of each output unit as the weighted sum of its inputs, namely

$$o_j := \sum_{i=1}^{p} T_{ij} I_j \quad (j = 1, \ldots, m).$$

  The winning output unit j^* is the unit with maximal output.

- Step 3, *weight adjustment.* Modify the connection weights of each output unit, as follows:

$$T_j := T_j + f(j, j^*)(I - T_j),$$

  where $T_j = (T_{ij})(i = 1, \ldots, p)$ is the weight vector of unit j, and f is a function of j and j^*.

Repeat Steps 2 and 3 until the weight vectors stabilize. At the end, each input vector is assigned to the output unit that maximizes the weighted sum of Step 2.

This weight learning algorithm deserves some additional comments. In Step 3 the range of function f is the interval [0, 1]. Typically, f is a decreasing function of the lateral distance between units j and j^* on the ring (e.g., if the two units are adjacent, the lateral distance is 1; if there are r units on the ring between the two units, the lateral distance is $r + 1$). Accordingly, the weight vector of the winning unit j^*, and the weight vectors of the units that are close to j^* on the ring all move towards the input vector I, but with decreasing magnitude as the lateral distance

to the winning unit increases. Quite often, f is also dynamically modified during the learning algorithm in order to progressively reduce the magnitude of the move of the weight vectors. During the first iterations, the closest units to the winning unit on the ring significantly move their weight vector towards the current input vector, so as to 'follow' the winning unit and be assigned to input vectors that are in the neighborhood of the current vector. During the last iterations, only the weight vector of the winning unit significantly moves towards the current input vector in order to fix the final assignment.

Self-organizing maps are used to find topological mappings from a high-dimensional input space to a low-dimensional output space. In Figure 9.1, for example, each two-dimensional input vector is mapped onto a one-dimensional output, namely the position in the ring of the unit that is finally assigned to this input vector. The transformation should preserve the topological relationships among the p input vectors. That is, two input vectors which are close in the two-dimensional input space should be assigned to units that are close on the ring.

In the VRP context, the input vectors are the two-dimensional coordinate vectors of the vertices. Several different rings are defined, each standing for a different route. The weight learning algorithm is also slightly modified. In Step 2 of the classical algorithm, the winning output unit is the unit whose weight vector is the most highly correlated with the current input vector (maximization of the inner product of the weight vector and input vector). For Euclidean VRPs the definition of the winning unit is modified to the unit whose weight vector is the closest in Euclidean distance to the current coordinate vector. In fact, maximizing the inner product is equivalent to minimizing the Euclidean distance when all vectors have the same norm.

Given that multiple rings are defined, the winning unit is determined among all units in all rings. Then the weight vectors of the units located on the ring of the winning unit are moved towards the current coordinate vector. At the end of the algorithm, each vertex (coordinate vector) is assigned to the closest unit (weight vector). Through this assignment, the ordering of the units in each ring define an ordering of the vertices in each route. Moreover, since neighboring units on a given ring tend to migrate towards neighboring vertices in the Euclidean plane, short routes are expected to emerge.

Figure 9.2 depicts the evolution of the rings in the two-dimensional Euclidean plane during the application of the weight learning algorithm. Vertices are the large white nodes, and units are the small black nodes on the rings. The coordinates of each unit in Figure 9.2 correspond to its current weight vector. Starting from the initial weight vectors in Figure 9.2(a) (initial position of the units in each ring), these weight vectors are progressively modified by the learning algorithm until there is a weight vector sufficiently close to each vertex. The final assignment defines an ordering of the vertices within each route. The number of weight vectors or units must obviously be greater than or equal to the number of vertices. Figure 9.2 also shows that the search for a solution takes place in the two-dimensional Euclidean plane. At the start, when the weight vectors are far

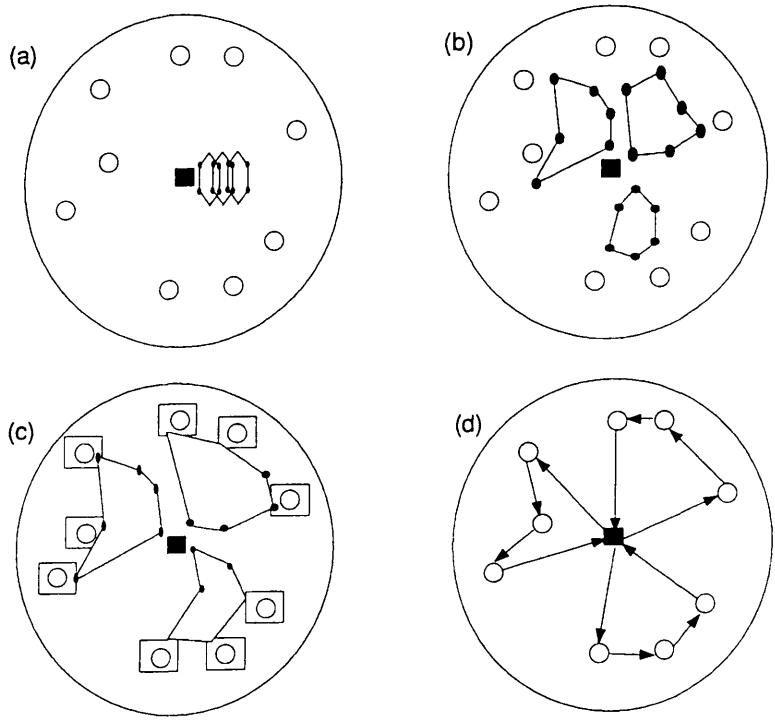

Figure 9.2 (a, b, c) Evolution of three rings; (d) the final solution

from the vertices, no solution to the problem can be inferred. However, as the weight vectors migrate towards the vertices, a solution progressively emerges. The final solution is determined at the end, during the final assignment of the vertices to the units on the rings. Section 5.2 will describe the self-organizing map of El Ghaziri [1993] in greater detail.

5.2 The hierarchical deformable net

The hierarchical deformable net of El Ghaziri [1993] involves a competition at two different levels: a deterministic competition among the units within each ring and a stochastic competition among the rings. The number of rings is fixed and this model is applied with an increasing number of rings until a feasible solution is found. The model is summarized as follows.

- Step 1, *initialization.* Randomly order the vertices. Then create one ring per route. Initialize the parameters h, η and G.

Repeat Steps 2 through 7 until there is a weight vector sufficiently close to each vertex.

- Step 2, *vector selection.* Select the coordinate vector I of the next vertex v, and set it as the current input vector (if all vertices are done, restart at the first vertex).
- Step 3, *competition within each ring.* Determine the winning unit u_{j^*k} within each ring r_k. The weight vector of this unit is the closest to the current coordinate vector I.
- Step 4, *competition among the rings.* Assign the following winning probability to each ring r_k:

$$p(r_k) := f(r_k, h) \Big/ \sum_{l=1}^{m} f(r_l, h),$$

$$f(r_k, h) := \mathrm{e}^{-d(T_{j^*k}, I)/h} / (1 + \mathrm{e}^{-\Delta Q/h}),$$

  where h is a parameter of the algorithm, m is the number of rings, $d(T_{j^*k}, I)$ is the Euclidean distance between the coordinate vector I and the weight vector T_{j^*k} of the winning unit u_{j^*k} on ring r_k, ΔQ is the difference between the capacity of the vehicle and the total demand on ring r_k (including vertex v).
- Step 5, *ring selection.* Select the winning ring r_{k^*} using a selection probability for each ring r_k proportional to $p(r_k)$. Vertex v is assigned to unit $u_{j^*k^*}$.
- Step 6, *weight adjustment.* Move the weight vector T_{jk^*} of each unit j on the winning ring r_{k^*} towards the coordinate vector I of vertex v as follows:

$$T_{jk^*} := T_{jk^*} + \eta f(j, j^*, k^*)(I - T_{jk^*}),$$

$$f(j, j^*, k^*) := \tfrac{1}{2}\sqrt{2}\,\mathrm{e}^{-dL(j, j^*, k^*)^2/G^2},$$

  where $dL(j, j^*, k^*)$ is the lateral distance on ring r_{k^*} between unit j and the winning unit j^*.
- Step 7, *parameter adjustment.* Slightly decrease the value of parameters h, η, and G.

In Step 4 parameter h is used to modify the probability distribution of the rings. When h is high the selection probability of each ring is about the same. When h is low the selection probability quickly increases as the winning unit on a ring gets closer to the current vertex. Moreover, the selection probability of a ring that violates the capacity constraint (with the addition of the current vertex) quickly reduces to zero. Therefore, the value of parameter h is slowly reduced in order to favor convergence towards a good feasible solution. It is worth noting that the assignment of a given vertex to a particular unit in Step 5 can change from one pass to another through the vertices, because the weight vector of each unit converges to a particular area in the Euclidean plane. At this point, the weights do not move much and the assignments of the vertices to the units stabilize. The weight update equations in Step 6 are also very important to explain this behavior. Through these equations, all units on the winning ring move towards the current vertex, but the magnitude of the move is much larger for the winning unit and its closest neighbors on the ring. Moreover, the magnitude of the move is small for small values of the parameters G and η. Accordingly, the value of these parameters is slowly reduced in order to obtain convergence to a stable state.

Table 9.5 Computational results for the hierarchical deformable net

		Hierarchical deformable net	Best known solution value
Problem	n	f^{*a}	f^{*a}
1	50	545.1	524.61[b,c,d]
3	100	911.2	826.14[b,c]
4	150	1 133.2	1 028.42[b]
5	199	1 421.5	1 298.79[b]
11	120	1 133.4	1 042.11[b,c]
12	100	826.6	819.56[b,c]

[a] Best distance.
[b] Taillard [1993a].
[c] Gendreau, Hertz & Laporte [1994].
[d] Osman [1993].

El Ghaziri [1993] has applied this model to problems 1, 3, 4, 5, 11, and 12 of Christofides et al. [1979] using *real* distances (Table 9.5). Although the solutions are good, better solutions can be found with other approximation algorithms. Computing times are not reported in this reference. However, the same author reports in a different reference [El Ghaziri, 1991] results for problems 1, 11, and 12, with respective objective function values of 586, 1133, and 836, and respective computing times of 25, 121, and 81 seconds in a SPARC workstation. Finally, Matsuyama [1991] uses a similar ideas to solve a VRP of size 532. This problem is derived from the classical TSP of Padberg & Rinaldi [1987]. Matsuyama adds capacity constraints to this problem, as well as incompatibility constraints between vehicles and vertices. Since no comparisons are provided with other heuristics, it is not possible to assess the quality of this approach.

6 CONCLUSION

The impressive computational results obtained with some of the local search algorithms developed for vehicle routing problems clearly demonstrate the potential of such methods for tackling complex, difficult combinatorial optimization problems. A more thorough analysis reveals, however, that the various approaches are not equally successful and that a fair amount of problem-specific knowledge must be embedded in any algorithm in order to realize the full potential of these techniques. This last point is well illustrated by a comparison of early simulated annealing or tabu search implementations with later algorithms using the same general approach. In both cases the early algorithms amount to straightforward applications of the general principles of simulated annealing or tabu search and produce fairly mediocre solutions; later simulated annealing and tabu search algorithms are significantly more sophisticated and much more difficult to implement, but the computational results are excellent. This discrep-

ancy in the performance of algorithms based on the same general philosophy can probably be attributed to two critical factors: the neighborhood structure and the scope of the search strategy. With regard to neighborhood structure, note that the best implementations consider at each iteration fairly large neighborhoods, which contain solutions differing significantly from the current one. As for the search strategy, there is always a danger that a local search algorithm will spend most of its time in a restricted subset of the solution space, thus leaving large portions of this space unexplored; it is therefore important that the procedure incorporate mechanisms allowing for sufficient search diversification.

Returning to the comparison between the various approaches themselves, it seems fair to say that neural networks do not seem well suited to the solution of VRPs. This is not very surprising, since these methods were not originally devised for optimization purposes. In this situation the fundamental analogy upon which the approach relies is probably too remote from the problem to be tackled. The best simulated annealing algorithms have proved reasonably successful, but they are not very robust and they do not perform as well as the best available methods. Tabu search has been applied to numerous versions of the VRP. In all cases the best tabu search implementations have produced excellent solutions. On the other hand a fair amount of ingenuity is required to cast VRPs in the framework of genetic algorithms. As a consequence there have been fewer attempts to use genetic algorithms to solve VRPs and there is much less computational evidence available to assess their potential. But the results obtained for the VRP with time windows are highly promising. This may be because genetic algorithms work with population of solutions, rather than a single solution, thus achieving a wider exploration of the solution space than other methods.

In the near future there should be many more attempts to apply genetic algorithms and tabu search to other variants of the VRP. For example, tabu search has been successfully applied to the m-TSP with minmax objective, a problem with applications in production planning [França et al., 1995]. Other applications of this technique to the VRP are under way and preliminary results are very encouraging. On the methodological side, we should expect the emergence of several research projects aimed at combining the best features of existing methods in new hybrid local search algorithms.

ACKNOWLEDGMENTS

This research was partially funded by the Canadian Natural Sciences and Engineering Research Council (grants OGP0038816, OGP0039682, and OGP0036662) and by the Quebec Government (FCAR grant 93ER0289). This support is gratefully acknowledged.

Local Search in Combinatorial Optimization
Edited by E. Aarts and J. K. Lenstra
© 1997 John Wiley & Sons Ltd.

10

Vehicle routing: handling edge exchanges

Gerard A.P. Kindervater
Erasmus University, Rotterdam

Martin W.P. Savelsbergh
Georgia Institute of Technology, Atlanta

1 INTRODUCTION 337
2 EDGE-EXCHANGE NEIGHBORHOODS 339
2.1 Edge exchanges for a single route 339
2.2 Edge exchanges for multiple routes 341
3 TECHNIQUES FOR HANDLING SIDE CONSTRAINTS 344
3.1 Preprocessing 346
3.2 Updating mechanisms 347
3.3 Lexicographic search 348
3.4 Computational evidence 353
4 PARALLEL IMPLEMENTATIONS 356
5 MODERN HEURISTICS 360

1 INTRODUCTION

Physical distribution management presents a variety of decision-making problems at the three levels of strategic, tactical, and operational planning. Decisions relating to the location of facilities (plants, warehouses, or depots) may be viewed as strategic, whereas problems of fleet size and fleet mix determination can be termed tactical. On the operational level, two problems prevail: the routing of capacitated vehicles through a collection of customers to pick up or deliver goods, and the scheduling of vehicles to meet time or precedence constraints imposed upon their routes.

The importance of effective and efficient distribution management is evident from its associated costs. Physical distribution management at the operational level, considered in this chapter, is responsible for an important fraction of the total distribution costs. Small relative savings in these expenses could already account for substantial savings in absolute terms. The significance of detecting these potential savings has become increasingly apparent due to the escalation of the costs involved, such as capital and fuel costs and driver salaries.

Not surprisingly, there is a growing demand for planning systems that produce economical routes. Although cost optimization is often the primary objective for purchasing computerized systems for physical distribution management, there are other benefits that should not be underestimated. The introduction of such systems enables companies to maintain a higher level of service towards their customers, it makes them less dependent on their planners, it supplies better management information facilities, and it makes the conduct of work faster and simpler.

Besides its obvious practical importance, physical distribution management also provides some fascinating basic models, such as the *traveling salesman problem* (TSP) and the *vehicle routing problem* (VRP). Consequently, researchers from the fields of operations research, mathematics, and computer science have spent considerable effort in trying to develop solution approaches for these problems. For an extensive survey of models and solution methods in vehicle routing and scheduling see Bodin et al. [1983], Laporte & Nobert [1987], and Golden & Assad [1988].

The past decade has seen enormous theoretical as well as practical advances. Certainly one of the most important advances is the incorporation of real-world characteristics, such as time windows and precedence relations, into the basic models.

In the *vehicle routing problem with time windows* [Desrochers et al., 1988; Solomon & Desrosiers, 1988] a number of vehicles, each with a given capacity, is located at a single depot and must serve a number of geographically dispersed customers. Each customer has a given demand and must be served within a specified time window. The objective is to minimize the total cost of travel.

In the *pickup and delivery problem* [Savelsbergh & Sol, 1995] there is a single depot, a number of vehicles with given capacities, and a number of customers with given demands. Each customer must be served, which means that goods must be picked up at the customer's origin during a specified time window, and delivered at the customer's destination during another specified time window. The objective is to minimize total travel cost. The special case in which all customer demands are equal is called the *dial-a-ride problem.* It arises in transportation systems for the handicapped and the elderly. In these situations the temporal constraints imposed by the customers strongly restrict the total vehicle load at any moment, and the capacity constraints are of secondary importance. The cost of a route is a combination of travel time and customer dissatisfaction.

An important consideration in the formulation and solution of vehicle routing problems is the required computational effort associated with various solution techniques. Virtually all vehicle routing problems belong to the class of NP-hard problems. This indicates that it is difficult to solve even small instances of a problem to optimality with a reasonable computational effort. As a consequence, when we have to solve real-life problems, we should not insist on finding an optimal solution, but on finding an acceptable solution within an acceptable amount of computation time. To accomplish this we have to resort to approximation algorithms.

Approximation algorithms for vehicle routing problems usually have two phases: a *construction phase*, in which an initial feasible solution is constructed, and a *local search phase*, in which an attempt is made to improve that initial solution by repeatedly searching a specified neighborhood for a better one.

Most neighborhoods that are being used in the context of vehicle routing problems are based on the well-known k-exchanges, which were originally proposed for the traveling salesman problem; see Chapter 8 of this book. The techniques developed for the TSP have to be extended in order to handle multiple routes and various side constraints.

Traditional k-exchange algorithms can obviously be used to improve a vehicle routing solution by considering the routes one at a time. However, the multiple-route structure offers additional opportunities. In Section 2 we first focus on edge-exchange neighborhoods for the single-route situation and then on neighborhoods that have been proposed to exploit the multiple-route structure.

Real-life vehicle routing problems are complicated by various side constraints, such as precedence relations between vertices, collections or deliveries at vertices, and time windows at vertices. The traditional k-exchange algorithms clearly have to be modified to ensure the feasibility of a route after an exchange has been made. Unfortunately, straightforward extensions lead to prohibitive computation times. In Section 3 we focus on proposed techniques to handle these side constraints efficiently and present several computational experiments we have conducted to evaluate their benefits.

Because efficiency is a key consideration in the development of local search methods, we also investigate the potential of parallel computations and discuss our findings in Section 4.

This chapter concentrates on the standard iterative improvement strategy. In Section 5 we briefly recall the application of search strategies such as simulated annealing and tabu search to vehicle routing, the subject of Chapter 9.

2 EDGE-EXCHANGE NEIGHBORHOODS

Most iterative improvement methods that have been applied to vehicle routing problems are edge-exchange algorithms. We distinguish between edge-exchange neighborhoods for a single route and edge-exchange neighborhoods for multiple routes. We will assume that the routes start and end at the depot, and that there are $n-1$ customers to be served. We will use the notation pre_i and suc_i to denote the predecessor and successor of vertex i in the current route.

2.1 Edge exchanges for a single route

The edge-exchange neighborhoods for a single route are sets of tours that can be obtained from an initial tour by replacing a set of k of its edges by another set of k edges. Such replacements are called *k-exchanges*, and a tour that cannot be improved by a k-exchange is said to be *k-optimal*. Verifying k-optimality requires $O(n^k)$ time.

The k-exchange neighborhoods depend on a fixed value of k. As k increases, they become more powerful, but the time needed to search them increases too. Lin & Kernighan [1973] developed a variable-depth exchange procedure by dynamically determining the cardinality of the set of edges to be replaced, thus finding a good balance between computational effort and solution quality.

For two reasons, we restrict our attention to k-exchanges for $k \leqslant 3$. First, k-exchanges for $k > 3$ are seldom used in iterative improvement methods for vehicle routing and scheduling problems. Second, k-exchanges for $k \leqslant 3$ suffice to illustrate the techniques we want to discuss. In the end, it should be clear how the presented techniques can be extended to the general case as well as to the Lin–Kernighan algorithm. The following two neighborhoods will be considered in detail:

1. Try to improve the tour by replacing two of its edges by two other edges, i.e., a 2-exchange, and iterate until no further improvement is possible. An example of a 2-exchange in given in Figure 10.1.
2. Try to improve the tour by relocating a chain of l consecutive vertices, and iterate until no further improvement is possible. This type of modification of the tour involves the replacement of a set of three edges, with a restriction on the choice of edges. Hence, the neighborhood is a subset of the 3-exchange neighborhood. Since the procedures involved are conceptually the same for all values of l, we will restrict ourselves to $l = 1$. This type of exchange was first

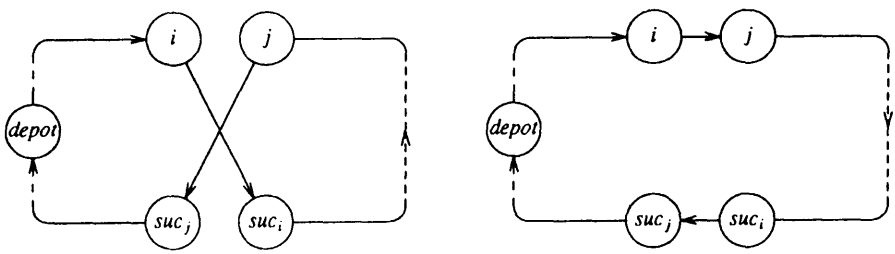

Figure 10.1 A 2-exchange

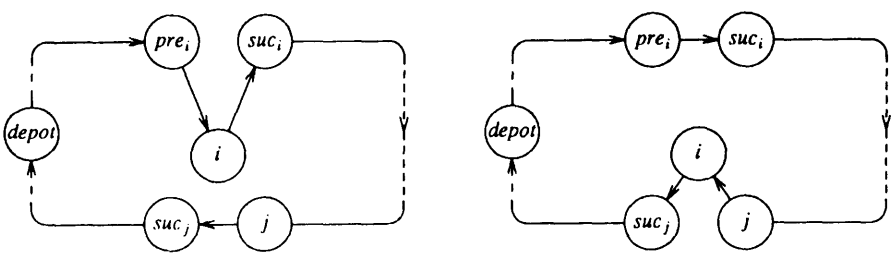

Figure 10.2 A forward Or-exchange

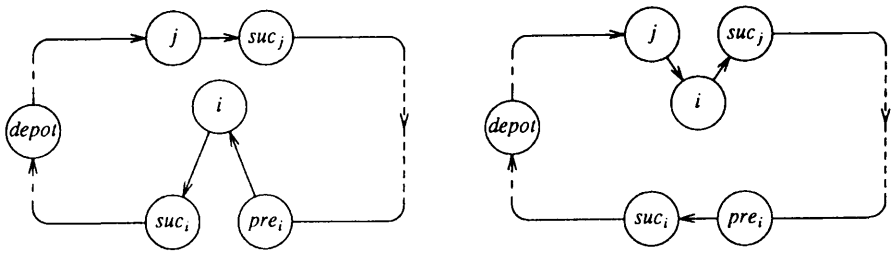

Figure 10.3 A backward Or-exchange

considered by Or [1976], and is therefore called an *Or-exchange*. We speak of a *forward* Or-exchange if the vertex is moved to a place further on the tour (Figure 10.2), and a *backward* Or-exchange otherwise (Figure 10.3). Testing optimality over the Or-exchange neighborhood requires $O(n^2)$ time.

2.2 Edge exchanges for multiple routes

Two types of decisions have to be made to obtain a solution to a vehicle routing problem: *assignment* decisions to determine which vehicle will serve which customers, and *routing* decisions to determine in which order the customers assigned to a vehicle will be visited. The local search methods in the previous section try to improve the routing decisions. Another possibility is to improve the assignment decisions. Improving assignment decisions may even be more effective when we consider the fact that in most instances of the VRP the number of customers per vehicle is fairly small, i.e., the resulting instances of the TSP are fairly simple. It is uncommon to find instances where the number of customers per vehicle exceeds 30. More often we are in a situation where the number of customers per vehicle is much less, in the order of 5–15. In that case it is unrealistic to hope for major improvements when routing decisions are revised. However, the total number of customers is usually large. Therefore, there is much more potential for improvement when assignment decisions are revised. In view of this, it is surprising to see how little is known about local search methods that revise assignment decisions.

The three basic k-exchange neighborhoods for the VRP [Savelsbergh, 1992] relocate vertices between *two* routes. The neighborhoods are chosen such that testing optimality over the neighborhood requires $O(n^2)$ time. For presentational convenience, we will first describe relocations of single vertices, and we will split the depot in the figures.

- *Relocation.* The edges $\{pre_i, i\}$, $\{i, suc_i\}$, and $\{j, suc_j\}$ are replaced by $\{pre_i, suc_i\}$, $\{j, i\}$, and $\{i, suc_j\}$, i.e., vertex i from the origin route is placed in the destination route. A relocation is pictured in Figure 10.4. Note that the vertex being relocated is inserted between two consecutive vertices on the destination

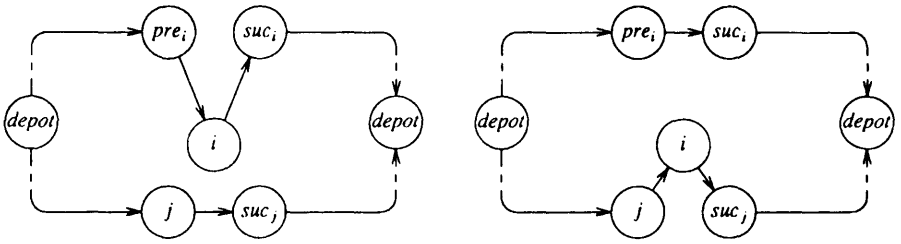

Figure 10.4 A relocation

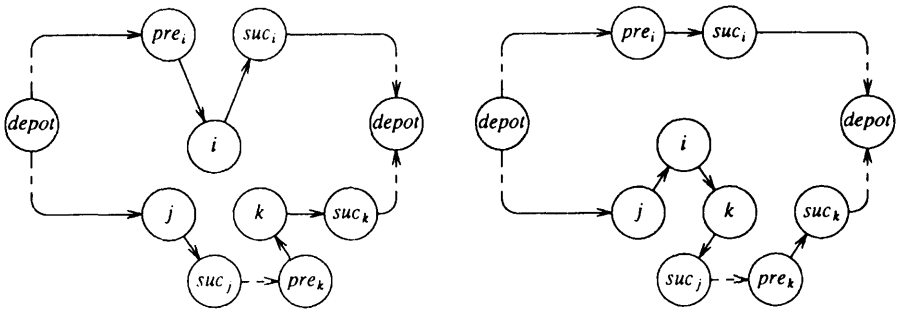

Figure 10.5 An insertion between two nonconsecutive vertices

route. Gendreau, Hertz & Laporte [1994] propose an extension of the relocate neighborhood in which a vertex can also be inserted between the two vertices on the destination route that are nearest to it, even if these vertices are not consecutive. Insertions of this type have been used with great success in an insertion algorithm for the TSP [Gendreau, Hertz & Laporte, 1992]. An example is shown in Figure 10.5. Observe that it is no longer obvious how to obtain a tour again after the edges $\{j, i\}$ and $\{i, k\}$ have been added. One possibility is to delete edges $\{j, suc_j\}$ and $\{pre_k, k\}$ and to relocate the path $\{suc_j, \ldots, pre_k\}$.

- *Exchange*. The edges $\{pre_i, i\}$, $\{i, suc_i\}$, $\{pre_i, j\}$, and $\{j, suc_j\}$ are replaced by $\{pre_i, j\}$, $\{j, suc_i\}$, $\{pre_j, i\}$, and $\{i, suc_j\}$, i.e., two vertices from different routes are simultaneously placed into the other routes. An exchange is pictured in Figure 10.6.
- *Crossover*. The edges $\{i, suc_i\}$ and $\{j, suc_j\}$ are replaced by $\{i, suc_j\}$ and $\{j, suc_i\}$, i.e., the crossing links of two routes are removed. A crossover is pictured in Figure 10.7. As a special case it can combine two routes into one if edge $\{i, suc_i\}$ is the first one on its route and edge $\{j, suc_j\}$ the last one on its route, or vice versa. Note that, if a crossover is actually performed, the last part of either route will become the last part of the other.

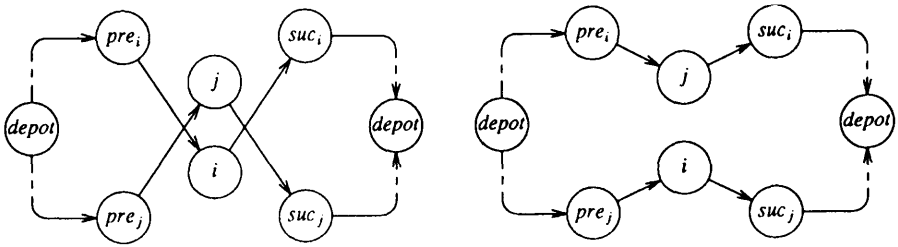

Figure 10.6 An exchange

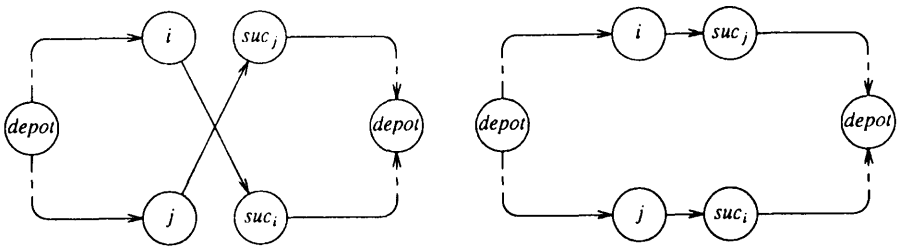

Figure 10.7 A crossover

Some of the iterative improvement methods described above can easily be extended to larger neighborhoods by the introduction of paths instead of vertices. This does not influence the worst-case time complexity of testing optimality, as long as the length of the paths is fixed. Figure 10.8 illustrates possible extensions.

All of the above neighborhoods can be viewed as special cases of cyclic transfers proposed for vehicle routing problems by Thompson & Psaraftis [1993]. The central idea behind cyclic transfers is to attempt to improve a solution to a partitioning problem by transferring small numbers of elements among cycles of sets. More formally, let $\{I^1, I^2, \ldots, I^m\}$ be a partition of a set of elements S, and let ρ be a cyclic permutation of a subset of $(1, 2, \ldots, m)$. Then a cyclic transfer is the simultaneous transfer of elements from I^j to $I^{\rho(j)}$ for each j.

Several special cases of cyclic transfers are of interest. Cyclic k-transfers transfer exactly k elements from I^j to $I^{\rho(j)}$ for each j and some integer k. The flexibility of k-transfers can be further increased by allowing dummy elements to be transferred, i.e., cyclic transfers are obtained in which the number of elements to be transferred from I^j to $I^{\rho(j)}$ for each j is bounded from above by k. If in addition the cyclic permutation has cardinality b, the set of transfers is called a b-cyclic k-transfer. Observe that an exchange is a 2-cyclic 1-transfer and a relocation is a 2-cyclic 1-transfer in which a dummy element is transferred from one of the two sets.

The cost of a cyclic transfer is the change in the objective function caused by the cyclic transfer, and a solution is cyclic transfer optimal if all cyclic transfers have

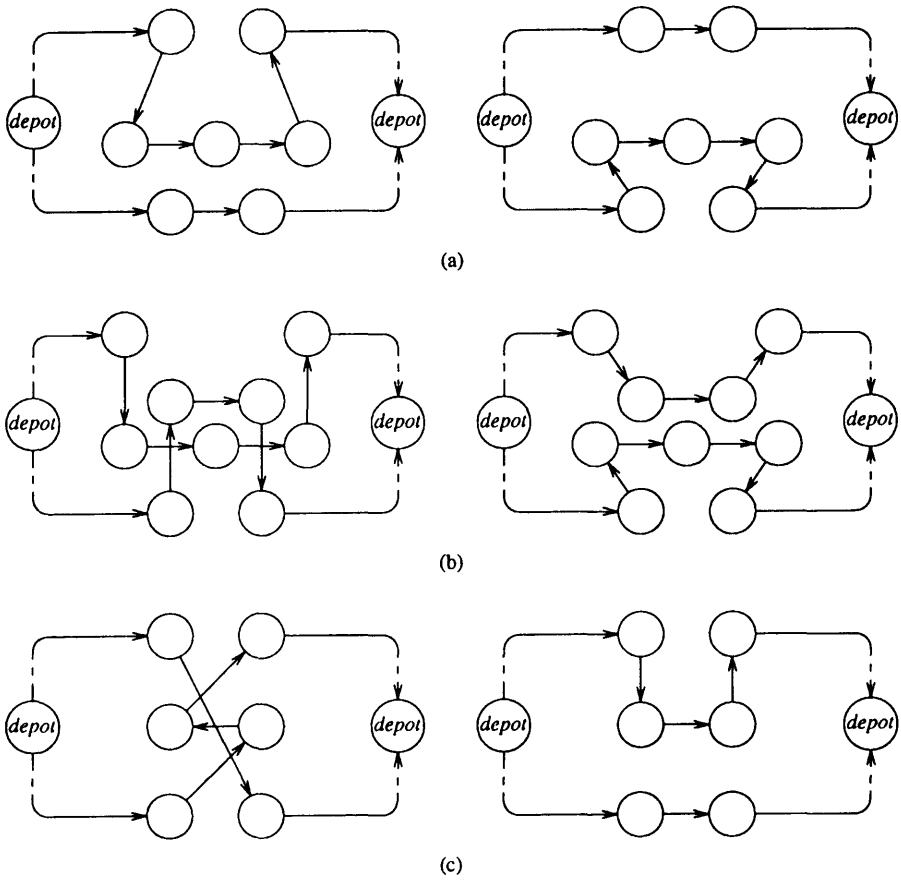

Figure 10.8 Possible extensions of the neighborhoods: (a) relocation of a path, (b) exchange of two paths, (c) a crossover plus 2-exchange

nonnegative cost. Thompson & Orlin [1989] develop a general methodology for searching cyclic transfer neighborhoods. They transform the search for negative cost cyclic transfers into a search for negative cost cycles on an auxiliary graph. Due to the computational intensity of searching cyclic transfer neighborhoods only b-cyclic k-transfers with small numbers b and k are considered, i.e., $b \leqslant 3$ and $k \leqslant 2$.

3 TECHNIQUES FOR HANDLING SIDE CONSTRAINTS

Real-life vehicle routing problems are complicated by various side constraints. The main thrust of research on modifying edge-exchange algorithms to handle these side constraints has not focused on how to incorporate the side constraints,

which is trivial, but on how to do this efficiently, which is nontrivial. We restrict our discussion to the following side constraints:

- precedence relations between vertices;
- collections or deliveries at vertices;
- time windows at vertices.

We will consider each of these side constraints separately and we will describe the techniques to efficiently handle side constraints only for edge-exchange neighborhoods involving a single route. However, it should be clear in the end that the techniques presented can easily be used to produce an edge-exchange algorithm that can handle any combination of side constraints for edge-exchange neighborhoods involving either a single route or multiple routes.

We start by introducing the three variants of the TSP for which we will study the exchange procedures in more detail.

In the *TSP with precedence constraints*, we are given, in addition to the travel times d_{ij} between each pair of vertices (i, j), precedence constraints specifying that some pairs of vertices have to be visited in a prescribed order. The *single-vehicle dial-a-ride problem*, where a single vehicle has to pick up and deliver n customers, is an example of the TSP with precedence constraints. Given a precedence-related pair of vertices $u \to v$, we will often refer to u as the origin and to v as the destination.

In the *TSP with collections and deliveries*, we are given, in addition to the travel times between vertices, for each vertex i an associated load q_i that is either positive or negative depending on whether the load has to be collected or delivered. The salesman uses a vehicle with fixed capacity Q.

In the *TSP with time windows*, we are given, in addition to the travel times, for each vertex i a time window on the departure time, denoted by $[e_i, l_i]$, where e_i specifies the earliest service time and l_i the latest service time. Arriving earlier than e_i introduces a waiting time at vertex i; arriving after l_i leads to infeasibility. We will use the following notation: A_i will denote the arrivial time at vertex i, D_i will denote the departure time at vertex i, and W_i will denote the waiting time at vertex i.

For notational convenience we will assume that the current tour is given by the sequence $(1, 2, \ldots, i, \ldots, n, n+1)$, where the origin 1 and the destination $n+1$ denote the same vertex, and i represents the ith vertex. We also assume that a 2-exchange replaces two edges $\{i, i+1\}$ and $\{j, j+1\}$, with $j > i$, by the edges $\{i, j\}$ and $\{i+1, j+1\}$, and that an Or-exchange involves the substitution of edges $\{i-1, i\}$, $\{i, i+1\}$ and $\{j, j+1\}$ by $\{i-1, i+1\}$, $\{j, i\}$ and $\{i, j+1\}$.

The main difficulty with the use of exchange procedures in the TSP with side constraints is testing the feasibility of an exchange, as opposed to the TSP where one only has to test whether the exchange is profitable and one does not have to bother about feasibility. A 2-exchange, for instance, will reverse the path $(i+1, \ldots, j)$, which means that one has to check the feasibility of at least all the vertices on the new path with respect to those constraints. In a straightforward implementation this requires $O(n^2)$ time in the TSP with precedence constraints, and $O(n)$ time in the TSP with collections and deliveries and the TSP with time windows.

By applying *preprocessing techniques* [Psaraftis, 1983; Solomon, Baker & Schaffer, 1988], *tailored updating mechanisms* [Solomon, Baker & Schaffer, 1988], and *lexicographic search strategies* [Savelsbergh, 1986, 1990, 1992], several researchers have been able to incorporate the various side constraints with an acceptable or even without an increase in computation times.

Before focusing on improvement methods of a given tour, it is worthwhile to mention the influence of the side constraints on the construction of an initial feasible tour. In the TSP with precedence relations, an initial tour can be obtained in polynomial time by visiting the vertices in topological order. In the other two cases, the problem of deciding whether a feasible tour exists is NP-complete [Savelsbergh, 1986, 1992]. Constructing a feasible initial tour may therefore constitute a problem. In the context of physical distribution management, however, the instances of the TSP stem most of the time from some constructive VRP heuristic, which also provides an initial feasible tour.

3.1 Preprocessing

Psaraftis [1983] was the first to study k-exchange procedures for a constrained variant of the TSP. He studied 2-exchanges and Or-exchanges in the context of the single-vehicle dial-a-ride problem. The dial-a-ride problem is a TSP with precedence constraints in which each vertex is related to precisely one other vertex.

First, we examine the 2-exchanges. A straightforward test for feasibility in this case requires $O(n^2)$ time. This can be seen as follows. The only way that a 2-exchange can be infeasible is if there is at least one precedence-related pair of vertices on the segment of the tour that is reversed. The simplest way to find out whether such a pair of vertices exists is to examine all pairs of vertices in the segment. This requires $O(n^2)$ time and would lead to an overall complexity for verification of 2-optimality of $O(n^4)$.

Psaraftis shows how this can be reduced to $O(n^2)$ by performing a *screening* procedure at the beginning of the algorithm which determines the feasibility of every possible 2-exchange. Information from the screening procedure is stored in a feasibility matrix to be examined during the execution of the actual algorithm.

The screening procedure is based on the following observation: if $firstdes_i$ denotes the first destination of a precedence constraint for which both origin and destination lie beyond i, then the exchange of $\{i, i+1\}$ and $\{j, j+1\}$ with $\{i, j\}$ and $\{i+1, j+1\}$ is feasible if and only if $j < firstdes_i$. The screening procedure first computes the values $firstdes_i$ and then constructs the feasibility matrix $feas$, i.e., $feas_{ij} = 1$ if $j < firstdes_i$ and $feas_{ij} = 0$ otherwise. Psaraftis shows that the values $firstdes_i$ can be computed in $O(n^2)$ time, thus proving that 2-optimality can be verified in $O(n^2)$ time.

Next, we examine the Or-exchanges. An advantage of Or-exchanges is that they are *direction preserving*, i.e., the segments determined by the deletion of $\{i-1, i\}$, $\{i, i+1\}$, and $\{j, j+1\}$ are traversed in the same direction in the final tour. Therefore, if these segments are feasible in the original tour, they will also be

feasible in the final tour. There is only one situation that leads to infeasibility: a precedence-related pair of vertices with origin i and destination in the segment $(i+1,\ldots,j)$ for a forward Or-exchange and a precedence-related pair of vertices with origin in the segment $(j+1,\ldots,i-1)$ and destination i for a backward Or-exchange, since the order in which the vertex i and the segment are traversed in the final tour is reversed.

Similar screening procedures can be developed for both forward and backward Or-exchanges. For forward Or-exchanges we need the following observation: if $firstdes_i$ denotes the first destination of a precedence constraint with origin i, the exchange of $\{i-1,i\}$, $\{i,i+1\}$, and $\{j,j+1\}$ with $\{i-1,i+1\}$, $\{j,i\}$, and $\{i,j+1\}$ is feasible if and only if $j<firstdes_i$. An analogous observation can be made for backward Or-exchanges. The values $firstdes_i$ can be computed in $O(n^2)$ time, thus proving an overall complexity of $O(n^2)$ for verification of Or-optimality.

Solomon, Baker & Schaffer [1988] have adapted this preprocessing scheme to the TSP with time windows. The idea is to identify precedence constraints between pairs of vertices based on their time windows. If $e_i+d_{ij}>l_j$, then vertex j has to precede vertex i. The use of this type of preprocessing does not eliminate the need for further checking of feasibility; it may be used as a filter to reduce the number of complete feasibility checks required.

3.2 Updating mechanisms

Solomon, Baker & Schaffer [1988] have carried out an extensive computational study on the efficient implementation of k-exchange procedures for the TSP with time windows. Their implementation incorporated the preprocessing scheme discussed above, as well as tailored updating mechanisms for direction-preserving exchanges. These updating mechanisms are based on the observation that if the direction in which we traverse a path is unchanged, we can check the feasibility of each vertex on the path by simply looking at the change in arrival time.

Consider a path $(u,u+1,\ldots,v)$ with associated departure times and suppose that the departure time at the first vertex of the path is decreased. This defines a *push backward*

$$B_u = D_u - D_u^{\text{new}},$$

where D_u and D_u^{new} define the current and the new departure time at vertex u. The push backward at the next vertex on the path can be computed by

$$B_{u+1} = \min\{B_u, D_{u+1} - e_{u+1}\}.$$

Obviously, all vertices on the path remain feasible and the departure times D_k need to be adjusted sequentially for $k=u,\ldots,v$ as long as $B_k>0$.

Similarly, consider a path $(u,u+1,\ldots,v)$ with associated departure times and suppose that the departure time at the first vertex of the path is increased. This defines a *push forward*

$$F_u = D_u^{\text{new}} - D_u.$$

The push forward at the next vertex on the path can be computed by

$$F_{u+1} = \max\{F_u - W_{u+1}, 0\},$$

where W_{u+1} denotes the waiting time at vertex $u+1$. If $F_u > 0$, some vertices on the path could become infeasible. The vertices on the path have to be checked sequentially. At a vertex k ($u \leqslant k \leqslant v$), it may happen that $D_k + F_k > l_k$, in which case the path is no longer feasible, or $F_k = 0$, which indicates that the path from vertex k to vertex v has not been changed.

Observe that, in the worst case, testing the feasibility of an exchange takes $O(n)$ time. However, in practice, the use of a push backward and a push forward leads to a substantial reduction in the number of vertices being examined for time feasibility. Note that these techniques are only useful in direction-preserving exchanges.

3.3 Lexicographic search

Savelsbergh [1986, 1990, 1992] introduced an approach that can be used to incorporate all three side constraints in exchange procedures without increasing the time complexity of verifying local optimality. The basic idea is to use a specific *search strategy* in combination with a set of *global variables* such that testing the feasibility of a single exchange and maintaining the set global variables requires no more than constant time. We begin by presenting the search strategy because of its crucial importance.

- *Lexicographic search for 2-exchanges.* We choose the edges $\{i, i+1\}$ in the order in which they appear in the current route starting with $i = 1$ up to $i = n-2$; this will be called the outer loop. After fixing an edge $\{i, i+1\}$ we choose the edge $\{j, j+1\}$ successively equal to $\{i+2, i+3\}$, $\{i+3, i+4\}, \ldots, \{n, n+1\}$ (Figure 10.9); this will be called the inner loop. Now consider all possible exchanges for a fixed edge $\{i, i+1\}$. The inspection of the 2-exchanges in the order given above implies that, in the inner loop, the previously reversed path $(i+1, \ldots, j-1)$, corresponding to the substitution of $\{i, i+1\}$ and $\{j-1, j\}$ with $\{i, j-1\}$ and $\{i+1, j\}$, is expanded by the edge $\{j-1, j\}$.
- *Lexicographic search for forward Or-exchanges.* We choose vertex i in the order of the current route starting with i equal to 2. After fixing i we choose the

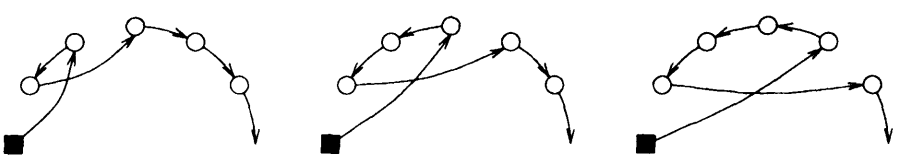

Figure 10.9 The search strategy for 2-exchanges

edge $\{j, j+1\}$ to be $\{i+1, i+2\}$, $\{i+2, i+3\}, \ldots, \{n, n+1\}$ consecutively. That is, the edge $\{j, j+1\}$ 'walks forward' through the route. In the inner loop in each newly examined exchange, the path $(i+1, \ldots, j-1)$ of the previously considered exchange is expanded with the edge $\{j-1, j\}$.

- *Lexicographic search for backward Or-exchanges.* We choose vertex i in the order of the current route starting with i equal to 2. After fixing i we choose the edge $\{j, j+1\}$ to be $\{i-2, i-1\}, \{i-3, i-2\}, \ldots, \{1, 2\}$ in that order. That is, the edge $\{j, j+1\}$ 'walks backward' through the route. In the inner loop in each newly examined exchange, the path $(j+2, \ldots, i-1)$ of the previously considered exchange is expanded with the edge $\{j+1, j+2\}$.

Now that we have presented the search strategy, let us return to the feasibility question. In order to test the feasibility of a single 2-exchange, we have to check all the vertices on the reversed path $(i+1, \ldots, j)$ and on the path $(j+1, \ldots, n+1)$; and in order to test the feasibility of a single forward (or backward) Or-exchange, we have to check besides vertex i the vertices on the paths $(i+1, \ldots, j)$ and $(j+1, \ldots, n+1)$ (or $(j+1, \ldots, i-1)$ and $(i+1, \ldots, n+1)$). In a straightforward implementation this takes $O(n)$ time for each single exchange. We will present an implementation that requires only constant time per exchange.

The idea is to define an appropriate set of global variables, which will of course depend on the constrained variant of the TSP we are considering, in such a way as to meet the following criteria:

- The set of global variables makes it possible to test the feasibility of an exchange in constant time, i.e., the feasibility of all the vertices on the paths in question can be checked in constant time.
- The lexicographic search strategy makes it possible to maintain the correct values for the set of global variables in constant time, i.e., when we go from one exchange to the next, the global variables can be updated in constant time.

To see how these ideas work in actual implementations, we show the pseudocode of a general framework for a 2-exchange procedure:

```
procedure TwoExchange
(* input: a route given as (1,...,n+1*)
(* output: a route that is 2-optimal*)
REPEAT:
  for i ← 1 to n − 2 do                          (* outer loop: fix edge {i, i+1}*)
    InitializeGlobalVariables (i, G)             (*initialize the set of global variables G*)
    for j ← i+2 to n do                          (* inner loop: fix edge {j, j+1}*)
      if ProfitableExchange (i, j, G) and        (* test whether the exchange is profitable*)
        FeasibleExchange (i, j, G)               (* test whether the exchange is feasible*)
      then
        PerformExchange (i, j)
        goto REPEAT      (*replace {i, i+1} and {j, j+1} by {i, j} and {i+1, j+1}*)
      UpdateGlobalVariables (i, j, G)
                         (* update the set of global variables G for the next iteration*)
END
```

Although the above pseudocode looks rather simple, defining a set of global variables in such a way that, in combination with the lexicographic search strategy, the functions InitializeGlobalVariables, ProfitableExchange, FeasibleExchange, and UpdateGlobalVariables do what they are supposed to do and take only constant time, is often not so obvious.

Reexamining a 2-exchange, a forward Or-exchange, and a backward Or-exchange, we see that the algorithms have to be able to handle reversing a path, relocating a path backward and relocating a path forward. As these three types of changes comprise all the possibilities that can occur in a k-exchange (for arbitrary k), the techniques can be used to implement k-exchange algorithms for the TSP with side constraints for arbitrary k without increasing the time complexity beyond $O(n^k)$.

Precedence constraints

As a first illustration of the proposed technique, we show how precedence constraints can be handled. We cannot improve the time complexity of $O(n^2)$ of Psaraftis' implementations, so we just present alternative and simpler implementations of his idea. Attach a label to each vertex u with information on the first vertex on the tour for which a precedence constraint $u \to v$ exists: $first_u = \min\{v \mid u \to v\}$.

- *2-exchanges.* Recall that the exchange of $\{i, i+1\}$ and $\{j, j+1\}$ with $\{i, j\}$ and $\{i+1, j+1\}$ is feasible if and only if $j < firstdes_i$, where $firstdes_i$ denotes the first destination of a precedence constraint for which both origin and destination lie beyond i. This is equivalent to stating that a 2-exchange is feasible if and only if there is no precedence-related pair of vertices on the path that is reversed. Hence, at any stage, information concerning this path is sufficient. The lexicographic search strategy makes it possible to gather the required knowledge using a single global variable. We introduce the variable F to denote the first destination of a precedence constraint for which the corresponding origin is on the path that is to be reversed. Feasibility of a 2-exchange is now established by verifying that the vertex denoted by the variable F is not on the reversed path.

  All we have to do now is to show that we can ensure that the global variable F contains the right information when it is examined. The lexicographic search strategy provides a simple way to accomplish this. In the outer loop, whenever we expand the path $(1, \ldots, i-1)$ with the edge $\{i-1, i\}$ we set F equal to $first_{i+1}$. In the inner loop, whenever we expand the path $(i+1, \ldots, j-1)$ with the edge $\{j-1, j\}$, we set F equal to the minimum of its current value and $first_j$.
- *Or-exchanges.* A forward Or-exchange is feasible if and only if there is no pair of precedence-related vertices with the first being i and the other on the path $(i+1, \ldots, j)$. Whenever we try to expand the path $(i+1, \ldots, j-1)$ with the edge $\{j-1, j\}$ and $i \to j$, i.e., vertex i is a predecessor of vertex j, the expansion will only result in infeasible exchanges. Similarly, a backward Or-exchange is

feasible if and only if there is no pair of precedence-related vertices with the first on the path $(j+1,\ldots,i-1)$ and the other being i. Whenever we try to expand the path $(j+2,\ldots,i-1)$ with the edge $\{j+1,j+2\}$ and $j+1 \to i$, i.e., vertex $j+1$ is a predecessor of vertex i, the expansion will only result in infeasible exchanges.

Before discussing collections and deliveries and time windows, we take another close look at a k-exchange and the lexicographic search strategy. A k-exchange is the substitution of k links of a tour with k other links. The first step is the deletion of k links, i.e., the tour is broken up into k paths. The second step is the addition of k other links, i.e., the k paths are concatenated in a different order to form a new tour.

More specifically, a 2-exchange deletes the links $\{i,i+1\}$ and $\{j,j+1\}$, with $j>i$, to form the paths $(1,\ldots,i)$, $(i+1,\ldots,j)$, and $(j+1,\ldots,n+1)$, then adds the links $\{i,j\}$ and $\{i+1,j+1\}$ to obtain the new tour $(1,\ldots,i,j,\ldots,i+1,j+1,\ldots,n+1)$; a forward Or-exchange deletes the links $\{i-1,i\}$, $\{i,i+1\}$, and $\{j,j+1\}$ to form the paths $(1,\ldots,i-1)$, (i), $(i+1,\ldots,j)$, and $(j+1,\ldots,n+1)$, then adds the links $\{i-1,i+1\}$, $\{j,i\}$, and $\{i,j+1\}$ to obtain the new tour $(1,\ldots,i-1,j,i,j+1,\ldots,n+1)$; and a backward Or-exchange deletes the links $\{i-1,i\}$, $\{i,i+1\}$, and $\{j,j+1\}$ to form the paths $(1,\ldots,j)$, $(j+1,\ldots,i-1)$, (i), and $(i+1,\ldots,n+1)$, then adds the links $\{i-1,i+1\}$, $\{j,i\}$, and $(i,j+1\}$ to obtain the new tour $(1,\ldots,j,i,j+1,\ldots,i-1,i+1,\ldots,n+1)$.

The key feature of the lexicographic search strategy is that in consecutive iterations the k paths which result after the deletion of the k links differ by at most a single vertex. The proposed implementation scheme for k-exchange methods associates a set of global variables with each of these paths containing information on its feasibility and its profitability. The global variables are chosen such that initializing the global variables for a single-vertex path and computing the values of the global variables for a concatenated path both take constant time.

Collections and deliveries

The following three quantities turn out to be sufficient for the analysis of feasibility for the TSP with collections and deliveries.

- *Maximum load* $L^{\max}_{(u_1,\ldots,u_k)}$ on the path $(u_1,\ldots,u_k)$, assuming the vehicle is empty when it arrives at vertex u_1, i.e.,

$$L^{\max}_{(u_1,\ldots,u_k)} = \max_{1\leq i\leq k} \sum_{1\leq j\leq i} q_{u_j}.$$

- *Minimum load* $L^{\min}_{(u_1,\ldots,u_k)}$ on the path $(u_1,\ldots,u_k)$, assuming the vehicle is empty when it arrives at vertex u_1, i.e.,

$$L^{\min}_{(u_1,\ldots,u_k)} = \min_{1\leq i\leq k} \sum_{1\leq j\leq i} q_{u_j}.$$

- *Final load* $L^{\text{final}}_{(u_1,\ldots,u_k)}$ on the path $(u_1,\ldots,u_k)$, assuming the vehicle is empty when it arrives at vertex u_1, i.e.,

$$L^{\text{final}}_{(u_1,\ldots,u_k)} = \sum_{1\leq j\leq k} q_{u_j}.$$

Initializing these quantities for a single-vertex path (u) is trivial: $L^{\max}_{(u)} = L^{\min}_{(u)} = L^{\text{final}}_{(u)} = q_u$. The following proposition states that, if we concatenate two (vertex-disjoint) paths, we can compute the quantities for the resulting path from the quantities of its constituent paths in constant time.

Proposition 1 *If two vertex-disjoint feasible paths* $(u_1, \ldots, u_k)$ *and* $(v_1, \ldots, v_l)$, *with associated maximal loads* $L^{\max}_{(u_1,\ldots,u_k)}$ *and* $L^{\max}_{(v_1,\ldots,v_l)}$, *minimal loads* $L^{\max}_{(u_1,\ldots,u_k)}$ *and* $L^{\min}_{(v_1,\ldots,v_l)}$, *and final loads* $L^{\text{final}}_{(u_1,\ldots,u_k)}$ *and* $L^{\text{final}}_{(v_1,\ldots,v_l)}$, *are concatenated, the same values for the resulting path* $(u_1, \ldots, u_k, v_1, \ldots, v_l)$ *are given by*

$$L^{\max}_{(u_1,\ldots,u_k,v_1,\ldots,v_l)} = \max\{L^{\max}_{(u_1,\ldots,u_k)}, L^{\text{final}}_{(u_1,\ldots,u_k)} + L^{\max}_{(v_1,\ldots,v_l)}\},$$

$$L^{\min}_{(u_1,\ldots,u_k,v_1,\ldots,v_l)} = \min\{L^{\min}_{(u_1,\ldots,u_k)}, L^{\text{final}}_{(u_1,\ldots,u_k)} + L^{\min}_{(v_1,\ldots,v_l)}\},$$

$$L^{\text{final}}_{(u_1,\ldots,u_k,v_1,\ldots,v_l)} = L^{\text{final}}_{(u_1,\ldots,u_k)} + L^{\text{final}}_{(v_1,\ldots,v_l)}.$$

From the discussion on k-exchanges and the lexicographic search strategy, it should be clear how Proposition 1 shows that checking the feasibility of an exchange, i.e., $L^{\max}_{(1,\ldots,n+1)} \leqslant Q$ and $L^{\min}_{(1,\ldots,n+1)} \geqslant 0$, as well as updating between consecutive iterations can be done in constant time.

Time windows

Under the assumption that, on its way, a vehicle always departs at a vertex as early as possible, which is the best choice from a feasibility point of view, a path can be completely specified by giving the sequence in which the vertices are visited and the departure time at the first vertex of the path.

The following quantities turn out to be useful in the analysis of feasibility and profitability of k-exchanges in the TSP with time windows:

- *Total travel time* $T_{(u_1,\ldots,u_k)}$ on the path $(u_1, \ldots, u_k)$, i.e.,

$$T_{(u_1,\ldots,u_k)} = \sum_{1 \leqslant i \leqslant k} d_{u_i u_{i+1}}.$$

- *Earliest departure time* $E_{(u_1,\ldots,u_k)}$ at vertex u_k of the path $(u_1, \ldots, u_k)$, assuming vertex u_1 is left at the opening of its time window, i.e.,

$$E_{(u_1,\ldots,u_k)} = \max_{1 \leqslant i \leqslant k} \{e_i + T_{(u_i,\ldots,u_k)}\}.$$

- *Latest arrival time* $L_{(u_1,\ldots,u_k)}$ at vertex u_1 of the path $(u_1, \ldots, u_k)$, such that the path remains feasible, i.e.,

$$L_{(u_1,\ldots,u_k)} = \min_{1 \leqslant i \leqslant k} \{l_i - T_{(u_1,\ldots,u_i)}\}.$$

Other interesting quantities can be obtained using the above values. For example, the waiting time on the path $(u_1, \ldots, u_k)$ is equal to $E_{(u_1,\ldots,u_k)} - e_{u_1} - T_{(u_1,\ldots,u_k)}$.

Proposition 2 shows that, if we concatenate two paths $(u_1, \ldots, u_k)$ and $(v_1, \ldots, v_l)$, we can compute the same quantities for the resulting path $(u_1, \ldots, u_k, v_1, \ldots, v_l)$ from the quantities of its constituent paths in constant time.

Proposition 2 *If two vertex-disjoint feasible paths* $(u_1, \ldots, u_k)$ *and* $(v_1, \ldots, v_l)$, *with associated total travel times loads* $T_{(u_1,\ldots,u_k)}$ *and* $T_{(v_1,\ldots,v_l)}$, *earliest departure times* $E_{(u_1,\ldots,u_k)}$ *and* $E_{(v_1,\ldots,v_l)}$, *and latest arrival times* $L_{(u_1,\ldots,u_k)}$ *and* $L_{(v_1,\ldots,v_k)}$, *are concatenated, the resulting path is feasible if and only if* $E_{(u_1,\ldots,u_k)} + d_{u_k,v_1} \leqslant L_{(v_1,\ldots,v_l)}$ *and the values for the resulting path are given by*

$$T_{(u_1,\ldots,u_k,v_1,\ldots,v_l)} = T_{(u_1,\ldots,u_k)} + d_{u_k,v_1} + T_{(v_1,\ldots,v_l)},$$

$$E_{(u_1,\ldots,u_k,v_1,\ldots,v_l)} = \max\{E_{(u_1,\ldots,u_k)} + d_{u_k,v_1} + T_{(v_1,\ldots,v_l)}, E_{(v_1,\ldots,v_l)}\},$$

$$L_{(u_1,\ldots,u_k,v_1,\ldots,v_l)} = \min\{L_{(u_1,\ldots,u_k)}, L_{(v_1,\ldots,v_l)} - T_{(u_1,\ldots,u_k)} - d_{u_k,v_1}\}.$$

From the discussion on k-exchanges and the lexicographic search strategy, it should be clear how Proposition 2 shows that checking the feasibility of a k-exchange as well as updating between consecutive iterations can be done in constant time.

The presence of time windows also allows for the specification of a variety of objective functions, such as the minimization of the total travel time, i.e., $T_{(1,\ldots,n+1)}$, the minimization of the completion time, i.e., $E_{(1,\ldots,n+1)}$, and the minimization of the route duration, i.e., $\max\{E_{(1,\ldots,n+1)} - L_{(1,\ldots,n+1)}, T_{(1,\ldots,n+1)}\}$.

Van der Bruggen, Lenstra & Schuur [1993] have developed an efficient variable-depth improvement algorithm for the single-vehicle dial-a-ride problem using the techniques described above. The lexicographic search strategy is applied until a feasible and profitable 2-exchange is found. Now instead of actually performing the exchange, they continue the search for a 2-exchange on the path that has not been affected by the exchange. They repeat this as long as the combined exchanges are profitable. In this way, in a single step, a series of exchanges are examined from which the best is selected (Figure 10.10).

One can imagine many variants of this procedure. Any type of exchange, e.g., Or-exchange or 3-exchange, can be used as a basic improvement step in the procedure. Also, the best possible exchange can be chosen at each step, instead of the first feasible and profitable one encountered.

3.4 Computational evidence

To illustrate the effectiveness of the techniques, we now briefly describe the computational experiments carried out by Knops [1993]. These experiments focus on the performance of 2-exchange algorithms for the TSP with time windows. Knops considers the following three implementations: a straightforward search of the 2-exchange neighborhood, the straightforward search enhanced with the improvements proposed by Solomon, Baker & Schaffer [1988] (cf. Section 3.1), and the lexicographic search approach of Savelsbergh [1986, 1990, 1992] (cf. Section 3.3).

The test problems are generated from Solomon's problem set for the VRP with time windows [Solomon, 1987]. From each of the problems in Solomon's class $R2$, instances of the TSP with time windows are created by taking the first k customers, for $k = 10, 15, 20$, and 30. Compared to Solomon's other classes, the

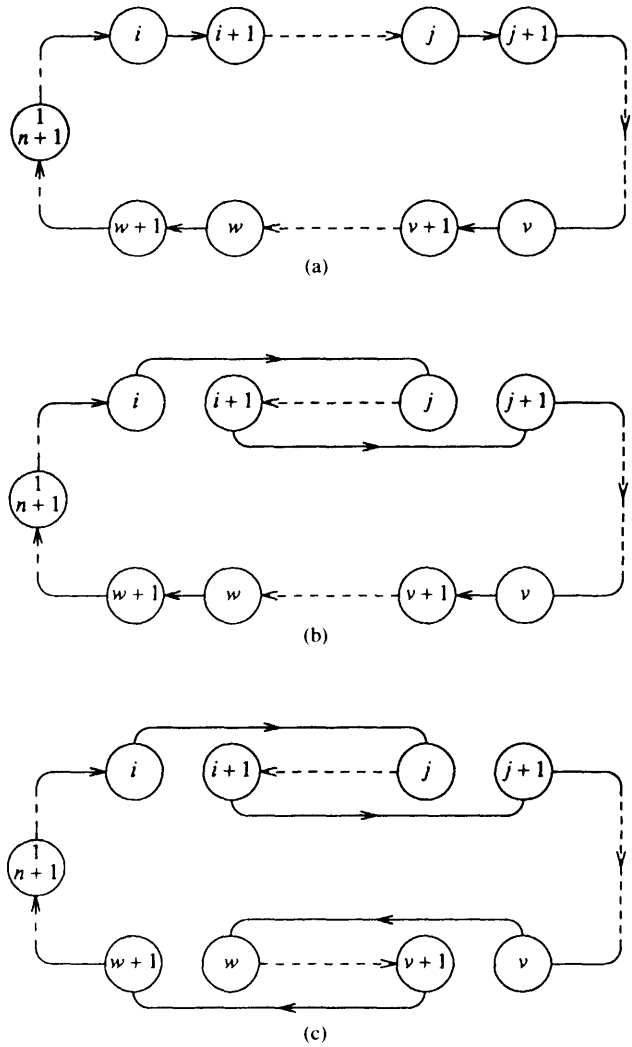

Figure 10.10 A variable-depth exchange: (a) the initial tour, (b) after the first step, (c) after the second step

class $R2$ is best fit for this purpose, since the location of the customers and the time windows are such that feasible routes exist and allow for a substantial number of improvement steps. For each of these problems an initial route was determined by some simple backtracking procedure based on the nearest neighbor heuristic for the TSP: among the customers not yet visited choose as the next customer the one that is closest to the last visited customer; if a particular choice leads to infeasibility of the route, choose the next closest customer, and so on.

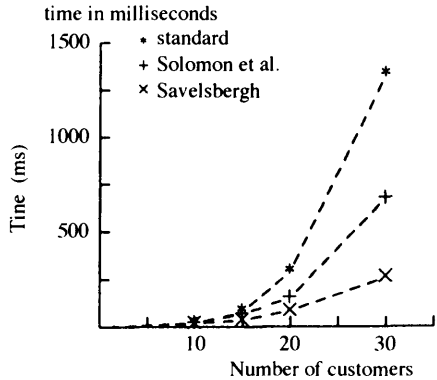

Figure 10.11 Computational results of searching the 2-exchange neighborhood: (●) standard (+) Solomon–Baker–Schaffer (×) Savelsbergh

Although the problem of deciding whether a feasible tour exists is NP-complete, the generated instances are such that the construction of an initial route hardly takes any time.

In the first experiment, the computation times for the three different implementations are compared for various instance sizes, i.e., number of customers. The results are shown in Figure 10.11. They show that the enhancements proposed by Solomon, Baker & Schaffer substantially reduce the computation times, compared to the straightforward implementation, but the techniques proposed by Savelsbergh perform even better and reduce the computation times even more significantly.

Besides the number of customers, another interesting parameter is the width of the time windows. In the second experiment, the effect of the width of the time

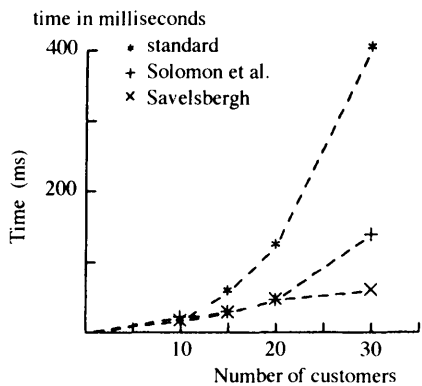

Figure 10.12 The effect of narrowing the time windows, $\gamma = 0.50$: (●) standard (+) Solomon–Baker–Schaffer (×) Savelsbergh

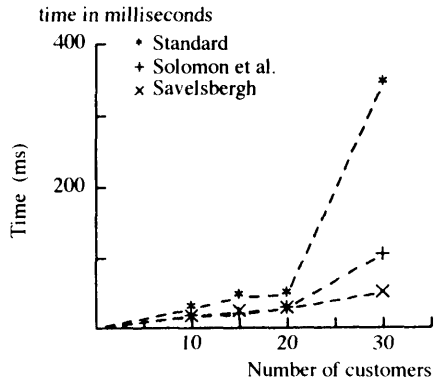

Figure 10.13 The effect of narrowing the time windows, $\gamma = 0.25$: (●) standard (+) Solomon Baker Schaffer (×) Savelsbergh

window on the performance of the three implementations is tested. The time windows of the customers were tightened by factors of 2 and 4. To maintain feasibility of the initial route found earlier, this tightening was performed in the following way. If the initial route arrives at a customer at time t, the original time window $[e, l]$ of this customer is transformed into $[(1-\gamma)t + \gamma e, (1-\gamma)t + \gamma l]$ with γ equal to 0.50 or 0.25.

The results are presented in Figures 10.12 and 10.13. One would expect that the Solomon–Baker–Schaffer algorithm benefits more from smaller time windows than the lexicographic search approach, since smaller time windows imply more precedence constraints. However, this does not seem to be the case. A simple analysis of the techniques based on lexicographic search reveals that they in fact identify and check the precedence constraints online as a subset of the feasibility tests performed [Knops, 1993].

4 PARALLEL IMPLEMENTATIONS

Local search algorithms for vehicle routing problems are often embedded in interactive planning systems for physical distribution management. Any algorithm embedded in an interactive system should have an acceptable response time. Therefore speed is a primary concern. Nowadays many computers are able to perform a number of operations in parallel. Parallel computers have a greater processing power than serial computers, thus making it possible to obtain substantial speedups.

In this section we will discuss the verification of local optimality on a parallel random access machine (PRAM), a machine model in which an unbounded number of processors operate in parallel and communicate with each other in constant time through a shared memory. The shared memory allows simultaneous reads from the same location but disallows simultaneous writes into the

same location. Although the PRAM is hardly a realistic computer model, the resulting algorithms can be adequately used for implementation on any real-world machine and the overhead introduced is only minimal (see e.g. Alt et al. [1987], Karlin & Upfal [1988]).

Before addressing the issue of local optimality, we will first consider the *partial sums* problem and describe a basic technique in parallel computing for its solution. Then we will address the verification of local optimality of a single route for the 2-exchange neighborhood in the presence of time windows. We will not deal with the other cases considered in this chapter, but we only mention that parallel algorithms can be obtained in much the same way. The subsequent discussion is extracted from Kindervater, Lenstra & Savelsbergh [1993].

Partial sums

For the sake of simplicity, let $n = 2^m$ and suppose that n numbers are given by a_n, $a_{n+1}, \ldots, a_{2n-1}$. We wish to find the partial sums $b_{n+j} = a_n + \cdots + a_{n+j}$ for $j = 0, \ldots, n-1$. The following procedure is due to Dekel & Sahni [1983]; it consists of two phases.

for $l \leftarrow m-1$ **downto** 0 **do**
  par $[2^l \leqslant j \leqslant 2^{l+1}-1]$ $a_j \leftarrow a_{2j} + a_{2j+1}$;
for $l \leftarrow 0$ **to** m **do**
  par $[2^l \leqslant j \leqslant 2^{l+1}-1]$
    $b_j \leftarrow$ **if** $j = 2^l$ **then** a_j **else if** j odd **then** $b_{(j-1)/2}$ **else** $b_{(j-2)/2} + a_j$.

Here, a statement of the form '**par** $[\alpha \leqslant j \leqslant \omega]$ s_j' denotes that the statements s_j are executed in parallel for all values of j in the indicated range.

The computation is illustrated in Figure 10.14. In the first phase, represented by solid arrows, the sum of the a_j's is calculated. The a-value corresponding to a nonleaf node is set equal to the sum of all a-values corresponding to the leaves

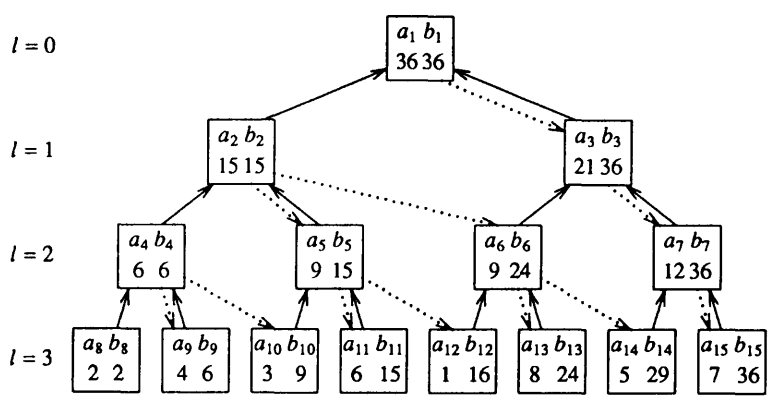

Figure 10.14 Partial sums: an instance with $n = 8$

descending from that node. In the second phase, represented by dotted arrows, the b-value of a certain node is set equal to the sum of all a-values of the nodes of the same generation, except those with a higher index. This implies, in particular, that at the end we have $b_{n+j} = a_n + \cdots + a_{n+j}$ for $j = 0, \ldots, n-1$.

The algorithm requires $O(\log n)$ time and n processors. This can be improved to $O(\log n)$ time and $O(n/\log n)$ processors by a simple device. First, the set of n numbers is partitioned into $n/\log n$ groups of size $\log n$ each, and $n/\log n$ processors determine the sum of each group in the traditional serial way in $\log n$ time. After this aggregation process, the above algorithm computes the partial sums over the groups; this requires $O(n/\log n)$ processors and $O(\log n)$ time. Finally, a disaggregation process is applied with the same processor and time requirements. The total computational effort is $O(\log n \cdot n/\log n) = O(n)$, as it is in the serial case. This is called a *full processor utilization* or a *perfect speedup*.

The technique described above is widely used in parallel computing. In many situations a kind of partial sums problem has to be solved where the addition is replaced by some other associative binary operator.

Verification of local optimality

We now return to the verification of local optimality. We start by deciding whether or not the tour $(1, 2, \ldots, n, n+1)$ is 2-optimal.

par $[1 \leqslant i < j \leqslant n]$ $\delta_{ij} \leftarrow d_{ij} + d_{i+1,j+1} - d_{i,i+1} - d_{j,j+1}$;
(*compute the effect of all possible 2-exchanges*)
$\delta_{\min} \leftarrow \min\{\delta_{ij} \mid 1 \leqslant i < j \leqslant n\}$; (*search for the best 2-exchange*)
if $\delta_{\min} \geqslant 0$
then $(1, 2, \ldots, n, n+1)$ is a 2-optimal tour
else let i^* and j^*, with $i^* < j^*$, be such that $\delta_{i^*j^*} = \delta_{\min}$,
$(1, \ldots, i^*, j^*, j^*-1, \ldots, i^*+1, j^*+1, \ldots, n+1)$ is a shorter tour.

By adapting the first phase of the partial sums algorithm such that it computes the minimum of a set of numbers and also delivers an index for which the minimum is attained, the above procedure can be implemented to require $O(\log n)$ time and $O(n^2 * \log n)$ processors. The total computational effort is $O(\log n \cdot n^2/\log n) = O(n^2)$, as it is in the serial case. Hence, we have obtained a perfect speedup.

Although the serial and parallel implementations seem similar, there is a basic distinction. When the tour under consideration is not 2-optimal, the serial algorithm will detect this after a number of steps that is somewhere in between 1 and $\binom{n}{2}$. In the parallel algorithm, confirmation and negation of 2-optimality or Or-optimality always take the same amount of time.

The presence of time windows complexifies the situation. Consider the tour $(1, 2, \ldots, n, n+1)$, which is assumed to be feasible. We start by looking at all partial paths along the tour. This enables us to construct the tours that can be obtained by a 2-exchange. For each path $(u_1, \ldots, u_k)$, we define $A_{(u_1, \ldots, u_k)}(t)$ as the earliest possible arrival time at vertex u_k when traveling along the path from

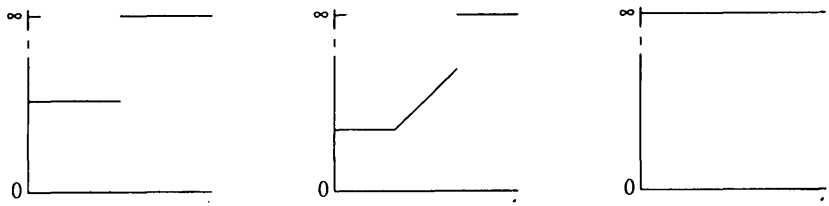

Figure 10.15 The three possible shapes of the functions A

vertex u_1 to vertex u_k after arriving at vertex u_1 at time t. Note that $A_{(1,\ldots,n+1)}(e_1)$ is the arrival time at vertex $n+1$ of the given tour. Our algorithm has three phases.

1. First, we compute the functions A for paths consisting of only one edge.

$$\textbf{par}\,[1 \leqslant i \neq j \leqslant n+1]\; A_{(i,j)}(t) \leftarrow \begin{cases} \max\{e_i, t\} + d_{ij} & \text{for } t \leqslant l_i, \\ \infty & \text{for } t > l_i. \end{cases}$$

   Next, we compute the functions A for all paths along the tour in both directions by composition. (The composition operator will be denoted by $\circ$ in the formulae.)

$$\textbf{par}\,[1 \leqslant i \leqslant n+1]\;\textbf{par}\,[i \leqslant j \leqslant n+1]\; A_{(i,i+1,\ldots,j-1,j)}(t) \leftarrow (A_{(j-1,j)} \circ \cdots \circ A_{(i,i+1)})(t);$$
$$\textbf{par}\,[1 \leqslant i \leqslant n+1]\;\textbf{par}\,[i \leqslant j \leqslant n+1]\; A_{(j,j-1,\ldots,i+1,i)}(t) \leftarrow (A_{(i+1,i)} \circ \cdots \circ A_{(j,j-1)})(t).$$

   By considering all possibilities, one can show that each of these functions has one of the three shapes shown in Figure 10.15. Composing functions is an associative operation. Hence, we can use the partial sums algorithm for obtaining the functions A in parallel. Since a composition of two functions of the type described here can be derived in constant time, we can in fact determine all functions A in $O(\log n)$ time with $O(n^2/\log n)$ processors.
2. Given the functions A for paths along the tour, we can compute them for the tours that are obtained after replacement of the edges $\{i, i+1\}$ and $\{j, j+1\}$ by the edges $\{i, j\}$ and $\{i+1, j+1\}$:

$$\textbf{par}\,[1 \leqslant i < j \leqslant n]\; A_{(1,\ldots,i,j,\ldots,i+1,j+1,\ldots,n+1)}(t)$$
$$\leftarrow (A_{(j+1,\ldots,n+1)} \circ A_{(i+1,j+1)} \circ A_{(j,\ldots,i+1)} \circ A_{(i,j)} \circ A_{(1,\ldots,i)})(t).$$

   For this phase we need $O(1)$ time and $O(n^2)$ processors, or $O(\log n)$ time and $O(n^2/\log n)$ processors.
3. We decide whether or not the given tour is 2-optimal in the same way as in the case without time windows.

$$A_{\min} \leftarrow \min\{A_{(1,\ldots,i,j,\ldots,i+1,j+1,\ldots,n+1)}(e_1) \mid 1 \leqslant i \leqslant j \leqslant n\};$$

   if $A_{(1,\ldots,n+1)}(e_1) \leqslant A_{\min}$
   then $(1, 2, \ldots, n, n+1)$ is a 2-optimal tour
   else let i^* and j^*, with $i^* < j^*$, be such that $A_{(1,\ldots,i^*,j^*,\ldots,i^*+1,j^*+1,\ldots,n+1} = A_{\min}$,
   $(1, \ldots, i^*, j^*, j^*-1, \ldots, i^*+1, j^*+1, \ldots, n+1)$ is a better feasible tour.

For this last phase, the same time and processor bounds as before suffice. So we end up with an algorithm that runs in $O(\log n)$ time using $O(n^2/\log n)$ processors, which is the same as in the unconstrained case.

For the verification of k-optimality for fixed $k > 2$, we can derive a logarithmic-time algorithm along similar lines. One has to take into account that, given k edges, several k-exchanges are possible. Further, the influence of a k-exchange on a tour is more complex. However, the running time remains $O(\log n)$ using $O(n^k/\log n)$ processors, which is optimal with respect to the number $\Theta(n^k)$ of k-exchanges.

5 MODERN HEURISTICS

Not surprisingly, people have started to investigate the potential of recently developed variants of local search, such as simulated annealing [Kirkpatrick, Gelatt & Vecchi, 1983] and tabu search [Glover, 1989, 1990], for the solution of vehicle routing problems. The majority of the simulated annealing and tabu search algorithms for vehicle routing problems use one or more of the neighborhoods described in Section 2 and some of the techniques discussed in Section 3. The simulated annealing algorithms [Alfa, Heragu & Chen, 1991; Osman, 1993] mainly differ in their choice of cooling scheme and parameters; the tabu search algorithms [Gendreau, Hertz & Laporte, 1994; Potvin et al., 1996; Semet & Taillard, 1993] mainly differ in their choice of tabu list structure and parameters. For a discussion of these methods, we refer to Chapter 9 of this book.

Local Search in Combinatorial Optimization
Edited by E. Aarts and J. K. Lenstra
© 1997 John Wiley & Sons Ltd.

11

Machine scheduling

Edward J. Anderson
University of New South Wales, Sydney

Celia A. Glass, Chris N. Potts
University of Southampton, Southampton

1 INTRODUCTION 362
2 SCHEDULING MODELS 363
2.1 Machine environment 363
2.2 Job characteristics 364
2.3 Optimality criteria 365
2.4 Three-field representation 365
2.5 Some examples 366
3 APPLYING LOCAL SEARCH 367
3.1 Representation of solutions 367
3.2 Neighborhood search 368
3.3 Genetic algorithms 369
3.4 Neural networks 370
3.5 Feasibility 370
4 NEIGHBORHOOD STRUCTURES AND RECOMBINATION OPERATORS 371
4.1 Sequencing problems 372
4.2 Assignment and partitioning problems 375
4.3 Combined assignment and sequencing problems 377
4.4 Multisequencing problems 378
4.5 Merging ordered lists of jobs 380
5 SINGLE-MACHINE PROBLEMS 382
5.1 Maximum lateness 382
5.2 Total weighted completion time 383
5.3 Total weighted tardiness 385
5.4 Weighted number of late jobs 387
5.5 Total weighted earliness and tardiness 387
6 PARALLEL-MACHINE PROBLEMS 389
6.1 Identical parallel machines 389
6.2 Uniform parallel machines 391
6.3 Unrelated parallel machines 391
7 MULTI-STAGE PROBLEMS 392
7.1 Flow shops 392
7.2 Open shops 396
7.3 Job shops 397

8 COMPUTATIONAL COMPARISONS 403
8.1 Merging ordered lists of jobs in $1|s_j|\sum w_j C_j$ 403
8.2 Assignment of jobs in $R \| C_{max}$ 404
8.3 Sequencing of jobs in $F \| \sum w_j C_j$ 407
8.4 Multisequencing of jobs in $J \| C_{max}$ 408
9 CONCLUDING REMARKS 413

1 INTRODUCTION

The scheduling of computer and manufacturing systems has been the subject of extensive research since the early 1950s. The range of application areas for scheduling theory goes beyond computers and manufacturing to include agriculture, hospitals, transport, etc. The main focus is on the efficient allocation of one or more resources to activities over time. It is convenient to adopt manufacturing terminology: we refer to a *job* which consists of one or more activities, and a *machine* which is a resource that can perform at most one activity at a time.

We restrict our attention to *deterministic machine scheduling* where it is assumed that the data that define a problem instance are known with certainty in advance. An excellent survey of the area is the paper by Lawler et al. [1993], and the textbooks of Conway, Maxwell & Miller [1967], Baker [1974], French [1982] and Błazewicz et al. [1994] provide an introduction.

Much of the early work on scheduling was concerned with the analysis of single-machine systems. Examples include Jackson's derivation of the earliest due date (EDD) rule in which jobs are sequenced in order of nondecreasing due dates [Jackson, 1955], and Smith's derivation of the shortest weighted processing time (SWPT) rule in which jobs are sequenced in order of nondecreasing processing time to weight ratios [Smith, 1956]. In addition to providing optimal solutions to single-machine problems, these orderings are used as priority rules for scheduling more complex systems.

A major theme in recent research has been the use of complexity theory to classify scheduling problems as polynomially solvable or NP-hard. Many fundamental results in this area are derived by Lenstra, Rinnooy Kan & Brucker [1977]. The NP-hardness of a problem suggests that it is impossible to find an optimal solution without the use of an essentially enumerative algorithm, for which computation times will increase exponentially with problem size. To obtain exact solutions of NP-hard scheduling problems, a branch-and-bound or dynamic programming algorithm is usually applied. In most cases where these algorithms have been successful in solving problems of reasonable size, problem-specific features are used to restrict the search.

In practice it may be acceptable to use a heuristic method to find an approximate solution for an NP-hard problem. There is clearly a trade-off between the computational investment in obtaining a solution and the quality of that solution. The performance of heuristic methods is often evaluated empirically, but it is sometimes possible to carry out a theoretical analysis of heuristic performance. Following the pioneering work of Graham [1966, 1969] on list

scheduling heuristics for parallel machines, there has been a significant interest in obtaining performance guarantees through worst-case analysis. An alternative approach is to use probabilistic analysis to predict the behavior for 'typical' problem instances.

There are, however, some classes of problems that have resisted attempts to design a satisfactory solution procedure: enumerative algorithms may be unable to solve problems with more than a handful of jobs, and the solutions generated by simple heuristic methods may be far from the optimum. Such problems can be tackled by local search methods. These methods have the advantage that they can be employed as 'black-box' techniques if no problem-specific knowledge is available. On the other hand, it is often possible to incorporate structural properties of a problem into the local search method to improve its performance.

In addition to reviewing the literature and presenting the results of computational tests, this chapter builds a framework for the application of local search methods to scheduling problems. Section 2 starts with a description of scheduling problems and then presents a classical representation scheme based on the physical environment and the performance criterion for the problem. The combinatorial nature of the various problem types is also identified; for example, many problems involve sequencing jobs or assigning jobs to machines. Section 3 discusses design features for local search algorithms including the representation of solutions and issues of feasibility. Section 4 outlines a range of possible local search algorithms for each of the broad classes of combinatorial structure that are encountered in scheduling problems. Specific applications of local search methods are reviewed in Sections 5, 6, and 7, within the general categories of single-machine, parallel-machine, and multi-stage problems. Section 8 compares the computational performance of different local search methods for four different types of scheduling problem. Section 9 makes some concluding remarks.

2 SCHEDULING MODELS

The machine scheduling problems that we consider can be described as follows. There are *m machines*, which are used to process *n jobs*. A *schedule* specifies, for each machine i and each job j, one or more time intervals throughout which processing is performed on j by i. A schedule is *feasible* if there is no overlapping of time intervals corresponding to the same job (so that a job cannot be processed by two machines at once), or of time intervals corresponding to the same machine (so that a machine cannot process two jobs at the same time), and also if it satisfies various requirements relating to the specific problem type. The problem type is specified by the machine environment, the job characteristics, and an optimality criterion.

2.1 Machine environment

Different configurations of machines are possible. In each case, however, all machines become available to process jobs at time zero.

A *single-stage* production system requires one operation for each job, whereas in *multi-stage* systems there are jobs that require operations on different machines. Single-stage systems involve either a single machine, or m machines operating in parallel. In the case of parallel machines, each machine has the same function. We consider three cases: *identical parallel machines* in which each processing time is independent of the machine performing the operation; *uniform parallel machines* in which the machines operate at different speeds but are otherwise identical; and *unrelated parallel machines* in which the processing time of an operation depends on the machine assignment.

There are three main types of *multi-stage* systems. All such systems that we consider contain m machines, each having a different function. In a *flow shop* with m stages, each job is processed on machines $1, \ldots, m$ in that order. In an *open shop* each job is also processed once on each machine, but the machine routing (that specifies the sequence of machines through which a job must pass) can differ between jobs and forms part of the decision process. In a *job shop* each job has a prescribed routing through the machines, and the routing may differ from job to job.

2.2 Job characteristics

The processing requirements of each job j are given: for the case of a single machine and identical parallel machines, p_j is the processing time; for uniform parallel machines, the processing time on machine i may be expressed as p_j/u_i, where u_i is the speed of machine i; for the case of unrelated parallel machines, a flow shop and an open shop, p_{ij} is the processing time on machine i; and for a job shop, p_{ij} denotes the processing time of the ith operation (which is not necessarily performed on machine i).

In addition to its processing requirements, a job is characterized by its availability for processing, any setup requirements on the machine, any dependence on other jobs, and whether interruptions in the processing of its operations are allowed. The availability of each job j may be affected by its *release date* r_j, which is the time that it becomes available for processing, or by its *deadline* $\bar{d}_j$, which specifies the time by which it must be completed.

Suppose that job j is sequenced immediately before job k on machine i. Then, after job j has completed processing on machine i, a *job setup time* t_{ijk} may be necessary before job k can start on i. If job k is scheduled first on machine i, the setup time required is t_{i0k}. Sometimes similar jobs share a setup. Specifically, suppose that jobs are partitioned into F families according to the similarity of their production requirements so that no setup on a machine is required between two consecutively sequenced jobs of the same family. The *family setup time* on machine i when a job of family g is immediately preceded by a job of family f is s_{ifg}, or s_{i0g} if there is no preceding job. If, for each k, we can write $t_{ijk} = t_{i0k} = t_{ik}$ for all jobs $j \neq k$, or if, for each g, $s_{ifg} = s_{i0g} = s_{ig}$ for all $f \neq g$ in the case of families, then the setup times on machine i are *sequence independent*.

Job dependence arises when there are *precedence constraints* on the jobs. If job j has precedence over job k, then k cannot start its processing until j is completed. Some scheduling models allow *preemption*: the processing of any operation may be interrupted and resumed at a later time. However, we restrict our attention to nonpreemptive scheduling.

2.3 Optimality criteria

For each job j, a *due date* d_j and a positive *weight* w_j may be specified. Given a schedule, we can compute for job j: the *completion time* C_j; the *lateness* $L_j = C_j - d_j$; the *earliness* $E_j = \max\{d_j - C_j, 0\}$; the *tardiness* $T_j = \max\{C_j - d_j, 0\}$; and the *unit penalty* $U_j = 1$ if $C_j > d_j$, $U_j = 0$ otherwise.

Some commonly used optimality criteria involve the minimization of: the maximum completion time $C_{\max} = \max_j C_j$; the maximum lateness $L_{\max} = \max_j L_j$; the total (weighted) completion time $\sum_j (w_j) C_j$; the total (weighted) tardiness $\sum_j (w_j) T_j$; the (weighted) number of late jobs $\sum_j (w_j) U_j$; or the total (weighted) earliness $\sum_j (w_j) E_j$; where each maximization and each summation is taken over all jobs j.

In some situations there is a setup cost, either in addition to or instead of a setup time. This is an example where it may be appropriate to adopt a composite objective which requires a weighted sum of two or more criteria to be minimized.

2.4 Three-field representation

It is convenient to adopt the representation scheme of Graham et al. [1979]. This is a three-field descriptor $\alpha|\beta|\gamma$ which indicates problem type: α represents the machine environment, β defines the job characteristics and γ is the optimality criterion.

Let $\circ$ denote the empty symbol. The first field takes the form $\alpha = \alpha_1 \alpha_2$, where α_1 and α_2 are interpreted as follows:

- $\alpha_1 \in \{\circ, P, Q, R, F, O, J\}$
  - $\alpha_1 = \circ$: a single machine
  - $\alpha_1 = P$: identical parallel machines
  - $\alpha_1 = Q$: uniform parallel machines
  - $\alpha_1 = R$: unrelated parallel machines
  - $\alpha_1 = F$: a flow shop
  - $\alpha_1 = O$: an open shop
  - $\alpha_1 = J$: a job shop
- $\alpha_2 \in \{\circ, m\}$
  - $\alpha_2 = \circ$: the number of machines is arbitrary
  - $\alpha_2 = m$: there are a fixed number of machines m

We note that for a single-machine problem $\alpha_1 = \circ$ and $\alpha_2 = 1$, whereas $\alpha_1 \neq \circ$ and $\alpha_2 \neq 1$ for other problem types.

The second field $\beta \subseteq \{\beta_1, \beta_2, \beta_3, \beta_4\}$ indicates job characteristics as follows:

- $\beta_1 \in \{\circ, r_j\}$
  - $\beta_1 = \circ$: no release dates are specified
  - $\beta_1 = r_j$: jobs have release dates
- $\beta_2 \in \{\circ, \bar{d}_j\}$
  - $\beta_2 = \circ$: no deadlines are specified
  - $\beta_2 = \bar{d}_j$: jobs have deadlines
- $\beta_3 \in \{\circ, t_{jk}, t_j, s_{fg}, s_f\}$
  - $\beta_3 = \circ$: there are no setup times
  - $\beta_3 = t_{jk}$: there are general job setup times
  - $\beta_3 = t_j$: there are sequence independent job setup times
  - $\beta_3 = s_{fg}$: there are general family setup times
  - $\beta_3 = s_f$: there are sequence independent family setup times
- $\beta_4 \in \{\circ, prec\}$
  - $\beta_4 = \circ$: no precedence constraints are specified
  - $\beta_4 = prec$: jobs have precedence constraints

Lastly, the third field defines the optimality criterion, which involves the minimization of

$$\gamma \in \{C_{\max}, L_{\max}, \sum (w_j) C_j, \sum (w_j) T_j, \sum (w_j) U_j, \sum (w_j) E_j\}.$$

Furthermore, as indicated in Section 2.3, it is sometimes appropriate to adopt a composite objective, one component of which may be a setup cost.

2.5 Some examples

To illustrate the three-field descriptor and to indicate the combinatorial nature of some scheduling problems, we present six examples.

$1 \| \sum w_j U_j$ is the problem of scheduling jobs on a single machine to minimize the weighted number of late jobs. An optimal solution can be assumed to have the property that the on-time jobs are sequenced first in nondecreasing order of their due dates, then the late jobs are sequenced in an arbitrary order after all the on-time jobs [Moore, 1968]. Thus, a *partition* of jobs into those that are on time and those that are late defines a solution.

$1 | s_f | \sum w_j C_j$ is the problem of scheduling families of jobs on a single machine to minimize the total weighted completion time, where a sequence independent setup time is necessary whenever the machine switches to processing jobs from a different family. Jobs within each family should be sequenced in nondecreasing order of p_j/w_j (SWPT order) [Monma & Potts, 1989]. Thus, a solution is obtained by *merging* the ordered lists of jobs for the different families.

$Pm | r_j | \sum C_j$ is the problem of scheduling jobs with release dates on a fixed number m of identical parallel machines to minimize the total completion time. After being assigned to machines, a sequence of jobs is required for each machine. Thus, a schedule is constructed using *combined assignment and sequencing*.

$R \| C_{\max}$ is the problem of scheduling jobs on an arbitrary number of unrelated parallel machines to minimize the maximum completion time. A solution is specified by an *assignment* of jobs to machines; the order in which jobs are processed on a machine is immaterial.

$F2 | prec | \sum w_j T_j$ is the problem of scheduling jobs with precedence constraints in a two-machine flow shop to minimize the total weighted tardiness. Since there exists an optimal schedule in which the same processing order is used on both machines [Conway, Maxwell & Miller, 1967], a schedule is obtained by *sequencing* the jobs.

$J \| C_{\max}$ is the problem of scheduling a job shop to minimize the maximum completion time. It is a *multisequencing* problem; implicit in a schedule is a sequence of operations on each of the machines.

3 APPLYING LOCAL SEARCH

3.1 Representation of solutions

For most applications of local search to scheduling problems, there is a natural way to represent the solutions. In a sequencing problem the *natural representation* of a solution is a permutation of the integers $1, \ldots, n$; for an assignment problem it is a list of the machines to which the respective jobs are assigned. In many cases there are, as we shall see, corresponding natural ways to define both a neighborhood structure and the operators required for a genetic algorithm. In this situation, a local search method can be applied as a 'black-box' technique, with the user required only to choose the algorithm parameters. If we wish to incorporate some problem-specific knowledge, this can sometimes be achieved by making a nonstandard choice of neighborhood structure, or by restricting the solutions that are allowed. Alternatively, a different representation for solutions of the problem may be more convenient.

One example of an alternative representation occurs for the problem $P \| \sum w_j T_j$ of scheduling identical parallel machines to minimize total weighted tardiness. It is clear that, in an optimal solution, the workload assigned to different machines should be balanced; solutions in which some job starts after another machine has completed all its processing can be discarded. By using a *list scheduling* method, a list of jobs in priority order can be used to define a schedule as follows. The first unscheduled job in the list is scheduled on the machine that first becomes idle; this process is repeated until all jobs are assigned to machines.

We can give a more formal description of these ideas by distinguishing between the set $\mathscr{S}$ of feasible (or restricted) solutions to the scheduling problem and some set $\mathscr{R}$ of representations upon which the local search will be performed. We use G to denote the map from $\mathscr{R}$ to $\mathscr{S}$ which corresponds to the method that is used to generate a solution from its representation. If there is just one way to represent any possible schedule, so that G is injective, it will make little difference whether we work with $\mathscr{R}$ or $\mathscr{S}$. In this case it is possible to translate the description of the algorithm from one set to the other without essential difference; so a neighbor-

hood structure in $\mathscr{R}$ implies a neighborhood structure in $\mathscr{S}$, and similarly, genetic crossover and mutation in $\mathscr{R}$ can be translated into corresponding operators in $\mathscr{S}$. On the other hand, if G is not injective, so that a single solution may have several representations, it will not be possible to translate the local search procedure into its equivalent in $\mathscr{S}$.

Returning to the problem $P \| \sum w_j T_j$, the natural representation is a list of processing orders, where each processing order defines the sequence of jobs assigned to the corresponding machine. The restricted set $\mathscr{S}$ of solutions is the set of lists of processing orders which produced a balanced schedule, as defined above. The set of all lists of job priority orders provides us with a *priority representation* $\mathscr{R}$, and G is the list scheduling method. In this case G is not injective, as a single solution may have several representations in $\mathscr{R}$.

A more complex way to use the idea of representation is to view a schedule as arising from the application of some rule or map H to data D. Here a representation of a schedule S is specified by a pair (D, H), with $H(D) = S$. Then the set $\mathscr{R}$ is given by some subset of $\mathscr{D} \times \mathscr{H}$, where $\mathscr{D}$ is a domain of possible data and $\mathscr{H}$ is a set of possible maps. We need $\mathscr{R}$ to be chosen in such a way that $H(D)$ is a feasible schedule for every $(D, H) \in \mathscr{R}$. Storer, Wu & Vaccari [1992] suggest that $\mathscr{H}$ is formed from a collection of heuristics appropriate to the problem at hand, and that $\mathscr{D}$ consists of different perturbations of the input data. It is natural to allow variations in just one of the two components of a representation, so we perform a local search either on $\mathscr{H}$ alone or on $\mathscr{D}$ alone.

If we perform a local search on a set of heuristics $\mathscr{H}$, the heuristics are effectively the different representations of the solutions. There are many different possible ways to define such a representation. For example, consider a single-machine sequencing problem where a variety of different 'basic' heuristics could be applied (such as scheduling in order of nondecreasing due dates or of nondecreasing processing times). Composite heuristics can then be constructed by allowing a different heuristic rule to be applied at each position in the sequence.

If a local search is performed on a set $\mathscr{D}$, which contains perturbations of the input data D, we must specify a fixed heuristic H. In a sequencing problem, for example, H might be the SWPT heuristic. This can be applied to input data with perturbed weights and processing times to generate a sequence of jobs. The sequence associated with a particular perturbation is then evaluated with respect to the original data D.

3.2 Neighborhood search

Neighborhood search algorithms include descent, simulated annealing, threshold accepting, and tabu search. The starting point for a neighborhood search method is the choice of neighborhood structure; this determines which pairs of solutions are regarded as adjacent to each other, and hence the moves that are allowed. Different neighborhoods have different properties, which may make them more or less suitable depending on the particular problem at hand. It is worth mentioning three in particular.

First, neighborhoods differ in *size*, i.e., in the number of neighbors for a single solution. Small neighborhoods are preferable from the point of view of the speed at which the local search proceeds. If the neighborhood is too small, however, the final solution may be of poor quality.

Second, neighborhoods differ in the *ease* with which new solution values can be computed. Neighborhood search techniques really come into their own when a move to an adjacent solution allows a rapid update of the objective function value, rather than a complete recalculation.

Third, the underlying *topology* of the neighborhood structure significantly affects the quality of solutions generated by a neighborhood search algorithm. We can think of the definition of a neighborhood as giving rise to an objective function *surface*. Specifying which pairs of solutions are neighbors implies a distance measure between solutions (the number of moves necessary to travel between them). If we could position the possible solution points in a plane so that the distances were correctly represented, then the local search procedure would move around on the surface generated by assigning objective function values as the vertical component at each solution point. A local optimum with respect to the neighborhood definition would then correspond to a local optimum in a classical continuous optimization sense. If the surface were very bumpy, there would be many local minima and a local search procedure might have trouble locating a good solution; on the other hand, if the surface were nicely behaved, a neighborhood search procedure could easily move downhill to find a very good solution. Among the different neighborhood structures for a particular problem, we prefer one for which the associated objective function surface is reasonably smooth. Unfortunately, however, it is difficult to define an easily computable measure of smoothness.

3.3 Genetic algorithms

The picture is more complicated for genetic algorithms than for neighborhood search. As with all local search methods, we first need to decide on how solutions are to be represented (usually in the form of a string). Secondly, a method is required to produce new solutions from two 'parent' solutions using some type of crossover operation. Although crossover is often regarded as a mechanical process that is performed on the string representation, a broader interpretation may be assumed. For example, a crossover operation might involve performing a sequence of moves in some suitably defined neighborhood on one of the parent solutions, where the characteristics of the other parent indicate which moves are candidates for consideration. After reversing the role of the two parent solutions, the process is repeated to produce another new solution. Thirdly, a suitable implementation of mutation is needed. Mutation can usually be thought of as a move to a random neighbor of the current solution, within some appropriately defined neighborhood. The process of generating new solutions using crossover and mutation applied to a suitable string is known as recombination.

A representation and a crossover method have to be selected together. There is often a choice between a simple representation which implies a complex crossover mechanism, and a complex representation for which a simple crossover method can be applied. The aim is to enable two parent solutions to combine to produce a new solution that has good features from both parents. Thus, crossover should try to preserve as many as possible of the elements of the parent solutions that are thought to be important in determining the quality of a solution. These are the so-called *building blocks* from which a new solution can be generated.

3.4 Neural networks

The natural approach when tackling a scheduling problem using a neural network is to adopt the methodology of Hopfield & Tank [1985]. Essentially, the problem is represented as one of constraint satisfaction.

In the Hopfield-Tank model, the outputs of the neurons, which are usually in the range zero to one, are used to define a solution. Thus, the problem should be represented by variables that take the value zero or one, so a zero-one programming formulation is useful. The 'system dynamics' of the network provide changes of state so that an energy function is decreased until it reaches a local minimum. An appropriate energy function includes the objective function for the scheduling problem, to which penalty costs corresponding to the constraints are added. The design of the network influences the success in obtaining a global minimum of the energy function.

Zero–one (or integer) programming formulations are available for most scheduling problems. Moreover, it is straightforward to convert these formulations into an energy function that can be used in the neural network. Thus, we shall not deal explicitly with these issues.

3.5 Feasibility

As pointed out in Section 2, the set of feasible schedules for a problem may be restricted by certain characteristics of the jobs, such as deadlines or precedence constraints. In other cases, we may know some structural properties of an optimal solution a priori, and it may be advantageous to treat this structural knowledge as an effective constraint on the schedules that we consider. In either circumstance, there are essentially four possible ways to proceed:

- *Exclusion.* Infeasible solutions are never considered, either because they are never generated or because they are always discarded. In neighborhood search it may be possible to employ some mechanism to ensure that only those neighbors that represent feasible solutions are generated. Alternatively, an infeasible neighbor can be discarded immediately it is generated. In a genetic algorithm, infeasible solutions are excluded by ensuring that the representation of schedules matches precisely those solutions we wish to consider.

- *Penalization.* Infeasible solutions are allowed but are penalized by adding a penalty cost to the objective function.
- *Repair.* After an infeasible solution has been generated, it is changed or repaired so that it becomes feasible. Thus, a method is required to find a feasible solution that retains the essential characteristics of the infeasible solution which has been generated.
- *Translation.* A repair mechanism can be used differently by regarding it as a translator; the original infeasible solution is left unaltered but is assigned an objective function value based on its feasible (translated) version. The local search is carried out on possibly infeasible solutions. In general different solutions may yield the same translated solution.

In neighborhood search, the simplest strategy to adopt is exclusion. It is implemented either by using some mechanism in the neighbor generation procedure to ensure that only feasible solutions are produced, or by discarding an infeasible solution immediately it is generated. For tightly constrained problems, however, to pass from one side of an infeasibility barrier to the other, so that another part of the solution space can be searched, may be difficult or impossible under exclusion. A penalty function approach may perform better. The repair and translation strategies should be used with caution because of the computational expense associated with applying the repair mechanism. Translation has the advantage that, like penalization, it allows an infeasibility barrier to be crossed.

For genetic algorithms, the exclusion strategy may be inappropriate because of the unavailability of a representation of solutions which avoids infeasibility. A penalty function approach is possible since a standard selection mechanism eliminates high-cost solutions. Alternatively, repair or translation strategies can be used. A possible advantage of translation is that the genetic information in the infeasible solution is left undisturbed, and could reemerge fruitfully in some later generation.

4 NEIGHBORHOOD STRUCTURES AND RECOMBINATION OPERATORS

Most scheduling problems fall into one of three types, according to their natural solution space representation. Some problems have solutions that are just a sequence of jobs; some problems have solutions that are assignments of jobs to machines, where a sequence of the assigned jobs may also be specified for each machine; and some problems have solutions that are multisequences – a set of sequences of operations, one for each machine. Our discussion also includes problems in which the natural representation is a partition of jobs into two subsets and the merging of ordered lists of jobs. Other natural representations of solutions do exist, but they occur less frequently. In this section, for each type of natural representation of solutions, we discuss alternative representations, give different neighborhoods for neighborhood search algorithms, and suggest various recombination operators for genetic algorithms.

4.1 Sequencing problems

There are many machine scheduling problems in which solutions are most naturally represented by stating the order, or sequence, in which the jobs are processed. Most of the single-machine problems that we discuss in this chapter are of this type.

Representation

The *natural representation* for sequencing problems is a permutation of the integers $1, \ldots, n$. However, there are other useful representations, three of which are given below.

In a *start time representation*, the start time of each job is specified. Applying local search operators to this representation is unlikely to produce feasible start times. Nevertheless, there is an obvious repair/translation mechanism: sequence the jobs in nondecreasing order of the start times resulting from the application of the local search operators.

If there are precedence constraints on the jobs, many sequences are infeasible. To overcome this drawback, we can use a *priority representation*, which is a priority order of jobs. A job becomes available when all of its predecessors are sequenced. We construct a feasible solution by selecting the first available unsequenced job in the priority order, and sequencing it next. Similar priority representations can be defined when there are release dates or deadlines; the essential difference is how availability of jobs is defined.

For scheduling problems of a sequencing nature, the crucial decisions may relate to whether one job is sequenced before or after another. This suggests an *ordered pair representation* that records, for each possible pair of jobs, which job comes before the other in the sequence. Thus, a sequence of six jobs (B, E, F, C, A, D) could be represented by a 15-element binary string as follows:

AB	AC	AD	AE	AF	BC	BD	BE	BF	CD	CE	CF	DE	DF	EF
0	0	1	0	0	1	1	1	1	1	0	0	0	0	1

Here the 0 against AB represents the fact that A does not appear before B in the sequence, whereas the 1 against BC indicates that B is sequenced before C. After applying local search operators to such a binary string, there is little likelihood that the result is a feasible solution. Nevertheless, this is not a problem if, as suggested in Section 3.5, a repair or translation operator is applied to the string. Suppose that we construct a directed graph G from the string in which vertices correspond to the jobs $1, \ldots, n$, and for each pair of jobs j and k there is an arc (j, k), or (k, j), according to whether or not the string specifies that j precedes k. The 'best' repair/translation operator would find an acyclic subgraph of G containing a maximum number of the arcs of G, which defines a sequence. However, since this *acyclic subgraph problem* is NP-hard – feedback arc set is NP-hard [Karp, 1972] – it is more appropriate to apply a constructive heuristic. For example, a greedy heuristic may be used: select a job for the first unfilled position

in the sequence such that this decision causes as few arcs as possible in G to be deleted.

Neighborhoods

Consider first the natural representation for a sequencing problem. We give four possible neighborhoods below; each is illustrated by considering a typical neighbor of the sequence (A, B, C, D, E, F, G, H) in a problem where there are eight jobs labeled $A, \ldots, H$.

- *Transpose.* Swap two adjacent jobs. Thus, $(A, B, \mathbf{D}, \mathbf{C}, E, F, G, H)$ is a neighbor.
- *Insert.* Remove a job from one position in the sequence and insert it at another position (either before or after the original position). Thus, $(A, \mathbf{E}, B, C, D, F, G, H)$ and $(A, B, C, D, F, G, \mathbf{E}, H)$ are both neighbors.
- *Swap.* Swap two jobs that may not be adjacent. Thus, $(A, \mathbf{F}, C, D, E, \mathbf{B}, G, H)$ is a neighbor.
- *Block insert.* Move a subsequence of jobs from one position in the sequence and insert it at another position. Thus, $(A, \mathbf{D}, \mathbf{E}, B, C, F, G, H)$ is a neighbor.

This terminology is not universally adopted: *shift* is sometimes used instead of *insert*, and *interchange* instead of *swap*. Moreover, there are other possible more complex neighborhoods. For example, by analogy with the neighborhood structures that are frequently used for the traveling salesman problem, we can define *k-opt* neighborhoods that move (and possibly reverse) more than one subsequence in the schedule.

As indicated in Section 3, size is an important consideration when choosing between neighborhoods. In calculating the neighborhood sizes, we must avoid double counting; for example, starting from (A, B, C, D, E, F, G, H), inserting job B after job C has the same result as inserting job C before job B. With n jobs to be sequenced, the neighborhood sizes are $n-1$ for transpose; $(n-1)^2$ for insert; $n(n-1)/2$ for swap; and $n(n+1)(n-1)/6$ for block insert. The neighborhood with the smallest size is not necessarily preferred because the topology of the solution space also influences our choice.

Another factor that affects the choice of neighborhood is the computational effort required to evaluate a neighbor. For many single-machine problems, a rapid update of the objective function is possible. Consider, as an example, the problem $1 \| \sum T_j$ of minimizing total tardiness on a single machine. Suppose that we start with the sequence $(1, \ldots, n)$. Then, to calculate the effect of inserting job j in position k $(k > j)$, we note that the tardiness T_l of job l, for $l = j+1, \ldots, k$, is reduced by $\min\{p_j, T_l\}$, whereas the lateness of job j is increased by $\sum_{l=j+1}^{k} p_l$. Since the tardiness of the other jobs is not altered, the change in total tardiness is easily computed. Similar arguments are possible for the other neighborhoods we have given.

For the priority representation mentioned above, exactly the same neighborhoods can be used. For the start time and ordered pair representations, it is more natural to consider changes to one or more elements of the representation, rather

than moving elements around. Thus, for example, a neighborhood for the start time representation can be obtained by allowing a single start time to change, with some limit imposed on the maximum increase or decrease. A natural neighborhood for the ordered pair representation is obtained by allowing up to two of the binary elements in the string to change (from 0 to 1, or vice versa).

Recombination

We first consider a natural representation of solutions in which each string is a permutation of the integers $1, \ldots, n$. For sequencing problems there are many methods of combining two solutions to produce two new solutions. We describe three of them below. Each of our descriptions relates to crossover operations that are defined by two randomly selected crossover points. For corresponding one-point crossovers, a single crossover point is randomly chosen and, by implication, a second point occurs at the beginning or the end of the sequence.

In a *partially matched crossover*, two positions are picked at random and the sections or subsequences of the two sequences between these positions are then interchanged element by element. This is much easier to describe with an example. Suppose we start with sequences

$$\pi_1 = A\ B\ C \mid D\ E\ F\ G \mid H\ I\ J$$
$$\pi_2 = C\ H\ G \mid A\ I\ D\ B \mid J\ F\ E$$

The two strings change places between the crossover points, and outside the crossover points the reverse changes are made in order to maintain valid sequences. We obtain

$$\pi'_1 = F\ G\ C \mid A\ I\ D\ B \mid H\ E\ J$$
$$\pi'_2 = C\ H\ B \mid D\ E\ F\ G \mid J\ A\ I$$

Here A is swapped with D and then D with F, to give the first element in π'_1. B is swapped with G to give the second element, and so on.

The *insertion crossover* is a version of the crossover operation which has been used successfully by Mühlenbein and coworkers for the traveling salesman problem [e.g., Mühlenbein, Gorges-Schleuter & Krämer, 1988]. Two positions are picked at random and the sections or subsequences between these points are then inserted into the other sequence at one of the point where an element in the subsequence occurs (the choice being made randomly). Elements in the inserted subsequence are removed from the other places where they occur, and the gaps closed up. If we take π_1 and π_2 as above, this crossover might produce

$$\pi'_1 = \mid A\ I\ D\ B \mid C\ E\ F\ G\ H\ J$$
$$\pi'_2 = C\ H\ A\ I \mid D\ E\ F\ G \mid B\ J$$

We have inserted the substring from π_2 in place of A in π_1, and the substring from π_1 in place of D in π_2. The end result is that a subsequence from one parent is

inserted into the other, with the remaining elements in the sequence occurring in the same order as before.

A *reorder crossover* picks two positions at random, as before, and then the subsequences between them are reordered to match the order of the elements in the other sequence. If we take π_1 and π_2 as above, this crossover might produce

$$\pi_1' = A\ B\ C \mid G\ D\ F\ E \mid H\ I\ J$$
$$\pi_2' = C\ H\ G \mid A\ B\ D\ I \mid J\ F\ E$$

These three types of crossover can also be applied to the priority representation in which the priority order is a sequence of jobs. For start time and ordered pair representations, a standard crossover can be used before applying the necessary repair/translation operators.

4.2 Assignment and partitioning problems

There are several scheduling problems whose solution can be viewed as an assignment of the jobs to machines. One example given in Section 2 is the problem $R \| C_{\max}$, which requires jobs to be assigned to unrelated parallel machines to minimize the maximum completion time. Another example involving assignment is $P \| L_{\max}$, in which jobs are to be scheduled on identical parallel machines to minimize the maximum lateness; once an assignment of jobs to machines is decided, the optimal schedule on each machine can be obtained by sequencing jobs in EDD order (nondecreasing order of due dates) [Jackson, 1955].

Some single-machine scheduling problems require jobs to be partitioned into two sets. An example is $1 \| \sum w_j U_j$, in which jobs are to be scheduled on a single machine to minimize the weighted number of late jobs. A partition of jobs into those which are late and those which are on time is sufficient to determine a schedule. The partition is constrained because not all subsets of jobs can be scheduled on time. Another example of a partitioning problem is $1 | d_j = d | \sum (w_j E_j + w_j' T_j)$, in which jobs with a common due date d are to be scheduled on a single machine to minimize the total weighted earliness plus total weighted tardiness. Assume that $d \geqslant \sum_j p_j$. Baker & Scudder [1990] generalize the properties of an optimal schedule derived by Kanet [1981] for unit weights. Specifically, there is no idle time between jobs, and one job completes at time d. Furthermore, jobs that are completed no later than time d are sequenced in order of nonincreasing p_j/w_j (longest weighted processing time or LWPT order) and jobs completed after time d are sequenced in order of nondecreasing p_j/w_j' (SWPT order). Thus, a solution is specified by a partition of the jobs into those that are completed no later than time d and those that are completed after time d.

A partitioning problem is obviously a special case of an assignment problem, and it is convenient to treat the two types of problems together.

Representation

Suppose that we have a problem in which n jobs are to be assigned to m machines. The *natural representation* for an assignment problem is a set of m lists, where each list contains the jobs assigned to one particular machine. Equivalently, this representation can be a single list containing the respective machine assignments of the n jobs.

As in the case of sequencing problems, we can also adopt a *priority representation*. To obtain an assignment of jobs to machines from the priority order, *list scheduling* is applied. More precisely, whenever a machine becomes idle, the next job on the priority list is scheduled on that machine. Since the priority representation is a sequence of jobs, the neighborhoods and recombination operators of Section 4.1 are applicable.

Neighborhoods

For the natural representation of solutions to assignment problems, we give three possible neighborhoods:

- *Reassign.* Remove a job from one machine and reassign it to another.
- *Swap.* Swap two jobs from different machines by reversing their machine assignments.
- *2-Reassign.* Remove either one or two jobs and reassign them to different machines.

More generally, we could define a *k-reassign* neighborhood which allows the reassignment of up to k jobs. Neighborhood sizes are $n(m-1)$ for reassign and $n(m-1)[1+(n-1)(m-1)/2]$ for 2-reassign. For swap, the maximum possible neighborhood size occurs when jobs are evenly divided between the machines. Thus, an upper bound on the neighborhood size is $\lceil n/m \rceil^2 m(m-1)/2$.

It may be preferable to reduce the neighborhood size using an exclusion mechanism. We define a job to be *critical* if any small delay to its start time causes an increase in the objective function value. As an example, for $R \| C_{\max}$, jobs on a most heavily loaded machine are critical for the maximum completion time objective. To obtain an improved solution, some critical job must be assigned to a different machine. Thus, we define the *critical reassign*, *critical swap*, and *critical k-reassign* neighborhoods, to be subsets of the original neighborhoods in which at least one critical job is moved to another machine.

Neighbors for assignment problems can often be evaluated quickly if appropriate values for the current solution are stored. For example, consider the problem $P \| \sum w_j C_j$ of scheduling jobs on identical parallel machines to minimize the total weighted completion time. Given an assignment of jobs to machines, the jobs on each machine are sequenced in SWPT order. Suppose that jobs $1, \ldots, k_1$ are sequenced in this order on some machine i_1, and jobs $k_2, \ldots, n$ are sequenced in this order on another machine i_2, and we wish to calculate the effect of removing job j from machine i_1 (where $j \leqslant k_1$) and reassigning it on machine i_2. Assume that,

in the new SWPT ordering on machine i_2, job j is sequenced immediately after job l (where $k_2 \leqslant l \leqslant n$). The total weighted completion time of jobs $j+1, \ldots, k_1$ on machine i_1 decreases by $p_j \sum_{h=j+1}^{k_1} w_h$, the total weighted completion time of jobs $l+1, \ldots, n$ on machine i_2 increases by $p_j \sum_{h=l+1}^{n} w_h$, and the weighted completion time of job j decreases by $w_j(\sum_{h=1}^{j-1} p_h - \sum_{h=k_2}^{l} p_h)$ (which may be negative). Since the weighted completion time of other jobs is unaltered, the change in the total weighted completion time is easily computed if, for the current schedule, partial, sums of weights and processing times on each machine are available. A similar efficient recompuation of the total weighted completion time is also possible for the swap and 2-reassign neighborhoods.

Recombination

Consider the natural representation of solutions. The application of genetic algorithms in this case is particularly easy, since a string representation is immediately available by listing for each job the machine to which it is assigned. Then crossover can be carried out using a standard one- or two-point method. In contrast to sequencing problems where standard crossover operations fail to yield a sequence, any list of machine numbers defines a possible assignment. It is worth noting, however, that the operation of the genetic algorithm will be affected by the order of the elements in the string. The jobs that are placed close together on the string are more likely to retain their relationship with each other under the standard crossover operation. Unfortunately, in most such problems, it is very hard to see what might be an appropriate ordering of the jobs. One option is to use a different form of crossover operation. Using the *uniform crossover*, two solutions can be mated by choosing randomly at each point in the string which parent should provide the element for the child string. The other child is obtained by making the opposite choice at each point. This will make the order of the elements in the string unimportant. If we think of genetic algorithms as working with building blocks, out of which good solutions can be put together, then the building blocks will be just the individual job assignments.

4.3 Combined assignment and sequencing problems

There are several problems in which an assignment of jobs to machines and a sequencing of the jobs on the machines are both necessary (e.g., the problem $P|r_j|\sum C_j$ of scheduling jobs with release dates on m identical parallel machines to minimize the total completion time). Since they contain elements of pure sequencing and assignment problems, generalizations of some of the approaches discussed above are appropriate.

Representation

The *natural representation* of solutions is a set of m ordered lists, where each list contains the sequence of jobs assigned to one particular machine.

Kanet & Sridharan [1991] suggest a *start time representation* which specifies, for each job, a vector containing its machine assignment and start time. Local search can be applied to this list of vectors in much the same way as for the start time representation of a sequencing problem, except that changes in job assignments are possible as well as changes in start times. As before, a repair/translation procedure is necessary for the start times obtained from the local search operators: for each machine the assigned jobs are sequenced in nondecreasing order of these start times.

Since the *priority representation* for pure assignment problems uses list scheduling to create a schedule, it also provides a valid representation for our combined assignment and sequencing problems. The neighborhoods and recombination operators of Section 4.1 can be applied to the priority sequence.

Neighborhoods

For a natural representation of solutions, the neighborhoods described for the pure sequencing problems can be generalized. Thus, *insert* removes a job from its current position on some machine and inserts it in a different position, either on the same machine or on a different machine. Also, jobs on the same or on different machines can be interchanged using *swap*.

Recombination

It is difficult to specify a method for recombination when the natural representation is adopted. For the start time representation, however, it is straightforward to apply crossover to lists of vectors. As pointed out in Section 4.2, different job orderings give different lists, and it is difficult to select a list that is suitable if standard one- or two-point crossover is used. However, uniform crossover avoids the need to choose a job ordering.

4.4 Multisequencing problems

Most multi-stage scheduling problems, such as a flow shops and job shops, require operations to be sequenced on several machines, so we call them multisequencing problems.

Representation

We first describe a *natural representation*. Flow shop and job shop schedules are defined by a set of sequences, one for each machine. For the open shop, in addition to a sequence for each machine, the solution must also specify a machine routing for each job. Due to the interrelationships between the different sequences, the obvious generalizations of the approaches outlined in Section 4.1 are likely to be ineffective. For example, suppose that the insert neighborhood is used for some flow shop problem in which different sequences of jobs are allowed on

different machines. If a job is moved from one of the initial positions on a machine to one of the final positions, then the resulting schedule is likely to be of poor quality unless corresponding adjustments are made to the sequences on other machines. Now consider a job shop problem. If job j has to be processed first by machine 1 and then by machine 2, and job k is processed by these two machines in the reverse order, we cannot have a schedule in which k comes before j on machine 1 and k comes after j on machine 2. A simple-minded approach may therefore prove infeasible. Similar disadvantages of ineffectiveness and infeasibility also occur for the open shop.

A *priority representation* helps to counter these drawbacks. For the flow shop and job shop a priority order of operations is given for each machine. An operation becomes available when the previous operation on the same job is completed. When a machine becomes idle, an available operation with highest priority is started (or if there is none, the first operation to become available is scheduled next, ties being broken by the priority rule). This always creates a nondelay schedule. Although there may be no optimal nondelay schedule, this approach can still be effective in generating near-optimal solutions. Furthermore, this potential loss of optimality can be overcome by modifying the heuristic to generate active rather than nondelay schedules (an active schedule is one in which no operation can be processed earlier without delaying the start of another operation). For the open shop we can use a similar representation. Suppose that we specify for each machine a priority order for the jobs, and for each job a sequence of machines that defines its route. A schedule can then be constructed using the same method as for a job shop.

Neighborhoods

The priority representation rather than the natural representation of solutions is generally recommended for the reasons indicated above. Using the priority representation of solutions for the flow shop or job shop, it is straightforward to apply neighborhood search. Since the priority order on each machine is a sequence, it is possible to use one of the neighborhoods given in Section 4.1 for pure sequencing problems, such as insert or swap. Similarly, for the open shop, a neighborhood is defined by allowing a change in either one of the priority orders or in one of the job route sequences.

It is also possible to use a natural representation for the flow shop or job shop, but incorporate an exclusion mechanism of the type introduced in Section 4.2, as proposed by Van Laarhoven, Aarts & Lenstra [1992]. Consider the minimization of the maximum completion time. As in Section 4.2, we define an operation to be *critical* if any small delay to its start time causes an increase in the objective function value. Furthermore, a maximal sequence of critical operations is called a *critical path*, and a corresponding maximal subsequence of critical operations on the same machine is called a *block*. To obtain an improved solution, some critical operation must start earlier by a suitable reordering of the operations in a block. If the transpose neighborhood is used for each of the sequences, a neighbor is considered only if two operations in the same block are swapped;

other neighbors are excluded. We refer to this neighborhood as *critical transpose*. It is shown by Van Laarhoven, Aarts & Lenstra that any solution obtained by critical transpose is feasible. A further restriction of this neighborhood is motivated by the observation that no improvement in the solution results by a reordering of the operations of a block if the first operation still precedes the others in the block and the last operation also remains at the end. The *critical end transpose* neighborhood is a restricted version of the critical transpose neighborhood in which the first or last operation of a block is transposed. Similarly, we define the *critical end insert* neighborhood as a restricted version of insert in which an operation is removed from a block and placed immediately before the first or immediately after the last operation in the block. For other optimality criteria, this type of exclusion approach is likely to be less effective because the number of critical operations is larger.

Under a natural representation, a *machine reschedule* neighborhood can be adopted for flow shop and job shop problems. In this neighborhood a machine is selected for rescheduling, but the processing order on the other machines remains fixed. A solution (either optimal or heuristic) of the resulting single-machine subproblem provides the new schedule. For the open shop, a similar approach can be used to reschedule a machine or to reroute a job through the machines. A more general *k-machine reschedule* neighborhood allows the rescheduling of up to k machines; this can be achieved by considering a k-machine subproblem, or by some sequence of k single-machine subproblems.

Recombination

For flow shop, job shop, and open shop problems, the priority representation can be used for genetic algorithms. In each case a solution is represented by a set of sequences and one of the methods of Section 4.1 can be used.

4.5 Merging ordered lists of jobs

Sometimes a problem with a more complex solution structure arises because we can deduce some properties of an optimal solution, and it is advantageous to restrict the search to solutions which have these required characteristics. An example of this phenomenon occurs for the problem $1|s_f|\sum w_j C_j$ in which families of jobs with associated sequence-independent setup times are to be scheduled on a single machine to minimize the total weighted completion time. This can be viewed as a sequencing problem. Since jobs within each family must be sequenced in a SWPT order [Monma & Potts, 1989], it can also be regarded as a problem of merging ordered lists of jobs, where each ordered list contains all the jobs in one family.

Representation

Since we know the order in which the jobs within a single family are to be processed, the *natural representation* of a schedule is obtained by simply indi-

cating the family of the job at each position in the schedule. Thus, we can represent a schedule for a problem with five jobs in each of three families by a string of the form $(A, C, C, A, A, A, B, B, B, C, C, C, B, A, B)$, where A, B, and C are the families.

It is sometimes the case that we can deduce further properties of an optimal schedule. In particular, if we decide on how each family is split up into batches, where a batch contains jobs that are to be scheduled contiguously, then we may determine the sequence of these batches. For example, in the problem $1|s_f|\sum w_j C_j$ there is a generalized SWPT rule which applies to complete batches as well as to individual jobs. This allows an alternative *batch-based representation*, which is obtained by using a binary string to mark the positions in each family at which a new batch begins. The schedule above is described by three binary strings $(1, 1, 0, 0, 1), (1, 0, 0, 1, 1), (1, 0, 1, 0, 0)$, where each string corresponds to a single family. If the jobs of family A are numbered $1, \ldots, 5$, then the first string $(1, 1, 0, 0, 1)$ indicates that the jobs of A are split into three batches that contain job 1, jobs 2 to 4, and job 5, respectively. Each of the binary strings must start with a 1, so we could omit the first element in each. Note also that the same schedule could arise from more than one set of binary strings; although we may choose to start a new batch at a particular position among the jobs from one family, the mechanism for sequencing the batches could schedule two batches of the same family consecutively.

Neighborhoods

Using the natural representation, it is necessary to retain the same number of elements, of each type in any changed string. This makes it appropriate to use the same type of neighborhoods as for the natural representation of sequencing problems (transpose, insert, swap, etc.). For this approach, some of the possible moves will not alter the resulting schedule (e.g., swapping two jobs of the same family). For each of the three neighborhoods transpose, insert, and swap, it is straightforward to specify the neighbors that will yield a different schedule and then to restrict attention to these neighbors.

One class of neighborhoods in the batch-based representation is defined by changing one or more elements in the solution representation (0 to 1, or vice versa). Once we have taken account of the fixed 1 at the start of each family's binary string, all other binary strings represent feasible schedules.

Recombination

If we use a natural representation, the following methods of performing a crossover operation are possible. First, we could apply a version of one of the crossovers used for a pure sequencing problem. However, this approach effectively ignores the special structure of the merged strings problem. An alternative is to use a standard one- or two-point crossover operation followed by a translation/repair mechanism to obtain the correct number of elements from each

family. It is more satisfactory, however, to use a batch-based representation. In this case we can concatenate the strings for different families into a single string and then use a standard crossover operation.

5 SINGLE-MACHINE PROBLEMS

In this section we indicate how local search can be applied to a variety of single-machine scheduling problems. We exclude from our discussion problems of minimizing the maximum completion time; most problems of this type are polynomially solvable, so local search is inappropriate. For the other optimality criteria, we briefly review the main results and, where appropriate, provide a guide to problem 'hardness' by indicating the size of problem for which the best currently available (enumerative) algorithm can generate an optimal solution.

Since many single-machine scheduling problems require jobs to be sequenced, the guidelines given in Section 4.1 are especially useful.

5.1 Maximum lateness

The problem $1 \| L_{\max}$ is solved in $O(n \log n)$ time by Jackson's earliest due date (EDD) rule [Jackson, 1955]: jobs are sequenced in order of nondecreasing due dates. When there are precedence constraints, Lawler [1973] shows there exists an optimal schedule in which a job that has the largest due date among those with no successors is processed last. Repeated application of this result yields an optimal schedule for $1|prec|L_{\max}$ in $O(n^2)$ time. Although Lenstra, Rinnooy Kan & Brucker [1977] show that $1|r_j|L_{\max}$ is strongly NP-hard, branch-and-bound algorithms are able to solve large instances. For example, Carlier [1982] solves 10000-job problems using an ingenious branching rule and lower bounds obtained by allowing preemption. Monma & Potts [1989] show that $1|s_f|L_{\max}$ is solvable by dynamic programming in $O(F^2 n^{2F})$ time (where F is the number of families), and Bruno & Downey [1978] prove NP-hardness in the ordinary sense for arbitrary F.

We now discuss the design of local search algorithms for the problem $1|r_j|L_{\max}$. The problem is of special interest since various algorithms for the job shop problem $J \| C_{\max}$ are based on solving a subproblem which is equivalent to $1|r_j|L_{\max}$. The search for an optimal solution to $1|r_j|L_{\max}$ can be restricted to *active* schedules, in which no job can be scheduled earlier without delaying the start of another job. Suppose first that a natural representation of solutions as sequences is adopted, so that there is a corresponding schedule in which each job is processed as early as possible subject to its release date and the machine availability. One of the approaches of Section 4.1 can be applied, and a repair/translation procedure used if a nonactive schedule is generated. An alternative and possibly preferable approach is to use a priority representation. We construct an active schedule by processing next the first unscheduled job in the priority order for which its start time is strictly smaller than the earliest possible

completion time among all jobs that can be processed next. The methods of Section 4.1 can be applied to this priority order.

For $1|s_f|L_{\max}$ Monma & Potts [1989] show that jobs within each family are sequenced in EDD order. Thus, the problem requires the merging of ordered lists of jobs, where each list contains all jobs of a family in EDD order. Moreover, by associating a due date with each batch, the EDD rule can be applied to sequence the batches. Thus, the approaches of Section 4.5 that use a batch-based representation can be applied.

5.2 Total weighted completion time

The basic problem $1\|\sum w_jC_j$ is solved in $O(n \log n)$ time by Smith's shortest weighted processing time (SWPT) rule [Smith, 1956]: jobs are sequenced in order of nondecreasing ratios p_j/w_j. In the case that jobs have unit weights and have deadlines, Smith generalizes the SPT rule (the shortest processing time rule, which is a special case of the SWPT rule) by showing that, among jobs j for which $\bar{d}_j \geqslant \sum_{k=1}^{n} p_k$, a job with the largest processing time is processed last. Repeated application of this result yields an optimal schedule for $1|\bar{d}_j|\sum C_j$ in $O(n \log n)$ time. Most other extensions of the basic model are strongly NP-hard, including $1|r_j|\sum C_j$ [Lenstra, Rinnooy Kan & Brucker, 1977], $1|prec|\sum C_j$ [Lawler, 1978; Lenstra & Rinnooy Kan, 1978] and $1|\bar{d}_j|\sum w_jC_j$ [Lenstra, Rinnooy Kan & Brucker, 1977]. Although Ahn & Hyun [1990] and Ghosh [1994] show that $1|s_f|\sum C_j$ and $1|s_f|\sum w_jC_j$ are solvable by dynamic programming in $O(F^2n^F)$ time, the complexity of these problems is open when F is arbitrary.

Branch-and-bound algorithms proposed by Chu [1992] and Belouadah, Posner & Potts [1992] for $1|r_j|\sum C_j$ and $1|r_j|\sum w_jC_j$ are capable of solving instances with up to 100 and 40 jobs, respectively. Both algorithms rely heavily on dominance rules to restrict the search. The design principles of local search algorithms for $1|r_j|L_{\max}$ given in Section 5.1 can also be used for $1|r_j|\sum(w_j)C_j$. Thus, a natural representation of solutions can be used with a repair/translation mechanism applied to generate active schedules, or a priority representation can be adopted and the active schedule generation procedure of Section 5.1 used to create a schedule from the representation.

For $1|\bar{d}_j|\sum w_jC_j$ the branch-and-bound algorithms of Posner [1985] and Potts & Van Wassenhove [1983] solve instances with up to 40 jobs. Suppose that a natural representation of solutions is adopted. Infeasible sequences can be discarded in neighborhood search. It is also possible to enforce the deadline constraints using a penalty function approach. A priority representation may alternatively be adopted. To construct a feasible sequence from a priority order, among jobs j for which $\bar{d}_j \geqslant \sum_{k=1}^{n} p_k$, the job that is nearest to the end of the priority order is processed last. Repeated application of this rule yields the required feasible sequence.

We now consider the problem $1|prec|\sum w_jC_j$. Computational results obtained by Potts [1985a] with his Lagrangean-based branch-and-bound algorithm show that instances with 100 jobs can be solved. Potts also proposes a descent

algorithm, which uses a block insert neighborhood on the natural representation of solutions and in which infeasible solutions are discarded. However, other neighborhood search algorithms can be designed. For example, the precedence constraints can be enforced using a penalty function approach. Another option is to use an ordered pair representation: by applying a suitable repair/translation mechanism, the resulting solution is guaranteed to be feasible (the greedy heuristic of Section 4.1 for the acyclic subgraph problem can be modified to ensure that all precedence constraints are satisfied).

For $1|s_f|\sum w_jC_j$ Mason & Anderson [1991] derive a branch-and-bound algorithm which relies mainly on dominance rules to restrict the search. Their computational results for $1|s_f|\sum C_j$ indicate that instances with up 30 jobs can be solved. Recall from Section 4.5 that the problem can be regarded as one of merging lists, where each list contains the jobs of a family in SWPT order. Adopting a natural representation of solutions, Ahn & Hyun [1990] propose a descent algorithm for $1|s_f|\sum C_j$ which uses the block insert neighborhood. By restricting the choice of blocks, it is possible to ensure that jobs within each family are sequenced in SWPT order. Mason [1992] proposes a genetic algorithm which uses the batch-based representation. Crauwels, Potts & Van Wassenhove [1997] design various neighborhood search algorithms. In multistart descent, simulated annealing, and threshold accepting, they use the block insert neighborhood of Ahn & Hyun, and the two latter methods use temperatures and threshold values that follow a periodic pattern. Their tabu search algorithm uses the restricted insert neighborhood, and a tabu list of length 7 stores the job that is inserted and prevents it from moving again. In a computational comparison of Mason's algorithm with these neighborhood search algorithms, Crauwels, Potts & Van Wassenhove find that all five methods generate solutions of high quality. Based on solution quality and computation time, tabu search is preferred to the other methods. These results are presented in Section 8.1.

Herrmann & Lee [1995] propose a genetic algorithm for the problem $1|\bar{d}_j, s_{fg}|\sum C_j$, which is shown by Bruno & Downey [1978] to the NP-hard. They use a representation which consists of binary encoding of perturbations of the original deadlines. To obtain a solution from the representation, a backward scheduling heuristic is used which aims to minimize the time spent on setups and to process the longer jobs as late as possible. The resulting schedule is not guaranteed to be feasible with respect to the original (or the perturbed) deadlines. Thus, a penalty function approach is used to drive the solution towards feasibility.

Laguna, Barnes & Glover [1991] propose tabu search algorithms for $1\|\sum w_jC_j + \sum c_{jk}$, in which there are general job setup costs and the objective is to minimize the total weighted completion time plus the total setup cost. They adopt a natural representation of solutions as a sequence, and compare a restricted version of the swap and insert neighborhoods which prohibits moves that change the position of a job by more than $n/2$. By storing partial sums of weights and processing times for the current sequence, candidate moves can be evaluated in constant time. For a swap move that interchanges the jobs in positions

j and k, where $j < k$, the tabu list entry prevents the first of these jobs from occupying the first j positions. After an insert move, the job that is removed and inserted in another position is stored on the tabu list and prevented from moving again. Computational results with tabu lists of length 7 indicate that the insert neighborhoods yields better quality solutions than the swap neighborhood. However, a slight improvement is observed, especially for larger problems, if a combined swap and interchange neighborhood is adopted. This approach is extended by Laguna, Barnes & Glover [1993] to the problem $1|t_{jk}|\sum w_j C_j + \sum c_{jk}$, in which there are job setup times and costs. Glover & Laguna [1991] and Laguna & Glover [1993] use 'target analysis' to develop an improved algorithm for $1\|\sum w_j C_j + \sum c_{jk}$. Target analysis is a technique which 'learns' heuristic rules that are appropriate for a class of problems. In this case, whenever the tabu search algorithm is forced to make a nonimproving move, an attempt is made to diversify the search by penalizing neighbors according to the frequency with which, in previous iterations, the same job pair has been swapped or the particular insert job is placed just before the same job. Results indicate that this modified tabu search algorithm is superior to the original version.

Arizono, Yamamoto & Ohta [1992] propose a neural network approach for $1|t_{jk}|\sum C_j$ (although they work with the equivalent problem $1|t_{jk}, \bar{d}_j = \bar{d}|\sum E_j$, in which jobs have a common deadline). Their model uses the output of a neuron to define whether or not a job occupies a particular position in the sequence.

5.3 Total weighted tardiness

For many years the complexity of $1\|\sum T_j$ remained open. In a recent paper, however, Du & Leung [1990] show that the problem is NP-hard in the ordinary sense. Lawler [1977] shows that it is pseudopolynomially solvable by deriving an $O(n^4 \sum_j p_j)$ algorithm, which is based on decomposition. More precisely, a job with the largest processing time partitions the problem into two subproblems: jobs with the smallest due dates are sequenced before this partition job, and the remaining jobs are sequenced after it. A search is performed for the optimal position of the partition job. Using a dynamic programming algorithm of Schrage & Baker [1978] to solve subproblems, Potts & Van Wassenhove [1982] develop this decomposition approach to solve instances with up to 100 jobs.

Lawler [1977] and Lenstra, Rinnooy Kan & Brucker [1977] show that $1\|\sum w_j T_j$ is strongly NP-hard. Successful enumerative algorithms rely heavily on elimination criteria of the type derived by Emmons [1969] and Rinnooy Kan, Lageweg & Lenstra [1975], which give conditions under which certain jobs can be assumed to precede others in the search for an optimal schedule. The branch-and-bound algorithm of Potts & Van Wassenhove [1985], which uses a relatively weak but quickly computed lower bound, can solve problem instances with up to 40 jobs. A survey of algorithms is given by Abdul-Razaq, Potts & Van Wassenhove [1990].

There are various descent and simulated annealing algorithms which use natural representations of solutions as sequences and employ one of the neigh-

borhoods of Section 4.1. The algorithm of Wilkerson & Irwin [1971] for $1 \| \sum T_j$ resembles a descent algorithm that applies the transpose neighborhood to the EDD sequence. Chang, Matsuo & Tang [1990] also consider descent algorithms for this problem. They show that the ratio of total tardiness given by a descent algorithm to the minimum total tardiness (for instances in which the minimum total tardiness is positive) can be arbitrarily large if the transpose or swap neighborhood is used, although the ratio is finite for the insert neighborhood.

Matsuo, Suh & Sullivan [1987] apply simulated annealing to $1 \| \sum w_j T_j$. In their algorithm the probability of accepting a worse solution is independent of the objective function values. Based on computational results with the transpose neighborhood, they find that a systematic search of the neighborhood outperforms a random search and that the use of a good initial sequence accelerates the process of finding a near-optimal solution. Potts & Van Wassenhove [1991] compare descent and simulated annealing algorithms with other special-purpose heuristics for $1 \| \sum T_j$ and $1 \| \sum w_j T_j$. For descent and simulated annealing, they use the swap neighborhood and impose a strict limit on run times. The descent method performs surprisingly well relative to simulated annealing for both the unweighted and weighted problems if moves to solutions with the same objective function value are accepted (which commonly occur if two early jobs are interchanged). For $1 \| \sum T_j$, however, a special-purpose decomposition heuristic is slightly superior. On the other hand, a finely tuned simulated annealing algorithm that employs a descent routine gives the best results for $1 \| \sum w_j T_j$.

Crauwels, Potts & Van Wassenhove [1998] propose various local search algorithms for $1 \| \sum w_j T_j$. For the natural sequence representation, the swap neighborhood is used in multistart descent, simulated annealing, threshold accepting and tabu search methods. A novel partition representation uses binary strings, where each element indicates whether the corresponding job is on time or late. To convert the partition representation into a sequence, a procedure is used which first sequences those jobs that are indicated as on time in EDD order. Any job that cannot be completed by its due date in this EDD sequence is removed, and is subsequently regarded as late. Then, where possible, late jobs are scheduled between the on-time jobs, provided this does not make them finish before their due dates, and finally any remaining late jobs are scheduled at the end of the sequence in SWPT order. Lastly, descent is applied to the resulting sequence, where the descent method uses the transpose neighborhood of Section 4.1. For the partition representation, neighborhood search algorithms use the reassign neighborhood and a genetic algorithm uses a modified version of the recombination operator of Section 4.2. Computational tests for problems with up to 100 jobs are used to compare the performance of the algorithms. The best-quality solutions are obtained with the genetic algorithm that uses the partition representation, and with the two tabu search algorithms that use the sequence and partition representations.

5.4 Weighted number of late jobs

To specify a solution of the problem $1 \| \sum w_j U_j$, it is sufficient to partition the jobs into two subsets: those which are on time and those which are late. A schedule is constructed from the partition by sequencing first the on-time jobs in EDD order; then the late jobs are sequenced arbitrarily after all on-time jobs. For $1 \| \sum U_j$ an algorithm of Moore [1968] solves the problem in $O(n \log n)$ time: jobs are added in EDD order to the end of a partial schedule of on-time jobs, and if the addition of job j results in it being completed after time d_j, a job in the partial schedule with the largest processing time is removed and deemed to be late.

The problem $1 \| \sum w_j U_j$ is shown by Karp [1972] to be NP-hard in the ordinary sense. However, it is pseudopolynomially solvable, since Lawler & Moore [1969] propose a dynamic programming algorithm that requires $O(n \sum_j p_j)$ time. Potts & Van Wassenhove [1988] present an $O(n \log n)$ procedure that solves the linear programming relaxation of a natural integer programming formulation of $1 \| \sum w_j U_j$. The resulting lower bounding scheme has two uses: first in reduction tests that are designed to eliminate jobs from the problem, and second in a branch-and-bound algorithm. In computational tests for problems with up to 1000 jobs, Potts & Van Wassenhove compare the performance of the algorithm of Lawler & Moore with their own branch-and-bound algorithm.

Since $1 \| \sum w_j U_j$ is a partitioning problem, the local search approaches of Section 4.2 are applicable. However, an arbitrary partition may not yield a feasible subset of on-time jobs, so it is appropriate to suggest a repair/translation procedure. We propose the following method which is a variant of Moore's algorithm. If some job j is completed after time d_j, selected jobs in the partial schedule up to and including job j are removed and deemed to be late until either job j is on time or j itself is removed: at each stage the job k to be removed is chosen so that p_k/w_k is as large as possible.

5.5 Total weighted earliness and tardiness

Most research that considers earliness as an optimality criterion also includes a tardiness component. A thorough survey of this area of scheduling is given by Baker & Scudder [1990]. One widely studied model is $1 | d_j = d | \sum (w_j E_j + w'_j T_j)$, in which jobs have a common due date d. For the case of unit weights, Hall, Kubiak & Sethi [1991] show that this problem is NP-hard in the ordinary sense, and they propose an $O(n \sum_j p_j)$ pseudopolynomial dynamic programming algorithm. As indicated in Section 4.2, Baker & Scudder show that the following results of Kanet [1981] generalize to the case of arbitrary weights: there is no idle time between jobs, and jobs that are completed no later than time d are sequenced in LWPT order, whereas jobs completed after time d are sequenced in SWPT order. This latter result is known as the V-shaped property. If $d \geqslant \sum_j p_j$, Kanet derives an $O(n \log n)$ algorithm for $1 | d_j = d | \sum (E_j + T_j)$ by showing that the job in position $\lceil n/2 \rceil$ completes at time d, and by assigning jobs in nonincreasing order of processing times alternately to the first and last unfilled position in the

sequence. Closely related to these earliness–tardiness problems is $1 \| \sum (C_j - \bar{C})^2$, where $\bar{C}$ is the average completion time, for which the objective is to minimize the completion time variance. Eilon & Chowdhury [1977] show that the V-shaped property holds in this case, although this result does not extend to $1 \| \sum w_j (C_j - \bar{C})^2$ [Cai, 1995]. Moreover, these problems are NP-hard in the ordinary sense, but $1 \| \sum (C_j - \bar{C})^2$ is pseudopolynomially solvable [Kubiak, 1993], and $1 \| \sum w_j (C_j - \bar{C})^2$ is open with respect to pseudopolynomial solvability.

Any problem in which the V-shaped property holds can be treated as one of partitioning. Thus, the approaches of Section 4.2 are applicable. Mittenthal, Raghavachari & Rana [1993] propose a simulated annealing algorithm which is applicable when the V-shaped property holds. Adopting the natural representation, they use the reassign neighborhood and a restricted swap neighborhood in which the only jobs that are eligible for interchange are adjacent pairs in an SPT ordering. Their algorithm first applies two descent procedures, the first of which uses the reassign neighborhood and the second uses the restricted swap neighborhood, and then applies simulated annealing with the restricted swap neighborhood. Computational results for $1 \| \sum (C_j - \bar{C})^2$ with instances containing up to 20 jobs show that the algorithm generates an optimal solution for each test problem. Lee & Kim [1995] propose a parallel genetic algorithm for $1 | d_j = d | \sum (w_j E_j + w'_j T_j)$ which also uses the natural representation for partitioning problems. Note, however, that the job identified to finish before or at time d may actually be completed after time d when a schedule is constructed, and vice versa. Rather than a standard mutation operator, Lee & Kim suggest recalculating the binary values in the string according to which jobs are actually late. Parallel implementation is carried out by arranging a number of subpopulations in a ring. Each subpopulation evolves independently of the others, except that the best solution in each subpopulation is communicated to each of the two neighboring populations. Computational results with the genetic algorithm, using population sizes that vary between 100 and 200, show that the average deviation of the solution value from the optimum is less than 0.25% for 40-job problems. Gupta, Gupta & Kumar [1993] propose a genetic algorithm for $1 \| \sum w_j (C_j - \bar{C})^2$ in which they adopt a natural representation of solutions as a sequence and use the partially matched crossover described in Section 4.1.

For $1 \| \sum (w_j E_j + w'_j T_j)$ an optimal schedule may include machine idle time between jobs. However, for a given processing order of jobs, procedures of Garey, Tarjan & Wilfong [1988] and Davis & Kanet [1993] can be used to construct an optimal schedule. Thus, solutions can be represented as sequences, and the local search methods of Section 4.1 are applicable. Yano & Kim [1991] adopt this approach by using the transpose neighborhood in their descent algorithm. Computational results for problems with up to 20 jobs in which the weights are proportional to the processing times of the respective jobs indicate that their algorithm consistently generates an optimal solution.

Woodruff & Spearman [1992] propose a tabu search approach for a variant of the problem $1 | \bar{d}_j, s_{fg} | \sum w_j E_j + \sum c_{fg}$, in which there are general family setup

times and costs. In this variant there is a selection process whereby some jobs are eligible for rejection, although this incurs a cost. By not allowing machine idle time, a representation of solutions as a sequence is adopted. The insert neighborhood is used, and a translation mechanism is applied to reject those eligible jobs that are completed after their deadlines.

6 PARALLEL-MACHINE PROBLEMS

In this section, we consider the application of local search to the scheduling of identical, uniform and unrelated parallel machines. As in Section 5, we review the main results and, where possible, give an indication of problem 'hardness' based on the performance of enumerative algorithms. Our discussion excludes the extensive research on worst-case analysis of approximation algorithms.

In our description of local search methods, it is convenient to picture a two-stage solution procedure: jobs are assigned to machines in the first stage, and each individual machine is scheduled in the second stage. For some problems, such as $P \| C_{\max}$, $Q \| C_{\max}$, and $R \| C_{\max}$, this second stage is trivial since job sequences do not affect the objective function. Other problems permit a simple polynomial algorithm for the second stage, so they can be viewed as pure assignment problems. Examples for unrelated parallel machines (which also hold for the corresponding problems with identical and uniform machines) include $R \| L_{\max}$, for which the EDD rule solves the single-machine problems, $R \| \sum w_j C_j$, for which the SWPT rule is used, and $R \| \sum U_j$, for which Moore's algorithm is used. For a third category, which includes $R \| \sum w_j T_j$, combined assignment and sequencing decisions are necessary.

For the pure assignment problems, the approaches given in Section 4.2 can be applied; Section 4.3 provides guidelines for combined sequencing and assignment problems. Our discussion is restricted to natural representations of solutions; we are not aware of any research on parallel-machine scheduling problems in which other representations are used.

6.1 Identical parallel machines

A generalization by Conway, Maxwell & Miller [1967] of the SPT rule allows the problem $P \| \sum C_j$ to be solved in $O(n \log n)$ time. For other optimality criteria, the situation is gloomy. Bruno, Coffman & Sethi [1974] and Lenstra, Rinnooy Kan & Brucker [1977] show that $P2 \| \sum w_j C_j$ and $P2 \| C_{\max}$ are NP-hard in the ordinary sense; this implies the NP-hardness of $P2 \| L_{\max}$ and $P2 \| \sum U_j$.

Finn & Horowitz [1979] propose a descent method for $P \| C_{\max}$ which requires $O(n \log m)$ time. Jobs are reassigned from a most heavily loaded to a least heavily loaded machine. A more sophisticated descent algorithm is suggested by França et al. [1994]. They use a combined critical reassign and critical swap neighborhood, as described in Section 4.2. A partitioning of jobs according to the similarity of their processing times is useful in selecting neighborhood moves. Hübscher & Glover [1994] propose a tabu search algorithm for $P \| C_{\max}$.

They replace the original maximum completion time objective function by $\sum_{i=1}^{m} (\bar{H}_i - \bar{H})^2$, where H_i is the total processing time assigned to machine i and $\bar{H} = \sum_{j=1}^{n} p_j/m$ represents the ideal machine load. A combined critical reassignment and critical swap neighborhood is used, where the job to be reassigned, or one of the jobs to be swapped, is moved from the most heavily loaded machine to a machine with load less than $\bar{H}$. After a job is moved from a machine, a tabu list entry prevents a job with the same processing time from moving to this machine. However, some tabu list entries are periodically activated and deactivated with the aim of diversifying and intensifying the search. A further special feature is the use of 'influential diversification', which is a device that modifies the current solution when no improvement in the best solution is observed for a long period. Here two machines are selected that have a surfeit and deficit of long jobs, and these long jobs are redistributed uniformly between this pair of machines. Computational results for problems with up to 50 machines and 2000 jobs indicate that the algorithm generates solutions with a makespan that is very close to $\bar{H}$, and that influential diversification is beneficial in detecting these solutions.

Among the various branch-and-bound algorithms for $P \| \sum w_j C_j$, the approach adopted by Belouadah & Potts [1994] in which lower bounds are obtained by a Lagrangean relaxation of machine capacity constraints appears best; problems with up to 20 jobs and 8 machines can be solved. Barnes & Laguna [1993] use the combined reassign and swap neighborhood of Section 4.2 in a tabu search algorithm for $P \| \sum w_j C_j$. By storing partial sums of processing times and weights on each machine in the current schedule, the total weighted completion time of a neighbor is evaluated in constant time. A tabu list of length 8 contains jobs which are forbidden to move from their current machine assignments; the job changing its machine assignment in a reassign move, or one of the jobs changing its assignment in a swap move, is stored at each iteration. In an attempt to diversify the search, swap moves are forbidden (except for the first few iterations) if they do not reduce the total weighted completion time. Computational results indicate that the algorithm consistently generates optimal solutions for problems with up to 4 machines and 30 jobs.

The problem $P|\bar{d}_j|\sum w_j E_j$ is equivalent to $P|r_j|\sum w_j C_j$ provided that the deadlines are sufficiently large. Laguna & González Velarde [1991] propose a tabu search algorithm for the former problem, although our description of their work refers to $P|r_j|\sum w_j C_j$. For this combined assignment and sequencing problem, they use a combination of the transpose, swap, and insert neighborhoods described in Section 4.3. To limit the size of the neighborhood, two jobs may be swapped or one job inserted after another job, only if the pair of jobs have similar release dates. The tabu list, which they suggest should have length $[\sqrt{n}] + 5$, contains jobs that are forbidden to move under a transpose, swap or insertion. After an insert move is executed, the corresponding job is placed on the tabu list, whereas after a transpose or swap move, the job experiencing the larger reduction in weighted completion time is stored. A special feature when no improving move is possible is that job j is forced to move, where j is chosen to have

the largest weighted completion time among jobs that are not on the tabu list. The authors suggest that, when the tabu search algorithm terminates, a branch-and-bound algorithm for $1|r_j|\sum w_jC_j$ can be applied to each of the m machines to optimally schedule the jobs currently assigned to those machines.

Hou, Ren & Ansari [1990] report some preliminary work on a genetic algorithm for $P|prec|C_{\max}$. They use a set of strings, one for each machine, to represent a solution. They also restrict solution strings by using a 'height' concept based on the maximum length chain of predecessors for a job. This is also used in the crossover mechanism to ensure that only feasible solutions are generated. A similar approach is used by Hou & Li [1991] for a related problem arising in the scheduling of flexible manufacturing systems in which the transport system between machines (using automated guided vehicles) must also be scheduled.

Motivated by the problem $P\|C_{\max}$, Hellstrom & Kanal [1992] propose a neural network approach which attempts to find a schedule in which all jobs are completed by a given threshold value. The output of a neuron determines whether or not a job is assigned to a particular machine.

6.2 Uniform parallel machines

A further refinement by Horowitz & Sahni [1976] to the generalized SPT algorithm for $P\|\sum C_j$ allows the problem $Q\|\sum C_j$ to be solved in $O(n \log n)$ time. The NP-hardness for other optimality criteria follows from the corresponding results for identical parallel machines.

Lo & Bavarian [1992] compare simulated annealing algorithms for $Q|\bar{d}_j|C_{\max}$. The deadline constraints are enforced using a penalty function approach. The problem is treated as one of combined assignment and sequencing, and the insert neighborhood is used.

6.3 Unrelated parallel machines

The problem $R\|\sum C_j$ is formulated by Horn [1973] and Bruno, Coffman & Sethi [1974] as a weighted bipartite matching problem; hence it is solvable in $O(n^3)$ time. As for uniform machines, the NP-hardness results given in Section 6.1 for identical machines imply NP-hardness for the corresponding unrelated machine problems.

Using a lower bound based on a surrogate relaxation, the branch-and-bound algorithm of Van de Velde [1993] solves instances of $R\|C_{\max}$ with up to 200 jobs and four machines. Hariri & Potts [1991] propose a descent algorithm for $R\|C_{\max}$ in which both the critical reassign and critical swap neighborhoods of Section 4.2 are used. With a suitable implementation, the objective function value of a neighbor can be computed in constant time. Computational results of Hariri & Potts indicate that this descent algorithm generates better-quality solutions than various two-phase heuristics [Potts, 1985b; Lenstra, Shmoys & Tardos, 1990], which use linear programming in their first phase to schedule most of the jobs. Hariri & Potts also report on initial experiments which indicate

that a more complicated neighborhood structure has little effect on solution quality. Glass, Potts & Shade [1994] propose simulated annealing and tabu search algorithms based on the critical reassign and critical swap neighborhoods. They also describe a genetic algorithm which uses the natural recombination operators suggested in Section 4.2, and a corresponding genetic descent algorithm in which the descent algorithm of Hariri & Potts is applied to each solution in every population. Computational results show that the standard genetic algorithm performs poorly relative to the other three algorithms which are roughly comparable. These results are presented in Section 8.2.

Kanet & Sridharan [1991] consider the problem $R|r_j, t_{jk}|\sum f_j(C_j)$ in which the objective function is an arbitrary convex function of job completion times. They report preliminary work using a start time representation in a genetic algorithm with the recombination operator which is described in Section 4.3.

7 MULTI-STAGE PROBLEMS

In this section we consider the application of local search to flow shop, open shop, and job shop problems. As is the case in most studies, the main focus of our discussion is on problems with the maximum completion time objective. Many local search algorithms for $F\|C_{\max}$, $O\|C_{\max}$ and $J\|C_{\max}$ are easily adapted to other optimality criteria. We follow the same format as in the two previous sections by reviewing the complexity results and commenting on problem 'hardness' by referring to the performance of the best currently available branch-and-bound algorithms.

Permutation flow shops in which each machine processes the jobs in the same order are sequencing problems, so Section 4.1 provides guidelines for the design of local search methods. More generally, flow shops, open shops, and job shops are multisequencing problems to which the approaches of Section 4.4 can be applied.

7.1 Flow shops

For $F\|C_{\max}$ Conway, Maxwell & Miller [1967] observe there exists an optimal schedule with the same processing order of jobs on the first pair of machines and the same order on the last pair of machines. Thus, for $F2\|C_{\max}$ and $F3\|C_{\max}$, it is sufficient to consider permutation schedules in which each machine processes the jobs in the same order. Johnson [1954] gives an $O(n \log n)$ algorithm for $F2\|C_{\max}$: the jobs with $p_{1j} \leqslant p_{2j}$ are sequenced first in nondecreasing order of p_{1j}; the remaining jobs are then sequenced in nonincreasing order of p_{2j}. Garey, Johnson & Sethi [1976] show that $F3\|C_{\max}$ is strongly NP-hard. Although permutation schedules are not guaranteed to provide optimal solutions for $Fm\|C_{\max}$ when $m \geqslant 4$, we follow a tradition in the literature and henceforth concentrate on the permutation flow shop, i.e., finding the best permutation schedule.

The best available branch-and-bound algorithms are those of Lageweg, Lenstra & Rinnooy Kan [1978] and Potts [1980], which compute lower bounds by solving two-machine subproblems using Johnson's algorithm. Their performance is not entirely satisfactory, however: they experience difficulty in solving instances with 15 jobs and 4 machines. Strong NP-hardness for $F2 \| L_{\max}$ and $F2 \| \sum C_j$ is established by Lenstra, Rinnooy Kan & Brucker [1977] and Garey, Johnson & Sethi, respectively, and NP-hardness in the ordinary sense for $F2|s_f|C_{\max}$ is established by Kleinau [1993] for the case of an arbitrary number of families F.

For heuristics that do not employ neighborhood search, the $O(mn^2)$ insertion method of Nawaz, Enscore & Ham [1983] is best for the permutation flow shop $F \| C_{\max}$; Taillard [1990] gives a derivation of its complexity. The insertion method builds a sequence by adding a job in the best position to the current partial sequence until it produces a complete sequence. Rather surprisingly, it outperforms the descent algorithm of Dannenbring [1977], which is based on the transpose neighborhood.

Various local search methods are available for the permutation flow shop $F \| C_{\max}$. They each use a natural representation of solutions as a sequence. Simulated annealing algorithms are proposed by Osman & Potts [1989] and by Ogbu & Smith [1990]. Both approaches compare the insert and swap neighborhoods; the insert neighborhood performs better in computational tests. Ogbu & Smith attribute this to the comparatively large size of the neighborhood, whereas Osman & Potts suggest it may depend upon the objective function. In the algorithm of Ogbu & Smith, the probability of accepting a worse solution is independent of the objective function values, whereas Osman & Potts use a standard acceptance probability. An evaluation of these two approaches by Ogbu & Smith [1991] finds they give similar results, although the algorithm of Osman & Potts is marginally more effective.

Tabu search algorithms are proposed by Widmer & Hertz [1989], Taillard [1990], Reeves [1993b], and Nowicki & Smutnicki [1996b]. Widmer & Hertz use the swap neighborhood, adopt a tabu list of length 7 that prevents either of the interchanged jobs from returning to its previous position, and search the complete neighborhood before making a move to the best neighbor (a best improve acceptance strategy). Taillard suggests an improvement to each of the key components in the method of Widmer & Hertz. Based on computational tests, he claims that the insert neighborhood is better than swap, that a tabu list containing values of $C_{\max}$ is superior to one storing forbidden positions for jobs, and that a first improve acceptance strategy making the first move which reduces the maximum completion time is slightly better than the best improve acceptance strategy. Taillard also derives a method for evaluating all insert neighbors in $O(mn^2)$ time (although to evaluate any one of the $(n-1)^2$ neighbors requires $O(mn)$ time). Reeves proposes the use of a restricted version of the insert neighborhood in which a specific subset of jobs is eligible for insertion at each iteration. The best computational results are obtained for subsets containing six jobs that are created by a random partition of the original set of jobs. After all

subsets are considered, new subsets are created by random partitioning. The algorithm has a tabu list of length 7 that prevents the inserted job from returning to its previous position, and uses a best improve acceptance strategy on the restricted neighborhood.

Nowicki & Smutnicki's [1996b] tabu search algorithm uses a restricted version of the insert neighborhood, where the block structure of some critical path (as defined in Section 4.4) determines the eligible neighbors. A job without an operation that begins or ends a block may be inserted in the position corresponding to the end of its block and in the following few positions, or in the position corresponding to the start of its block and in the previous few positions. Alternatively, the eligible positions for insertion of a job which has an operation that ends one block and an operation that starts another block are as follows: the end of the block that it starts and the following few positions, and the start of the block that it ends and the previous few positions. A tabu list of length 8 contains precedence constraints, which are created as follows. For two adjacent jobs in the original sequence, a tabu list entry defines the second job to be a predecessor of the first if either the second of these jobs is inserted in an earlier position or the first job is inserted in a later position. Using ideas from the analysis of Taillard [1990], together with the observation that the block structure limits the number of positions to which a job is allowed to move, Nowicki & Smutnicki establish that all neighbors can be evaluated in $O(m^2n)$ time. Another special feature of their algorithm is a backtracking procedure: when an iteration limit is reached, the procedure is restarted from the best solution, but with a different neighborhood move to that made previously.

Computational results of Reeves [1993b] indicate that his tabu search algorithm is superior to the simulated annealing method of Osman & Potts and also to the corresponding tabu search algorithm which uses the full insert neighborhood. The computational results of Nowicki & Smutnicki [1996b] show that their algorithm generates better-quality solutions than those of Taillard or Werner [1993] (see below for a description of Werner's path algorithms) and that it requires much less computation time.

Werner [1993] proposes a class of 'path' algorithms for the permutation flow shop $F \| C_{\max}$. Each algorithm can be viewed as descent with exploration and backtracking. A restricted version of the insert neighborhood is used: a neighbor is not considered if it allows the possibility that all critical operations in the current solution are to remain critical and consequently cannot improve upon the objective function value. A sequence of restricted insert moves is performed (they are accepted irrespective of objective function values). If a solution is found which is better than all of those that are previously generated, a new sequence of restricted insert moves is initiated from this solution. On the other hand, after generating a prespecified number of moves, if no improved solution is found, the algorithm backtracks to the best solution and another sequence of restricted insert moves is made. The procedure for selecting a neighbor is quite complicated, and is based on quickly computed lower bounds for the maximum completion time. Stochastic versions of the algorithm perform a move to the

best of a random selection of restricted insert neighbors, and backtracking may occur when a probabilistic condition is satisfied, rather than after a prespecified number of moves. Upon termination of this procedure, a descent algorithm based on the block transpose neighborhood is applied. Computational results indicate that a stochastic version of this algorithm yields the best-quality solutions, and that this algorithm is superior to the simulated annealing and tabu search methods of Osman & Potts and Widmer & Hertz.

Reeves [1995] proposes a genetic algorithm which uses the reorder crossover. In comparative computational results which have the same computation time limit for each algorithm, the genetic algorithm produces better-quality solutions than multistart descent and is comparable with the simulated annealing method of Osman & Potts.

Several of the algorithms for permutation flow shop $F \| C_{\max}$ are tested on the problems generated by Taillard [1993b], thus enabling some additional comparisons to be made. From the various sets of computational results, we conclude that the tabu search algorithms of Reeves and Nowicki & Smutnicki generate the best quality solutions: they appear to outperform the simulated annealing algorithm of Osman & Potts, Reeves' genetic algorithm, and Werner's path algorithms. The current champion is the method of Nowicki & Smutnicki, which exhibits superiority over Reeves' tabu search algorithm.

We now discuss research which assumes different optimality criteria. Kohler & Steiglitz [1975] compare different neighborhoods in descent algorithms for $F2 \| \sum C_j$, and Krone & Steiglitz [1974] propose a descent algorithm for $F \| \sum C_j$ in which permutation schedules are not assumed. Based on the insert neighborhood, the algorithm of Krone & Steiglitz searches for the best permutation schedule, and then attempts insertions on the individual sequences defining the processing orders on the different machines. For the permutation flow shop $F \| \sum w_j C_j$, Glass & Potts [1996] perform a computational comparison of multistart descent, simulated annealing, threshold accepting, tabu search, and two genetic algorithms, one of which applies descent to each solution in every population. The neighborhood search algorithms each use the swap neighborhood, which performs marginally better than insert in initial experiments. Also, based on further initial tests, the insertion crossover is used in the first genetic algorithm, whereas the genetic descent algorithm employs the reorder crossover. Simulated annealing and the genetic algorithm that incorporates descent generate the best-quality solutions, and the genetic descent algorithm is slightly superior. These results are presented in Section 8.3.

In independent studies by Kim [1993] and Adenso-Dias [1992], the tabu search algorithm of Widmer & Hertz is adapted to the permutation flow shop problems $F \| \sum T_j$ and $F \| \sum w_j T_j$. In the algorithm of Adenso-Dias, the swap neighborhood is used in the first part of the search; thereafter, insert neighbors only are considered. To save on computation time, the complete neighborhood is not searched: a swap is allowed only if the positions of the two jobs are closer than a threshold value, and a job may only be reinserted in a position that is closer than a threshold value to its previous location in the sequence. The

threshold value decreases as the search progresses from an initial value of $n-1$ to a final value of 4. Computational results indicate that restricting the neighborhood reduces computation time, without affecting average solution quality. Cleveland & Smith [1989] compare genetic algorithms for a variant of the problem $F \| \sum (E_j + T_j^2)$ in which there are identical parallel machines at each stage. The problem is regarded as one of sequencing jobs at the first stage: jobs are processed in order of arrival at subsequent stages.

The permutation flow shop problem $F|s_f|C_{\max}$ is the subject of two studies. Vakharia & Chang [1990] represent solutions as sequences and propose a simulated annealing algorithm that uses a neighborhood based on the transpose of adjacent batches and of adjacent jobs. A genetic algorithm of Whitley, Starkweather & Shaner [1990] uses a recombination operator, called edge recombination, which is designed for use with the traveling salesman problem. They perform computational tests on problems with six machines in which all jobs from the same family are identical.

Various conclusions can be drawn from some of the studies on neighborhood search for the permutation flow shop problem. First, insert is the best neighborhood when minimizing the maximum completion time. Second, it is often advantageous to restrict the search to a subset of the neighbors. Moreover, the critical operations provide a useful guide as to which neighbors should be avoided. Third, an efficient computation of the exact or estimated objective function value for a neighbor has a substantial effect on the efficiency of the neighborhood search algorithm.

7.2 Open shops

Gonzalez & Sahni [1976] derive an elegant $O(n)$ algorithm for $O2 \| C_{\max}$, and show that $O3 \| C_{\max}$ is NP-hard in the ordinary sense. We are not aware of any literature on enumerative algorithms for $O \| C_{\max}$. Strong NP-hardness of $O2 \| L_{\max}$ and $O2 \| \sum C_j$ is established by Lawler, Lenstra & Rinnooy Kan [1981, 1982] and Achugbue & Chin [1982], respectively.

A problem that is related to the open shop occurs when scheduling jobs for which a number of process plans are available. The process plan for a job specifies which machines may carry out each operation and the precedence relations that exist between operations. Detailed process plans may also include choices on the tools to be used for particular operations (which will affect setup times). An open shop thus corresponds to a problem in which there are no precedence relationships between operations and in which each operation can only be performed by one machine. Bagchi et al. [1991] describe a genetic algorithm for the scheduling problem with process plans in which the partially matched crossover is used to handle the priority order between jobs and a type of uniform crossover is used for the machine and process plan selection. Husbands, Mill & Warrington [1991] discuss a genetic algorithm in which there are separate populations of process plans for each part to be made, and also a population of priority orders that define

a schedule. Each population evolves at the same time, and the individual process plans are evaluated by simulating their effect when run in conjunction with other process plans and a particular priority order.

7.3 Job shops

The problem $J2 \| C_{\max}$, and consequently $J2 \| L_{\max}$, is shown by Garey, Johnson & Sethi [1976] to be strongly NP-hard. Strong NP-hardness of $J2 \| \sum C_j$ is implied by the corresponding result for the two-machine flow shop. Various enumerative and heuristic methods for $J \| C_{\max}$ employ the following *disjunctive graph* formulation. For each operation O_{ij}, there is a vertex with weight p_{ij}. The precedence between each pair of consecutive operations on the same job is represented by a directed arc. And there is an undirected edge corresponding to each pair of operations that require the same machine. Choosing a processing order on every machine corresponds to orienting the edges to produce a directed acyclic graph. It is therefore required to find an orientation that minimizes the length of a longest or critical path, where the length is defined as the sum of weights of vertices that lie on the path.

The job shop problem $J \| C_{\max}$ is regarded as one of the hardest in combinatorial optimization. For example, a classic 10-job 10-machine instance that is originally given by Fisher & Thompson [1963] has only fairly recently been solved to optimality by Carlier & Pinson [1989], Applegate & Cook [1991], and Brucker, Jurisch & Sievers [1994]. Within their algorithms these authors each consider single-machine subproblems which are obtained by relaxing all edges in the disjunctive graph that are not yet oriented, except those corresponding to operations on some selected machine. The resulting problem is equivalent to $1|r_j|L_{\max}$, where all due dates are nonpositive. For each operation on the selected machine in this single-machine subproblem, its release date is the length of a longest path to any predecessor of this operation, and its due date is minus the length of a longest path from any sucessor.

Many heuristics are based on the use of priority rules, which are surveyed by Haupt [1989]. Such approaches use a priority rule to select an operation from a set of candidates to be sequenced next. The candidates may be chosen to create a nondelay schedule in which no machine idle time is allowed if operations are available to be processed. There is no guarantee of an optimal solution that is a nondelay schedule, so it may be preferable to generate an active schedule: Giffler & Thompson [1960] propose a procedure for active schedule generation. A limited delay schedule offers a useful compromise between nondelay and active schedules. Although priority rule heuristics are undemanding in their computational requirements, the quality of schedules that are generated tends to be erratic.

An effective heuristic method is the *shifting bottleneck* procedure of Adams, Balas & Zawack [1988]. It is based on the observation that a schedule can be constructed by selecting each machine in turn and orienting all of the corresponding edges in the disjunctive graph formulation. The problem of orienting these

edges is equivalent to solving $1|r_j|L_{\max}$, where previously oriented edges are used in the computation of release dates and due dates: the branch-and-bound algorithm of Carlier [1982] computes a solution. The unscheduled (bottleneck) machine is selected so that the maximum lateness for the corresponding problem $1|r_j|L_{\max}$ is as large as possible. After the scheduling of a bottleneck machine by orienting the corresponding edges, the machine reschedule neighborhood of Section 4.4 is searched in an attempt to find an improved partial schedule (where each neighbor requires the solution of $1|r_j|L_{\max}$ by Carlier's algorithm). When all edges are oriented, a final step of the procedure searches for a further improvement by considering one k-machine reschedule neighbor, where $k \leqslant \sqrt{m}$, for which a sequence of k instances of $1|r_j|L_{\max}$ are solved.

Adams, Balas & Zawack also propose an enumerative version of their procedure in which several alternatives are considered for the machine that is to be scheduled next. Various improvements to the original shifting bottleneck procedure have been suggested. Applegate & Cook [1991] propose variants (Bottle-k for $k = 4, 5, 6$) in which, for the last k machines, every possibility is considered when selecting the machine to be scheduled next. In another procedure (Shuffle), they use their branch-and-bound procedure to generate a k-machine reschedule neighbor of the schedule obtained using Bottle-5, where k is quite large (for example, $k = 15$ for $m = 20$).

Dauzère-Pérès & Lasserre [1993] observe that the current orientation of some edges in the shifting bottleneck procedure may create a path between two operations that require the same machine. In this case the minimum time delay between the start times of these operations can be computed. Thus, the $1|r_j|L_{\max}$ problems that are considered within the shifting bottleneck procedure should ideally incorporate *delayed precedence constraints* to account for these delays. Dauzère-Pérès & Lasserre use a heuristic approach for these single-machine problems with delayed precedence constraints, whereas Balas, Lenstra & Vazacopoulos [1995] obtain an exact solution by designing a generalized version of Carlier's algorithm. The quality of the solution obtained from the basic shifting bottleneck procedure of Adams, Balas & Zawack is relatively poor, although its computational requirements are modest. Its variants each exhibit an improved performance at the expense of a greater investment in computation time.

Various local search algorithms for $J \| C_{\max}$ have been proposed, and they are reviewed by Vaessens, Aarts & Lenstra [1996]. The neighborhood search approaches use a natural representation, but with an exclusion mechanism of the type outlined in Section 4.4.

Simulated annealing algorithms are the subject of three studies for $J \| C_{\max}$. First, Van Laarhoven, Aarts & Lenstra [1992] suggest the use of the critical transpose neighborhood. On the other hand, Matsuo, Suh & Sullivan [1988] adopt the smaller critical end transpose neighborhood. However, they observe that complex precedence relations between operations propagate the effect of a local change throughout the entire job shop system, and hence a neighborhood move may lead to deterioration of the objective function value, even when a superior solution can be obtained by a small number of further moves. Thus, if

a neighbor yields an inferior solution, further transposes are attempted involving a predecessor or sucessor of one of the originally transposed operations. Yamada, Rosen & Nakano [1994] use the critical end insert neighborhood. Their algorithm backtracks to the best schedule that is currently generated whenever 3000 moves are accepted that do not improve the best solution value, and resets the temperature appropriately. Computational results in these three studies show that each of the simulated annealing algorithms generates better-quality solutions than those of the shifting bottleneck procedure and its variants. The method of Yamada, Rosen & Nakano finds better solutions than the other two algorithms, although at considerable computational expense. The best compromise between solution quality and computation time is offered by the algorithm of Matsuo, Suh & Sullivan.

Tabu search provides an attractive alternative to simulated annealing for $J \| C_{\max}$. We describe four algorithms. Taillard [1994] adopts the critical transpose neighborhood that is used by Van Laarhoven, Aarts & Lenstra [1992]. Following the transpose of two operations, the tabu list forces these operations to be sequenced in adjacent positions. Special features of the algorithm include the replacement of exact evaluations of the maximum completion time of each neighbor by quickly computed lower bound estimates, a tabu list length that changes randomly after specified numbers of iterations are performed, and the use of a penalty function which aims to prevent the repeated transpose of an operation to earlier positions in the schedule.

Barnes & Chambers [1995] use the framework of Taillard's method to design an alternative tabu search algorithm. We highlight the special features of their approach. To enable the execution of a backtracking procedure, a schedule is stored if it has a better objective function value than any previously generated solution. When the algorithm fails to detect an improved solution after a specified number of iterations, it restarts using the best of the stored schedules as an initial solution. Several runs are performed from this initial schedule, each with a different tabu list length. Following these runs with a given selection of tabu list lengths, the corresponding initial schedule is removed from the collection of stored solutions, and further exploration is initiated from the best schedule that is currently stored.

Dell'Amico & Trubian [1993] use a composite neighborhood consisting of generalized critical end transpose – which allows the reordering of three critical operations (one of which must start or end a block) – and critical end insert. Their algorithm adopts the idea used by Taillard of selecting moves according to quickly computed lower bounds for the maximum completion time, rather than performing a full objective function evaluation of each neighbor. They also use a variable-length tabu list which prevents the reorientation of disjunctive arcs that are reversed in the previous moves.

Lastly, Nowicki & Smutnicki [1996a] use the critical end transpose neighborhood. Their tabu list also prevents the reversal of the most recent transpose moves. However, when there is no allowable move, the tabu list is updated by discarding the oldest entry and adding a duplicate of the most recent entry. As in

their tabu search algorithm for the permutation flow shop problem, Nowicki & Smutnicki incorporate a backtracking procedure: when an iteration limit is reached, the procedure is restarted from one of the best solutions, but with a different neighborhood move to that made previously. On the evidence of the computational results quoted by the different authors, the four tabu search algorithms are generally superior to simulated annealing and to variants of the shifting bottleneck procedure. The method of Nowicki & Smutnicki is clearly superior to the other three methods in terms of both solution quality and computation time.

Balas & Vazacopoulos [1998] propose a *guided local search* procedure, which resembles tabu search with backtracking. The critical end insert neighborhood is used to construct a sequence of 'neighborhood trees', where a feasible solution is associated with each node of a tree. For any node, each of its immediate descendants is a neighbor of the corresponding solution. Using lower bound estimates of the maximum completion time for each neighbor, a given number of nodes are created for the solutions with the lowest estimates. For a node that is created by inserting an operation before or after another operation, a precedence constraint is added to prevent the order of this pair of operations being reversed, and the immediate descendants that are created subsequently each have a precedence constraint which forces this pair to be sequenced in the same order as for the parent node. In addition to the restriction on the number of immediate descendants for each node, a limit is placed on the number of levels that are explored. After the creation of a neighborhood tree, one of the best solutions that is generated from a previous tree becomes the root node of a new tree.

Balas & Vazacopoulos also suggest several hybrids in which guided local search is embedded in the shifting bottleneck procedure. In the basic hybrid, instead of searching the machine reschedule neighborhood after a bottleneck machine is scheduled, a given number of neighborhood trees are generated using the current (partial) solution from the shifting bottleneck procedure at the root of the first tree. The best schedule from these neighborhood trees is then used for the continuation of the shifting bottleneck procedure. The other hybrids extend the basic version by allowing additional search. This further search is performed by removing the orientation of edges on certain machines, then constructing a new schedule using a combination of guided local search and shifting bottleneck routines. Computational results show that the guided local search procedure is superior to simulated annealing and variants of the shifting bottleneck procedure. The hybrid shifting bottleneck/guided local search methods produce high-quality solutions at reasonable computational expense, and are preferred to pure guided local search. These hybrids also compare favorably with most of the tabu search algorithms, although it is not clear whether they are superior to the method of Nowicki & Smutnicki [1996a].

Genetic algorithms which use a variety of different representations have been proposed for $J \| C_{max}$. Several of these algorithms use a heuristic-based representation: a solution is constructed from its representation by applying some

heuristic method. Storer, Wu & Vaccari [1992] propose the data perturbation representation described in Section 3.1. A schedule is constructed from the perturbed data by using SPT as a priority rule to construct a limited delay schedule in which a small amount of unforced machine idle time is allowed. They also suggest a heuristic set representation: after partitioning the time horizon into time windows, each solution is represented by a string, where the entry in a particular position is the priority rule (chosen from a group of 6) used within the corresponding time window. Limited delay scheduling converts the representation into a schedule. Storer, Wu & Vaccari also consider a hybrid algorithm which uses a representation based on data perturbations and heuristic sets.

Dorndorf & Pesch [1995] implement a heuristic set representation in which a string entry is the priority rule (chosen from a group of 12) that is used in the corresponding iteration when creating an active schedule. A similar approach is used by Smith [1992]. Della Croce, Tadei & Volta [1995] propose the use of priority representation, an approach suggested previously by Davis [1985] and Falkenauer & Bouffouix [1991]. In this representation, each machine has a sequence that defines its priority order, and a type of insertion crossover is used in which the inserted subsection of a string is put into the same position that it previously occupied in the original string. Another limited delay scheduling procedure is used for generating a schedule from the representation. The algorithm performs better if the scheduling procedure is treated as a repair mechanism, so the priority representation for each machine is replaced by the corresponding processing order of operations in the schedule that is generated. A similar approach is used by Yamada & Nakano [1992] who use completion times to define priorities. For recombination, a set of candidate operations in an active schedule is found using the procedure of Giffler & Thompson [1960], and a random choice is made between the two candidates with the smallest completion time in each parent schedule. After a complete schedule is generated, this procedure is repeated with the same two parents. Dorndorf & Pesch design a genetic algorithm based on the shifting bottleneck procedure. More precisely, using a representation that is a sequence of machines, a solution is constructed by using the representation to define the order in which the machines are scheduled within the shifting bottleneck procedure.

Three further genetic algorithms are proposed by Pesch [1993]. The first is based on the observation that a schedule can be constructed by selecting job pairs in turn, solving the corresponding two-job subproblem, and using the solution to orient the edges between operations on these jobs in the disjunctive graph formulation. The genetic algorithm is used to select the order in which the job pairs are considered. The second genetic algorithm of Pesch uses a representation that gives an (artificial) upper bound on each two-job subproblem. For each subproblem, the upper bound may allow some edges to be oriented on the basis of consistency tests. A complete schedule is constructed using a similar approach to that in the algorithm of Yamada & Nakano: the Giffler–Thompson procedure selects candidate operations, and a selection between candidates is

based on the two parent schedules. The third genetic algorithm of Pesch is similar to the second, but uses single-machine subproblems instead of two-job subproblems.

There are also genetic algorithms that do not rely on a heuristic-based representation. Nakano & Yamada [1991] use a type of ordered pair representation (see Section 4.1). They employ a standard crossover operation on the resulting binary strings (with the information corresponding to a single job pair for different machines making up a single string). A comparison of approaches based on translation and on repair (which they call 'forcing') shows repair to be more effective. Aarts et al. [1994] perform a computational comparison of several local search methods. Included in their comparison is a hybrid which incorporates descent into a genetic algorithm. Recombination in this genetic descent algorithm is achieved by choosing one of the two parents and then finding two operations that are eligible for transpose under the critical transpose neighborhood and that have the reverse order in the second parent. These jobs are then reversed in the first parent, and the whole process is repeated several times. A descent algorithm, which may use either the critical transpose or the extended version of the critical end transpose neighborhood, is applied to each newly generated solution. The performance of the genetic algorithms is variable. The most effective appears to be the third algorithm of Pesch [1993], which uses upper bounds on single-machine subproblems to orient edges. Although comparing favorably with simulated annealing, it cannot compete with the best tabu search method or the hybrid shifting bottleneck/guided local search algorithms.

Several of the algorithms for the job shop $J \| C_{\max}$ are tested on the problems generated by Fisher & Thompson [1963], Lawrence [1984], and Taillard [1993b]. Vaessens, Aarts & Lenstra [1996] collate the objective function values generated by the various algorithms and provide standardized computation times. These results are extended by Balas & Vazacopoulos [1998]. The candidates for champion algorithm are the tabu search algorithm of Nowicki & Smutnicki [1996a] and the hybrid shifting bottleneck/guided local search algorithms of Balas & Vazacopoulos. Section 8.4 presents the main findings of these computational studies that compare different approaches.

There are several neural network studies for the problem $J \| \sum C_j$. Foo & Takefuji [1988a, 1988b] propose a model in which the output of neurons defines precedences between operations. On the other hand, Foo & Takefuji [1988c] and Zhou et al. [1991] use a formulation in which the output of a neuron defines the starting time of the corresponding operation.

Some scheduling problems in flexible manufacturing systems have a job shop structure. Widmer [1991] proposes a tabu search algorithm for a variant of the problem $J \| C_{\max} + \sum w_j U_j$. Associated with each operation on every job is a requirement for certain tools to be loaded in a tool magazine that has limited capacity. The weighted number of reloadings of the tool magazine forms an additional component in the objective function. The algorithm uses a natural representation of solutions and employs the insert neighborhood.

In common with the permutation flow shop, there emerge some performance-enhancing features of neighborhood search algorithms for $J \| C_{max}$. First, it is beneficial to restrict the neighborhood so that a critical operation is transposed, swapped, or inserted. Second, to use computation time efficiently, it may be preferable to estimate objective function values rather than perform an exact evaluation. Third, the possibility of backtracking to a previous solution, rather than always continuing the search from the current solution, allows the search to be redirected to parts of the solution space with a higher chance of finding a good-quality solution.

8 COMPUTATIONAL COMPARISONS

There are obvious similarities between different local search methods; even a genetic algorithm, which contains some unique features, will often be implemented with a pure neighborhood search component. Due to the scarcity of comparative computational studies for scheduling problems in the literature, it is not clear which of the various methods is the best. In this section we briefly summarize four computational studies in which different local search methods are compared against each other. These studies cover most of the different problem types that are introduced in Section 4.

When evaluating local search algorithms, there is clearly a trade-off between the investment in computation time and the quality of solution. To provide a fair comparison, the same computation time should be allocated to different methods; otherwise it is difficult to draw firm conclusions. It is usual to measure the effectiveness of a local search method by the relative (percentage) deviation of the value of the solution that is generated from the best known solution value (or from the best known lower bound if such a bound is sufficiently tight). These deviations are often averaged over several runs of the algorithm on the same problem or on different problems. Furthermore, maximum deviations are often used by themselves or in conjunction with averages to give an indication of the consistency of the heuristic in finding a near-optimal solution.

8.1 Merging ordered lists of jobs in $1|s_f|\sum w_jC_j$

Crauwels, Potts & Van Wassenhove [1997] describe a computational study for the problem $1|s_f|\sum w_jC_j$ in which F families of jobs with associated sequence-independent setup times are to be scheduled on a single machine to minimize the total weighted completion time. Recall that a solution procedure requires lists of jobs, each containing the jobs of a family in SWPT order, to be merged. The methods on which the computational comparison is performed are multistart descent (MD), simulated annealing (SA), threshold accepting (TA), tabu search (TS), and a genetic algorithm (GA), details of which are given below.

- *MD.* The multistart descent algorithm represents solutions as sequences and uses a type of block insert neighborhood, as proposed by Ahn & Hyun [1990],

which guarantees to maintain the jobs within each family in SWPT order. There are $\lfloor n/3 \rfloor$ restarts.

- *SA*. The simulated annealing method uses the same representation and neighborhood as in MD. The 'cooling' scheme follows a periodic pattern in which temperatures decrease and increase. Also, descent iterations are performed between each of these temperature changes. The method is applied to four different initial solutions, and the best solution is selected.
- *TA*. The threshold accepting method also uses the same representation and neighborhood as in MD. The method is implemented in a similar way to SA with periodic threshold values, and with descent iterations performed between changes of threshold values. The best solution obtained from four different starting solutions is chosen.
- *TS*. Tabu search is implemented with the same representation as in MD, but uses a subset of the shift neighborhood in which the shifted job can only be inserted between batches and the SWPT order within families is preserved. A tabu list of length 7 stores jobs that are forbidden to move. The best solution obtained from $\lfloor n/3 \rfloor$ different starting solutions is selected.
- *GA*. The genetic algorithm, proposed by Mason [1992], uses the batch-based representation of solutions described in Section 4.5. A further sophistication is introduced by noting that an optimal solution will contain no very small batches (in which the total processing is not sufficiently large relative to the setup). Whenever a new batch is started, a minimum batch size can be calculated and used to restrict the solutions that are considered. The population size is $2n$. The best solution from two independent runs is selected.

For different combinations of n and F, Crauwels, Potts & Van Wassenhove [1997] randomly generate 50 test problems. Computational results for problems in which setup times have the same distribution as processing times are summarized in Table 11.1. In particular, the number of problems (out of 50) for which an optimal solution is found (NO), the maximum relative deviation from an optimal solution value (MRD) expressed as a percentage, and the average computation time required in seconds (ACT) on an HP 9000/825 are listed for each algorithm.

It is seen from Table 11.1 that each of the algorithms MD, SA, TA, TS, and GA generates solutions of high quality. The best results, as measured by numbers of optimal solutions generated and maximum relative percentage deviations, are obtained using TS. A further advantage of TS is that it requires less computation time than the other methods; MD, SA, TA, and GA have similar performance.

8.2 Assignment of jobs in $R \| C_{\max}$

Glass, Potts & Shade [1994] give a computational comparison of local search algorithms for the problem $R \| C_{\max}$ of scheduling jobs on unrelated parallel machines to minimize the maximum completion time. A schedule is specified by

Table 11.1 Summary of results for $1|s_f|\sum w_j C_j$

		MD			SA			TA			TS			GA		
n	F	NO	MRD	ACT	NO	MRD	ACT	NO	MRD	ACT	NO	MRD	ACT	NO	MRD	ACT
30																
	4	49	0.35	0.7	50	0.00	1.0	47	0.35	1.1	50	0.00	0.3	46	0.47	1.3
	6	48	0.09	0.8	49	0.13	1.1	50	0.00	1.2	50	0.00	0.5	49	0.18	1.2
	8	49	0.25	0.9	50	0.00	1.2	48	0.25	1.3	49	0.25	0.8	50	0.00	1.2
	10	46	0.07	1.0	48	0.05	1.2	48	0.02	1.3	50	0.00	0.9	50	0.00	1.2
40																
	4	47	0.05	1.9	48	0.04	2.5	47	0.13	2.9	48	0.05	0.7	45	0.11	2.9
	6	49	0.02	2.2	49	0.32	2.8	49	0.03	3.1	48	0.05	1.1	48	0.17	3.0
	8	48	0.11	2.7	50	0.00	3.2	46	0.08	3.3	50	0.00	1.7	49	0.01	2.9
	10	46	0.09	2.5	48	0.14	2.8	45	0.07	3.0	49	0.00	2.1	49	0.03	2.6
50																
	4	46	0.26	3.9	48	0.04	4.8	47	0.03	5.5	50	0.00	1.2	40	1.42	5.7
	6	47	0.03	4.9	45	0.10	5.7	46	0.07	6.2	50	0.00	2.1	44	0.17	5.9
	8	43	0.08	5.5	46	0.21	6.2	45	0.03	6.3	47	0.05	3.1	46	0.08	5.5
	10	46	0.10	6.1	45	0.09	6.4	43	0.09	6.4	50	0.00	4.2	50	0.00	5.6
Average		47	0.13	2.8	48	0.09	3.2	47	0.10	3.5	49	0.03	1.6	47	0.22	3.3

NO = Number of problems (out of 50) for which an optimal solution is found.
MRD = Maximum relative deviation (%) of heuristic solution value from that of the optimum.
ACT = Average computation time in seconds on an HP 9000/825.

Table 11.2 Average relative deviations (%) for $R \| C_{max}$

			20 seconds				100 seconds		
m	n	DA	SA	TS	GA	GD	SA	TS	GD
5									
	20	3.1	0.1	0.8	5.1	0.0	0.0	0.8	0.0
	50	3.0	0.7	0.6	6.3	0.5	0.2	0.5	0.1
	100	1.3	0.8	0.5	4.4	0.7	0.6	0.4	0.2
10									
	20	6.6	0.1	0.9	15.0	0.4	0.0	0.9	0.1
	50	4.3	1.4	0.5	12.5	0.5	0.6	0.5	0.1
	100	2.7	1.5	0.2	10.3	1.1	0.5	0.1	0.7
20									
	20	5.5	0.1	0.4	12.7	0.4	0.0	0.4	0.2
	50	7.8	0.7	0.4	21.3	1.7	0.4	0.4	0.9
	100	3.4	2.7	0.3	20.7	2.2	0.8	0.2	1.3
50									
	20	5.8	0.0	0.5	16.2	0.0	0.0	0.5	0.0
	50	5.8	0.1	0.3	19.3	0.3	0.0	0.3	0.1
	100	5.7	0.4	0.4	30.2	1.8	0.2	0.3	1.2
Average		4.6	0.7	0.5	14.5	0.8	0.3	0.4	0.4

an assignment of jobs to machines. The study compares the following five algorithms:

- *DA*. The descent algorithm adopts a natural representation of solutions. The combined critical reassign and critical swap neighborhood described in Section 4.2 is used, and the algorithm terminates when a local minimum is detected.
- *SA*. The simulated annealing method uses the same representation and neighborhood as in DA. Another feature is the cooling schedule of Lundy & Mees [1986], which outperforms geometric cooling in initial tests.
- *TS*. Tabu search also uses the same representation and neighborhood as in DA. A tabu list of length 7 stores values of the maximum completion time.
- *GA*. The genetic algorithm uses the natural representation suggested in Section 4.2, in which each string is a list of machines to which the jobs are assigned. Recombination consists of performing a two-point crossover and mutating each element by changing the machine assignment of the corresponding job. The population size is 100.
- *GD*. The genetic descent algorithm is a variant of GA in which DA is applied to each new solution that is generated. The population size is 20.

The earliest completion time (ECT) heuristic of Ibarra & Kim [1977] provides the initial solution for the neighborhood search algorithms DA, SA, and TS.

Moreover, the initial populations in GA and GD are seeded with two copies of the ECT solution. For several solutions with the same maximum completion time, one with the smallest number of machines achieving the maximum machine load is generally preferred. This preference is incorporated into the acceptance rule for the neighborhood search algorithms.

Glass, Potts & Shade [1994] generate 20 test problems for various combinations of m and n, where some of the processing time matrices contain rows and columns that are correlated. The five algorithms are first run using a time limit of 20 seconds on an IBM 3090 computer. Then the more successful algorithms, SA, TS, and GD, are run using a time limit of 100 seconds. Average relative percentage deviations from the best known solution values are listed in Table 11.2.

Algorithm GA performs very poorly in this study, even compared with DA, which uses only a small proportion of the allowed computation time. However, when descent is incorporated into GA, we observe from the results for GD that the algorithm is competitive. This substantiates a widely held belief that, for a genetic algorithm to yield high-quality solutions, it needs to incorporate another heuristic or some problem-specific features. With the time limit of 20 seconds, TS performs marginally better than the other algorithms. However, both SA and GD improve substantially with additional run time, and the performances of SA, TS, and GD are comparable when the time limit is 100 seconds.

8.3 Sequencing of jobs in $F \| \sum w_j C_j$

Glass & Potts [1996] describe a computational study for the permutation flow shop problem $F \| \sum w_j C_j$, which requires the minimization of total weighted completion time. In contrast to the case of minimizing the maximum completion time, there is no efficient update mechanism for computing the objective function for a new solution generated by a local search algorithm. The study compares six local search algorithms, described below. In each case the natural representation of solutions as sequences is adopted.

- *MD*. Multistart descent repeatedly applies descent, using different randomly generated initial sequences, until a prespecified time limit is exceeded. In initial experiments with the insert and swap neighborhoods of Section 4.1, swap yields slightly better results. Thus, the swap neighborhood is adopted.
- *SA*. The simulated annealing algorithm uses the swap neighborhood. Moreover, a geometric cooling schedule is used since it performs better than the scheme of Lundy & Mees [1986] in initial experiments.
- *TA*. Threshold accepting is implemented with the swap neighborhood, and the threshold value decreases linearly after each iteration. A quadratic decrement of threshold values produces inferior results in initial experiments.
- *TS*. The tabu search method also uses the swap neighborhood. According to initial experiments, slightly better results are obtained with a tabu list that prevents either of the swapped jobs from returning to its original position,

Table 11.3 Average relative deviations (%) for $F \| \sum w_j C_j$

m	n	MD	SA	TA	TS	GA	GD
4							
	20	0.28	0.21	0.35	0.21	0.77	0.24
	30	0.33	0.19	0.22	0.70	1.21	0.15
	40	0.64	0.24	0.30	1.09	1.51	0.23
	50	0.73	0.37	0.26	1.49	1.64	0.28
10							
	20	0.16	0.12	0.27	0.35	1.09	0.20
	30	0.69	0.50	0.74	1.51	2.03	0.32
	40	1.08	0.79	0.95	1.89	2.32	0.45
	50	0.93	0.67	0.95	1.88	2.35	0.48
Average		0.61	0.39	0.51	1.14	1.62	0.29

compared with a tabu list that maintains the order between the two swapped jobs. The former list is adopted, and the tabu list length is dependent on the number of jobs.

- *GA*. The genetic algorithm employs the insertion crossover. The solutions obtained in initial experiments with the reorder crossover are inferior. A population size of 50 is used.
- *GD*. The genetic descent algorithm is a variant of GA in which a descent algorithm with the swap neighborhood is applied to each newly generated solution to find a local minimum. The reorder crossover is used, which is more effective than insertion crossover, according to initial experiments. The population size is 10.

Glass & Potts [1996] generate 10 test problems for each of the selected values of m and n. The six algorithms are each run three times on an IBM 3090 computer. Each run is initialized by randomly generating a sequence or a population, and has a time limit of 100 seconds. Average relative percentage deviations from the best known solution values are presented in Table 11.3.

The algorithms that generate the best-quality solutions are SA and GD. Although SA gives better results for the 20-job problems, GD is generally superior. This is especially noticeable for the large problems with 10 machines and 40 or 50 jobs. As in Section 8.2, GA gives the worst results. The performance of TS is rather disappointing. Algorithms MD and TA both give reasonable results, although they are inferior to both SA and GD.

8.4 Multisequencing of jobs in $J \| C_{max}$

As observed in Section 7.3, various local search algorithms are available for the job shop problem $J \| C_{max}$. Since most of the computational experiments use

the same test problems, Vaessens, Aarts & Lenstra [1996] and Balas & Vazacopoulos [1998] are able to provide a comparison between the quality of solutions generated by most of the methods. We provide similar computational results for a selection of algorithms. In addition to choosing a variety of different types of algorithms, our comparison includes all algorithms which are not dominated by others in terms of solution quality and computation time in the results given by Vaessens, Aarts & Lenstra and Balas & Vazacopoulos. The following algorithms are considered:

- *SB1.* This method is the enumerative version of the shifting bottleneck procedure of Adams, Balas & Zawack [1988], which they call SBII. The results of Vaessens, Aarts & Lenstra [1996] show that it is among the most competitive variants of this type of procedure (excluding those that incorporate guided local search).
- *SB2.* A hybrid algorithm of Balas & Vazacopoulos [1998], which they call SB-GLS2, embeds guided local search into the shifting bottleneck procedure. The guided local search uses the critical end insert neighborhood.
- *SB3.* This is another hybrid of Balas & Vazacopoulos [1998], which they call SB-RGLS1. It extends the search in SB2 by rescheduling $[\sqrt{m}]$ machines using a combination of guided local search and shifting bottleneck routines.
- *MD.* A multistart descent algorithm of Aarts et al. [1994], which the authors call MSII2, employs the extended version of the critical end transpose neighborhood, where transposes of predecessor and successor operations are attempted, as proposed by Matsuo, Suh & Sullivan [1988]. The solution quality with this algorithm is significantly better than obtained with the critical transpose neighborhood.
- *SA1.* A simulated annealing algorithm of Aarts et al. [1994] uses the critical transpose neighborhood and adopts the cooling schedule described by Van Laarhoven, Aarts & Lenstra. Algorithm SA1* is a variant of SA1 in which a much slower cooling schedule is used.
- *SA2.* A simulated annealing algorithm of Matsuo, Suh & Sullivan [1988], which the authors call CSSA, uses the extended version of the critical end transpose neighborhood, and employs an acceptance probability that is independent of objective function values.
- *SA3.* A simulated annealing algorithm of Yamada, Rosen & Nakano [1994], which the authors call CBSA, uses the critical end insert neighborhood, and employs a backtracking procedure.
- *TA.* Threshold accepting, as implemented by Aarts et al. [1994] with the critical transpose neighborhood and named TA1, uses threshold values that are multiples of those proposed by Dueck & Scheuer [1990].
- *TS1.* The tabu search algorithm of Barnes & Chambers [1995] uses the critical transpose neighborhood, and employs a backtracking procedure.
- *TS2.* The tabu search algorithm of Dell'Amico & Trubian [1993] uses a composite neighborhood consisting of generalized critical end transpose and critical end insert.

- *TS3*. The tabu search algorithm of Nowicki & Smutnicki [1996a], which the authors call TSAB, uses the critical end transpose neighborhood, and employs a backtracking procedure.
- *GS*. The guided local search procedure of Balas & Vazacopoulos [1998], which they call GLS/1, uses the critical end insert neighborhood.
- *GA1*. The genetic algorithm of Della Croce, Tadei & Volta [1995] uses a priority representation.
- *GA2*. The first genetic algorithm of Dorndorf & Pesch [1995], which the authors call P-GA, uses a heuristic set representation.
- *GA3*. The second genetic algorithm of Dorndorf & Pesch [1995], which they call SB-GA(40), is based on the shifting bottleneck procedure.
- *GA4*. The genetic algorithm of Pesch [1993], which the author calls 1MCP-GA, uses a representation that sets upper bounds for single-machine subproblems. The results of Pesch indicate that it is superior to his other genetic algorithms.
- *GD*. The genetic descent algorithm of Aarts et al. [1994], which they call GLS2, uses a descent algorithm which employs the extended version of the critical end transpose neighborhood.

For some of these algorithms, a single run is performed. However, the performance of MD, SA1, SA3, TA, TS2, and GD is averaged over five runs of the algorithm, whereas results for GA3 are averaged over two runs. Moreover, results for multistart versions of TS3 (with three starts) and GS (with four starts) are also available. A superscript indicates that the best of several runs from different starting solutions is selected. For example, $TS3^3$ refers to the best of three runs of algorithm TS3.

We compare algorithms by providing computational results from the relevant papers for a subset of the test problems of Fisher & Thompson [1963] and Lawrence [1984]. These instances are chosen to be among the 'hardest', and are therefore able to discriminate between the different algorithms. The instances considered are FT2 and FT3 of Fisher & Thompson that have $m \times n$ equal to 10×10 and 5×20 respectively, and those of Lawrence that are of type A (10×10), type B (10×15), type C (10×20), and type I (15×15).

In the computational results that are quoted in the literature, different computers are used and computation times vary from one algorithm to another. Thus, consideration should be given to both solution quality and computation time when interpreting these results. Table 11.4 shows (average) values of the objective function value C_{max}. The column OPT gives the optimal value of the maximum completion time or, when this is not known, the best known solution value is listed and marked with an asterisk. Also listed for each algorithm is the average relative deviation (ARD) of the maximum completion time from the best solution value, expressed as a percentage, where the averages are over the 10 test problems (or less for some methods because results are unavailable). Vaessens, Aarts & Lenstra [1996] are able to standardize computation times so that they are independent of the computer on which the tests are performed. Figure 11.1

Table 11.4 Average values for C_{max} for $J \| C_{max}$: best know solutions are marked with an asterisk

	OPT	SB1	SB2	SB3	MD	SA1	SA2	SA3	TA	TS1	TS2	TS3	GS	GA1	GA2	GA3	GA4	GD
FT2	930	930	930	930	1 057	969	946	931	1 004	930	948	930	930	965	960	938	930	982
FT3	1 165	1 178	–	–	1 329	1 216	–	–	1 229	–	1 167	1 165	–	1 199	1 249	1 178	1 165	1 294
A1	945	978	945	945	1 019	977	959	–	978	947	946	946	945	989	1 008	961	945	977
A4	842	860	842	842	900	855	842	–	926	848	847	842	846	–	880	863	842	859
B1	1 046*	1 084	1 048	1 048	1 181	1 084	1 071	1 055	1 104	1 053	1 057	1 055	1 053	1 114	1 139	1 074	1 070	1 085
B4	935	976	937	937	1 076	962	973	947	1 015	946	944	948	950	–	1 014	960	940	981
C2	1 235	1 291	1 240	1 235	1 448	1 282	1 274	1 266	1 290	1 250	1 252	1 259	1 244	–	1 378	1 272	1 268	1 301
C4	1 157*	1 239	1 164	1 164	1 426	1 233	1 196	1 193	1 262	1 219	1 195	1 164	1 157	–	1 336	1 204	1 204	1 260
I1	1 268	1 305	1 268	1 268	1 457	1 308	1 292	–	1 386	1 278	1 289	1 275	1 269	1 330	1 373	1 317	1 273	1 311
I3	1 196*	1 255	1 198	1 198	1 442	1 235	1 231	1 214	1 323	1 211	1 210	1 209	1 213	–	1 296	1 251	1 204	1 283
ARD		3.4	0.2	0.1	14.5	3.7	2.3	1.6	7.4	1.3	1.2	0.7	0.6	4.5	8.3	2.7	1.1	5.6

ARD = Average relative deviation (%) of heuristic solution value from that of the best known solution.

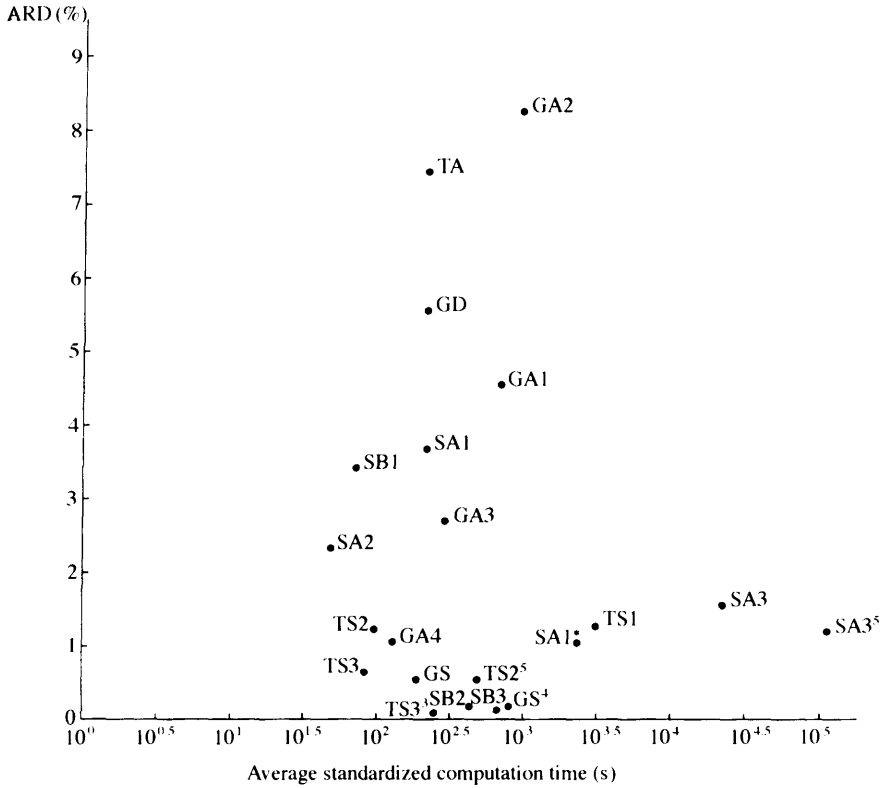

Figure 11.1 Relationship between average relative deviations (%) and standardized average computation time (in seconds)

depicts the relationship between the standardized average computation time in seconds, and the solution quality as measured by the average relative deviation ARD for the various algorithms. We omit MD from Figure 11.1 because of its large average relative deviation of 14.5%, but results for SA1*, $SA3^5$, $TS2^5$, $TS3^3$, and GS^4 are included, even though they do not appear in Table 11.4.

We first note from Table 11.4 that SB1 generates reasonable-quality solutions, whereas Figure 11.1 shows that its average computation time is modest. However, substantial improvements are obtained by including guided local search, as indicated by the results for SB2 and SB3. MD is clearly uncompetitive. Of the three simulated annealing algorithms, SA2 performs best when computation time is limited. However, comparison of SA1 and SA1* in Figure 11.1 shows that this type of algorithm can generate very good solutions if sufficient computation time is allocated. Results also show that the threshold accepting method TA is one of the least effective algorithms.

The three tabu search algorithms all produce solutions of high quality. Since the extra computational requirements of TS1 are not rewarded through better-

quality solutions, TS2 and TS3 are preferred. Moreover, Figure 11.1 shows that TS3 gives slightly better results than TS2, and $TS3^3$ generates solution values that are extremely close to the optimum. The guided local search method GS generates reasonable-quality solutions without using excessive computation time, and these solutions can be improved with a multistart version GS^4. Of the four genetic algorithms, GA4 is superior, and it is also preferred to the genetic descent algorithm GD. Taking an overview of all results, the tabu search algorithm TS3, and the multistart version $TS3^3$ are among the most competitive methods. However, high-quality solutions are also obtained with the two hybrid shifting bottleneck/guided local search methods SB2 and SB3. There is insufficient evidence at present to make a strong claim for the superiority of $TS3^3$ over SB2 and SB3.

9 CONCLUDING REMARKS

It is apparent from our discussion in the previous sections that local search techniques are the method of choice for NP-hard scheduling problems when the problem size makes a branch-and-bound approach (or other enumerative methods) impractical. Recent work on job shop problems demonstrates the power of these approaches in producing excellent solutions for intractable problems. Most studies concentrate on 'standard' scheduling models which do not contain many of the complicating features that are present in practical situations. Nevertheless, the flexibility of local search techniques enables such complications to be handled more effectively than with most other approaches.

Recent research with neighborhood search algorithms reveals some useful innovations. One successful idea is to consider a suitably chosen subset of some neighborhood, where the choice may be based on some problem-specific features. Another useful device is to allow backtracking. Using this technique, a sequence of neighborhood moves is provisionally accepted so that their effect can be explored, but if no overall improvement in solution value is observed after a specified number of iterations, the sequence of moves is rejected, and the search continues from the solution obtained prior to these exploratory moves.

The performance of genetic algorithms tends to be variable. When used as a 'black-box' technique, a genetic algorithm usually fails to generate good-quality solutions. However, by incorporating some problem-specific knowledge, which can be achieved by the use of a suitable representation of solutions, or by embedding a heuristic such as a descent method, the performance of a genetic algorithm often improves substantially.

We are not aware of any comparative computational studies for scheduling problems involving neural networks. Moreover, there is no current evidence that these methods are competitive with neighborhood search or genetic algorithms.

It is appropriate to add some comments on the empirical work that is necessary when evaluating local search algorithms. For many scheduling applications, we have found substantial difficulties in drawing firm conclusions on the relative merits of different methods. It is vital to experiment carefully with respect to

individual problem characteristics (particularly size), the computation time allowed, and the method used for generating a starting solution. There are also difficulties in knowing how much tuning of parameters is appropriate for some methods. It is remarkably easy to arrive at a faulty conclusion because of insufficient experimental work.

Finally, although there has been a large amount of research on the application of local search to scheduling problems, much remains to be done. First, the ideas we have emphasized in this chapter, such as on different representations of solutions, provide many possible avenues of research that have yet to be explored. Second, there is a wide range of different local search techniques, and it is not yet clear for which problem types the different techniques are most suited. Third, now that the potential of local search methods has been demonstrated, mainly on 'standard' problems, their use should be exploited by applying them to a broad range of more difficult practical problems.

Local Search in Combinatorial Optimization
Edited by E. Aarts and J. K. Lenstra
© 1997 John Wiley & Sons Ltd.

12

VLSI layout synthesis

Emile H. L. Aarts
Philips Research Laboratories, Eindhoven

Peter J. M. van Laarhoven
Eindhoven University of Technology

C. L. Liu
University of Illinois at Urbana-Champaign

Peichen Pan
Clarkson University, Potsdam

1 LAYOUT SYNTHESIS 415
2 SIMULATED ANNEALING 417
2.1 Placement 419
2.2 Floorplanning 421
2.3 Channel routing 423
2.4 Compaction 425
3 TABU SEARCH 429
3.1 Partitioning 430
3.2 Placement 432
4 GENETIC ALGORITHMS 433
4.1 Floorplanning 434
4.2 Global improvement over macro cell layouts 436

1 LAYOUT SYNTHESIS

Layout synthesis in VLSI chip design refers to the process of transforming the structural specification of a circuit in the form of modules and interconnects to detailed geometrical data and processing information for chip production. The two main tasks in layout synthesis are allocating spaces to the modules to be placed on a chip, and interconnecting points on the modules, called terminals, according to a prescribed netlist. Each net in the netlist specifies a set of terminals that must be electrically connected. The objective in layout synthesis is usually to minimize the total chip area, though nongeometrical issues such as performance and power consumption have recently been considered. Layout synthesis is an extremely complex problem, impossible to tackle as a whole to arrive at an

optimal solution. In practice the problem is broken up into several subproblems which are then solved individually. The solutions to the subproblems are combined to yield a solution to the overall problem. The combined solution to the overall problem is most likely suboptimal since the objective in a subproblems can only partially reflect the overall objective. Actually, most subproblems are still too difficult to be solved to optimality. Heuristic algorithms that seek good but not necessarily optimal solution are employed. Typical subproblems in layout synthesis are partitioning, floorplanning/placement, routing, and compaction.

- *Partitioning* is the process of breaking up a large design into several smaller parts. The usual objective is to minimize the number of nets that have terminals in more than one part. Among the many applications of partitioning in layout synthesis, one is to break up a large design so that each resulting part can be efficiently and effectively managed and optimized.
- *Placement* is the process of placing without overlaps modules of fixed sizes on a two-dimensional surface representing the chip. The purpose is to facilitate routing with an overall objective such as minimizing the total chip area. If the modules have flexible sizes and shapes, one often speaks of *floorplanning*. Placement is often classified according to the targeted design methodology, e.g., gate array placement, standard cell placement, and macro cell placement.
- *Routing* is the process of connecting terminals on modules as specified by the netlist while satisfying both physical constraints and design rules. Due to its complexity, routing is conventionally done in two phases, *global routing* and *detailed routing*. Global routing can be viewed as a routing planning phase. Its purpose is to determine the routing regions that each net is going to pass through. After global routing, detailed routing is carried out in each routing region. Detailed routing produces detailed information for realizing the nets passing through a routing region in terms of tracks and vias. Depending on the types of routing regions or routing resources available, detailed routing can be classified into channel routing, switchbox routing, single-layer routing, two-layout routing, etc.
- *Compaction* is the process of further minimizing the layout area of a design without changing the topological relations of the layout components. Compaction is subject to design rules, which are constraints that specify the relative positions of the components associated with a given IC process technology. Compaction is also used to migrate a layout from one processing technology to another.

For more detailed discussions of layout synthesis and various layout problems, we refer the reader to Lengauer [1990] and Preas & Lorenzetti [1988].

Over the years much effort had been invested in designing efficient and effective algorithms for various layout problems. For some of the early work see Soukup [1981]. For more recent work see Kuh & Ohtsuki [1990]. The success of the existing algorithms was, however, far from satisfactory. On the one hand this was because almost all layout problems are NP-hard [Sahni & Bhatt, 1980], which

makes it unlikely that optimal solutions can be found in a reasonable amount of time. Therefore, heuristics for finding approximate solutions are often employed. On the other hand it appeared to be very hard to construct efficient and effective approximation algorithms that could handle the many different aspects in layout problems.

The introduction of generally applicable combinatorial optimization methods, such as simulated annealing, tabu search, and genetic algorithms, initiated a wave of research on their application to layout problems. Local search methods are attractive for their flexibility, which helps them to tackle complex problems such as arise in layout synthesis, problems in which too many factors need to be considered. Also attractive is the trade-off they allow between quality of solutions and computation time. And their relative ease of implementation makes them very popular, too. In this chapter we present some work on applying simulated annealing, tabu search, and genetic algorithms to solve several important layout problems.

2 SIMULATED ANNEALING

Simulated annealing was introduced by Kirkpatrick, Gelatt & Vecchi [1983] and independently by Cerný [1985] as a general technique for approximately solving combinatorial optimization problems; see Chapter 4.

Simulated annealing can be viewed as a probabilistic version of conventional local search with the additional ability of moving out of local optima. Conventional local search starts with an initial solution. It then searches for a better solution in the neighborhood of the current solution. If a better solution is found, it is accepted as the new current solution. This process is continued until no better solution is found, which terminates the search process. A conspicuous shortcoming of conventional local search is that it terminates at the very first local optimum. Another shortcoming is that the final solution strongly depends on the choice of the initial solution.

Simulated annealing is also based on the idea of exploring the solution space of a problem by moving around in the neighborhood structure. However, there are several pronounced distinctions between conventional local search and simulated annealing. First, simulated annealing randomly generates a solution in the neighborhood of the current solution instead of searching for a better one. Second, whether the generated solution will be accepted is decided by an acceptance criterion that allows both improvements and, in a probabilistic way, deteriorations in the cost function. Deteriorations are initially accepted with a large probability. The probability is slowly decreased to become zero at the end. Such decrease in acceptance probability is accounted for by a control parameter called *temperature*. The higher the temperature, the higher the probability that deteriorations will be accepted. Allowing deteriorations enables simulated annealing to escape from local optima. Also, with high temperature at the beginning, most solutions are accepted. This makes the final result more or less independent of the choice of the initial solution.

We now give the framework of the homogeneous simulated annealing algorithm; this is the simplest algorithm and it is widely used. The temperature is decreased step by step. For each temperature value, it generates a sequence of solutions of fixed length, say L_t for the tth temperature value c_t, following the generation mechanism and the acceptance criterion. Let f denote the objective function, and $\mathcal{N}(i)$ denote the neighborhood of solution i of an optimization problem. (We always assume the smaller the cost the better the solution.) Suppose i is the current solution. Suppose $j \in \mathcal{N}(i)$ is generated by the generation mechanism. j is accepted as the new current solution with the probability computed according to an *acceptance function*. The most common acceptance function is $\exp(-(f(j)-f(i))^+/c)$, where c is the current temperature, and $(f(j)-f(i))^+ = f(j)-f(i)$ if $f(j)-f(i)>0$, and $(f(j)-f(i))^+ = 0$ otherwise. There are many other acceptance functions [Nahar, Sahni & Shragowitz, 1985]. They usually have the property that a solution with improved cost is always accepted and a solution with deteriorated cost is more likely to be accepted in high temperature than in low temperature. Another important component is the temperature decrement function, which is usually referred to as the *cooling schedule*. The cooling schedule affects both the quality of the solution and the running time. A commonly used decrement function is $c_{t+1} = \alpha c_t$ where α is a positive constant smaller than but close to 1. For other cooling schedules see Aarts et al. [1985, 1986], Hajek [1988], Huang, Romeo & Sangiovanni-Vincentelli [1986], Lam & Delosme [1988b]. Figure 12.1 shows the general framework of the algorithm. For more advanced topics on simulated annealing, we refer the reader to Aarts & Korst [1989a], Faigle & Schrader [1988], Gidas [1985], Van Laarhoven [1988], Van Laarhoven & Aarts [1987], Lundy & Mees [1986], and Mitra, Romeo & Sangiovanni-Vincentelli [1986].

```
begin SA
  select an initial solution i_1;
  select an initial temperature c_1;
  t ← 1;
  while the stopping condition is not met do
      count ← 0;
      while count < L_t do
        count ← count + 1;
        i ← a random solution in N(i_t);
        i_t ← i with probability exp(-(f(i) - f(i_t))^+ / c_t);
      end while
      c_{t+1} ← αc_t;
      i_{t+1} ← i_t;
      t ← t + 1;
  end while
end SA
```

Figure 12.1 Homogeneous simulated annealing algorithm

Application of the algorithm to an optimization problem requires the following items to be specified: a concise representation of the solutions, a neighborhood structure on the solution space, and the cooling schedule. In practice, the algorithm stops when the improvement on the cost function between two consecutive temperature values is small enough. The final output can be either the last solution or the best solution ever observed; see also Chapter 4.

Simulated annealing has been used to tackle problems in layout synthesis. For an overview see Van Laarhoven & Aarts [1987], and Wong, Leong & Liu [1988]. Simulated annealing is considered to be one of the most effective methods for solving many layout problems in terms of the solution quality. As an example, we mention the placement and routing package Timber Wolf by Sechen & Sangiovanni-Vincentelli [1985]. This package has been improved over the years [Sechen, 1985b; Sechen & Lee, 1987; Sechen & Sangiovanni-Vincentelli, 1986; Sun & Sechen, 1993] and is widely used.

We will now discuss the application of simulated annealing to four layout problems: placement, floorplanning, channel routing, and compaction.

2.1 Placement

Here we present a simulated annealing algorithm for the sea-of-gates placement problem. The problem is to assign the modules to the points of a rectangular grid such that the estimated total wire length is minimized.

Let $G = (V, E)$ denote the circuit, where V represents the set of modules and E the set of nets (two terminal nets are assumed). Edges are weighted. A weight on an edge represents the connectivity between the two modules. Let $w(u, v)$ denote the weight of edge $\{u, v\}$. A *placement* is an assignment of the modules to the points of an $N \times M$ rectangular grid, with $NM \geqslant |V|$. Furthermore, let $x_i(u)$ and $y_i(u)$ denote the x and y coordinates, respectively, of module u in placement i.

The application of simulated annealing to this problem is straightforward and can be carried out as follows:

- The solution space is given by the set of all possible placements.
- The cost of placement i is chosen to be

$$f(i) = \sum_{\{u,v\} \in E} w(u, v) \{(x_i(v) - x_i(u))^2 + (y_i(v) - y_i(u))^2\}. \tag{1}$$

- The neighborhood of a placement is generated by exchanging the modules assigned to two grid points. The difference in cost between two neighboring solution i and j obtained by exchanging modules u and v in i then equals

$$\begin{aligned}\Delta f &= f(j) - f(i) \\ &= \sum_{\{u,z\} \in E_u} w(u, z) \{(x_j(z) - x_j(u))^2 + (y_j(z) - y_j(u))^2\} \\ &\quad - \sum_{\{u,z\} \in E_u} w(u, z) \{(x_i(z) - x_i(u))^2 + (y_i(z) - y_i(u))^2\}\end{aligned}$$

$$+ \sum_{\{v,z\} \in E_i} w(v,z)\{(x_j(z) - x_j(v))^2 + (y_j(z) - y_j(v))^2\}$$
$$- \sum_{\{v,z\} \in E_i} w(v,z)\{(x_i(z) - x_i(v))^2 + (y_i(z) - y_i(v))^2\}, \qquad (2)$$

where E_u and E_v denote the sets of edges incident on u and v, respectively, and

$$x_j(u) = x_i(v),$$
$$y_j(u) = y_i(v),$$
$$y_j(z) = y_i(z) \text{ and } y_j(z) = y_i(z), \quad \text{for all } z \neq u, v.$$

Calculation of (2) can be done in $O(m)$ time, whereas calculation of (1) requires $O(m^2)$ time, where $m = N \times M$. Hence, (2) allows more efficient calculation of cost differences.

Experiments were conducted to compare the performance of simulated annealing with that of iterative improvement; the results are shown in Table 12.1. Column n lists the numbers of modules for the problems. The table gives the minimum cost W_{opt}, the cost of a random placement W_r, the cost of the placement obtained by iterative improvement W_{ii}, and by simulated annealing W_{sa}. The corresponding running times in seconds are t_{ii} and t_{sa}, respectively. The costs for simulated annealing are the averages over five runs with different initial solutions. From the results we conclude that the running times of simulated annealing become prohibitive for the larger problem instances (up to 15 hours for problem p_7). Furthermore, simulated annealing yields substantially better results than iterative improvement, whereas the running time of iterative improvement is only three to four times smaller than for simulated annealing. Considering the quality of the final results and the computation times, it suggests that simulated annealing also outperforms time-equivalent multistart iterative improvement, i.e., iterative improvement repeated a number of times with different initial solutions such that the total running time is about equal to that of simulated annealing.

Table 12.1 Results of sea-of-gates placement by simulated annealing and iterative improvement for seven problems

Problem	n	W_{opt}	W_r	W_{ii}	t_{ii}	W_{sa}	t_{sa}
p_1	400	760	81 284	7 584	202	1 256	872
p_2	625	1 200	216 445	14 429	479	2 261	2 348
p_3	900	1 740	454 897	23 728	1 343	3 150	5 581
p_4	1 225	2 380	846 986	34 835	4 567	4 013	11 182
p_5	1 600	3 120	1 465 858	56 078	9 936	5 836	20 008
p_6	2 025	3 960	2 362 427	86 753	15 289	6 793	32 483
p_7	2 500	4 900	4 091 694	142 055	18 478	8 845	48 899

For other placement algorithms based on simulated annealing, see De Bont et al. [1988], Greene & Supowit [1984], Grover [1986], Jepsen & Gelatt [1983], Kim et al. [1993], Kraus & Mlynski [1991], Mallela & Grover [1988], Nagahara et al. [1989], Sechen [1988b], Sechen & Lee [1987], Sechen & Sangiovanni-Vincentelli [1985], Siarry, Bergonzi & Dreyfus [1987], and Sun & Sechen [1993].

One drawback of simulated annealing algorithms is that they are very time-consuming. To alleviate this problem, a number of parallel simulated annealing placement algorithms were proposed: Banerjee & Jones [1986], Casotto, Romeo & Sangiovanni-Vincentelli [1987], Darema-Rogers, Kirkpatrick & Norton [1987], Kravitz & Rutenbar [1987], RaviKumar & Patnaik [1987], Rose et al. [1986], and Wong & Fiebrich [1989].

2.2 Floorplanning

In the floorplanning problem we are given n modules with aspect ratios (height/width) that may vary within limits determined by *shape constraints*. Let $(A_1, r_1, s_1), (A_2, r_2, s_2), \ldots,$ and (A_n, r_n, s_n) be the triples specifying the area A_t and the bounds r_t and s_t of the shape constraints of module t. Furthermore, for $t = 1, \ldots, n$, let h_t and w_t denote the respective height and width of module t. Then it must be the case that

$$w_t h_t = A_t \quad \text{and} \quad r_t \leqslant \frac{h_t}{w_t} \leqslant s_t \quad \text{for} \quad t = 1, \ldots, n.$$

The objective is to place the modules in a rectangular floorplan such that they do not overlap, their aspect ratios satisfy the shape constraints, and a weighted sum of the estimated total wire length and area of the floorplan is minimized.

Wong & Liu [1986] proposed a floorplanning algorithm based on simulated annealing. They restrict themselves to slicing floorplans. A *slicing floorplan* is a rectangle dissection obtained by recursively cutting rectangles into two rectangles (Figure 12.2). A slicing floorplan can be represented by a binary tree, called a *slicing tree*, which describes the way that the floorplan is constructed. The leaves of the tree correspond to the modules. The internal nodes correspond to the cut lines. Figure 12.3 shows the slicing tree for the slicing floorplan in Figure 12.2. If we associate operator + to a horizontal cutline and $\star$ to a vertical cutline in a slicing tree, it becomes an expression tree with the modules as operands. The *Polish expression* of the tree is used to represent the floorplan by Wong & Liu [1987]. For the floorplan in Figure 12.2, $12\star3\star4+6\star578+\star+$ is the corresponding expression, which can be obtained by a postorder traversal of the slicing tree in Figure 12.3.

The simulated annealing algorithm is specified in the following way:

- The solution space is given by the set of the Polish expressions on the modules.
- The cost of solution i is defined as

$$f(i) = A(i) + \lambda W(i),$$

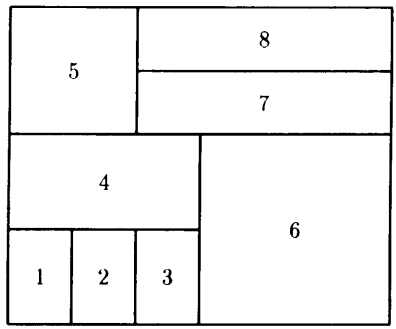

Figure 12.2 A slicing floorplan

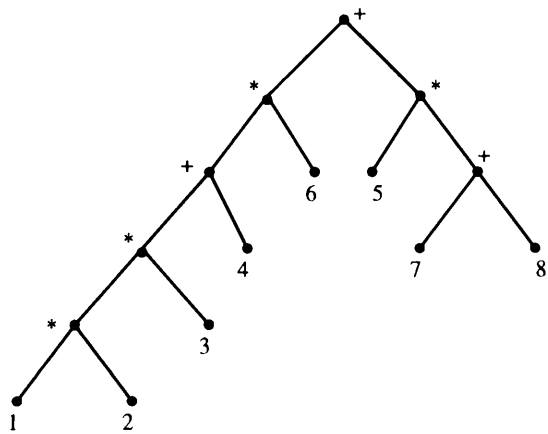

Figure 12.3 The slicing tree of the floorplan in Figure 12.2

where $\lambda \in \mathbb{R}^+$ is a weighting factor. The total area $A(i)$ of the floorplan can be obtained by running the area minimization algorithm by Otten [1982] to determine the shape constraint of the floorplan, then choosing the point with minimum area on the shape constraint of the floorplan. Once the shape of the floorplan is fixed, the shapes of all modules can also be determined in a top-down fashion. This also gives the total estimated length of connections $W(i)$. For each connection between modules u and v, its length is estimated by the distance between the centers of modules u and v in the current floorplan.

- The neighborhood is generated by one of the following techniques:

  - exchanging two consecutive operands in the current Polish expression;
  - interchanging $\star$ and $+$ in a subsequence of operators (e.g., a change of $\star + \star + \star$ into $+ \star + \star +$);
  - exchanging an operand with an adjacent operator in the current expression.

Table 12.2 Results of floorplanning by simulated annealing for eight problems

Problem	n	A_0	A_r	W_r	A_{sa}	W_{sa}
p_1	15	137.08	491.62	106.02	137.86	34.97
p_2	20	198.88	745.35	258.84	202.17	80.49
p_3	20	197.15	808.72	423.90	198.98	197.4
p_4	25	244.68	1 165.7	398.21	245.43	209.7
p_5	25	238.15	1 026.6	576.71	244.63	151.9
p_6	30	333.92	1 549.6	1 023.2	340.15	294.9
p_7	30	314.45	1 476.9	1 095.7	319.40	429.2
p_8	40	407.44	1 934.2	1 002.2	422.95	265.8

Efficient ways for calculating the differences in cost incrementally are also given by Wong & Liu [1986].

The algorithm was applied to a number of randomly generated problems, varying in size from 15 to 40 modules; see Table 12.2. Column n lists the numbers of modules for the problems. Table 12.2 gives the total area A_0 of the modules, the area A_r and wire length W_r of a random floorplan, and the area A_{sa} and wire length W_{sa} of the floorplan obtained by simulated annealing. CPU times on a PYRAMID computer range from 1 to 13 minutes. For all problems the area of the final floorplan obtained by simulated annealing is less than 4% larger than the total area of all modules (a theoretical lower bound for the global minimum).

For other floorplanning algorithms based on simulated annealing see Koakutsu, Sugai & Hirata [1992], Otten & Van Ginneken [1984], Saha, Mohan & Raghunathan [1989], Sechen [1988a], and Wong & Liu [1987].

2.3 Channel routing

In the (two-layer) channel routing problem we are given two sets of l terminals, $T = \{t_1, t_2, \ldots, t_l\}$ and $B = \{b_1, b_2, \ldots, b_l\}$, along the top and bottom boundaries of a rectangular grid representing the routing region. Each terminal is assigned a number which is an integer in $\{1, 2, \ldots, m\}$. The terminals with the same number form a net. The objective is to route the nets in such a way that the number of horizontal tracks in the channel is minimized. Figure 12.4 shows two solutions to a 3-net problem. Note that nets are not allowed to overlap. In this section we present two routing approaches based on simulated annealing.

The first algorithm is due to Leong, Wong & Liu [1985]. It is assumed that a net cannot change tracks; doglegs are not allowed. A *vertical constraint graph* $G_c = (V, A)$ can be constructed by associating each net with a vertex and defining an arc from u to v if the horizontal segment of net u must be placed above that of net v. (For the problem in Figure 12.4 there is an arc from 2 to 3.) Further-

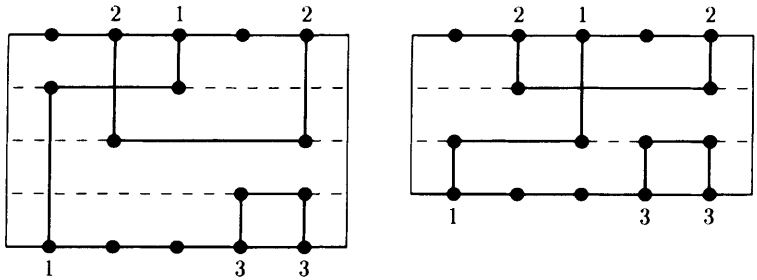

Figure 12.4 Two solutions of a 3-net

more, there is a *horizontal constraint* between vertices u and v if the intervals defined by the leftmost and rightmost terminals of nets u and v overlap. For the problem in Figure 12.4 there are horizontal constraints between 1 and 2 and between 2 and 3.

A partition $i = \{V_1, \ldots, V_m\}$ of the vertex set V induces a second graph G_i as follows: each subset V_p is associated with a vertex $\hat{v}_p$ in G_i, and there is an arc from $\hat{v}_p$ to $\hat{v}_q$ if there is an arc from some $v_r \in V_p$ to some $v_s \in V_q$ in G_c. A partition i is said to be *valid* if G_i is acyclic and there are no horizontal constraints between any two vertices in the same subset. For the problem in Figure 12.4, both $\{\{1\}, \{2\}, \{3\}\}$ and $\{\{1, 3\}, \{2\}\}$ are valid partitions. Leong, Wong & Liu [1985] showed that there is a one-to-one correspondence between the valid partitions of the nets and the solutions of the routing problem. Moreover, the number of subsets in a partition corresponds to the number of horizontal tracks in the routing solution.

Simulated annealing is applied in the following way:

- The solution space is given by the set of all valid partitions.
- The cost of partition i is chosen to be

$$f(i) = W(i)^2 + \lambda_1 P(i)^2 + \lambda_2 U(i),$$

  where $W(i)$ is the number of subsets of i. The second and third terms are added to the cost function to obtain distinct cost values for solutions with the same number of subsets; $P(i)$ (the length of a longest path in G_i) is a lower bound on the number of horizontal tracks needed for all partitions obtained from i by the second type of transitions [Leong, Wong & Liu, 1985], and $U(i)$ is a measure of the amount of unoccupied track, summed over all $W(i)$ tracks. λ_1 and λ_2 are weighting factors, which are experimentally determined.
- The neighborhood is generated by one of the following techniques:

  – exchanging two vertices belonging to different subsets in the current partition;
  – moving a vertex from one subset to another;
  – creating a new subset by separating a vertex from a subset.

Each newly generated partition should be checked to see whether it is valid. A number of ways for carrying out these checks efficiently were given by Leong, Wong & Liu [1985]. Furthermore, the authors present efficient strategies for calculating the differences in cost.

The algorithm was applied to a number of well-known problems from the literature as well as randomly generated problems. For the problems from the literature, including Deutsch's difficult problem [Deutsch, 1976], the algorithm found the same solutions as those obtained by traditional algorithms [Rivest & Fiduccia, 1982; Yoshimura & Kuh, 1982], but using considerably more computation time. On the randomly generated problems, it usually outperformed other routing algorithms, but the improvement was usually small.

Different simulated annealing approaches to the routing problem are proposed by Acan & Unver [1992] and Zargham [1992]. Unlike the previously discussed approach, these ones assume as fixed channel width; a net can change tracks. The objective is then to find a feasible routing (meaning that there is no overlap among different nets) with minimum total wire length and minimum number of vias. By minimizing the total wire length and the number of vias, the performance of the circuits can be increased and the manufacturing cost reduced.

The application of simulated annealing is carried out as follows:

- The solution space is given by the set of all possible ways of connecting the nets, including infeasible ones in which there are overlaps among different nets.
- The cost function is chosen to be

$$f(i) = \lambda_1 O(i) + \lambda_2 W(i) + \lambda_3 V(i),$$

  where $O(i)$, $W(i)$, and $V(i)$ are the total amount of overlaps, total wire length, and total number of vias of solution i, respectively. λ_1, λ_2, and λ_3 are weight factors. To stress the importance of the feasibility of a solution, λ_1 is usually chosen to be considerably larger than λ_2 and λ_3.
- The neighborhood is generated by one of the following techniques:
  - using the multiterminal Lee algorithm [1961] to modify the current routing of a net;
  - reshaping the current routing of a net by reassigning its wire segments to layers; the purpose here is to change the number of vias.

This approach can handle the switchbox routing problem as well. Another advantage of this approach is that doglegs can be introduced automatically.

All channel routing benchmarks in the literature are tested on this algorithm. Good improvements in terms of the wire length and the number of vias are observed [Acan & Unver, 1992].

For other simulated annealing routers see Leong & Liu [1987], Rossi [1990], and Vecchi & Kirkpatrick [1983].

2.4 Compaction

In the compaction problem we are given a layout, consisting of a number of

(layout) components such as modules and wires. A module usually has fixed dimensions; a wire has a fixed width but is lengthwise stretchable and shrinkable. If a wire is connected to a module, it can slide along the boundary of that module but it cannot turn around a corner. In other words, topological relations exist between modules and wires. The problem is to minimize the area of the bounding rectangle while preserving these relations.

An example is shown in Figure 12.5(a); an optimally compacted layout is shown in Figure 12.5(d). The compacted layout has shrunk some of the wires to zero length, e.g., for three of the five wires connecting modules 1 and 3. And some wires have a negative length, shifting modules 4 and 5 from a position to the left of modules 6 and 7 in the initial layout to a position to the right in the compacted layout.

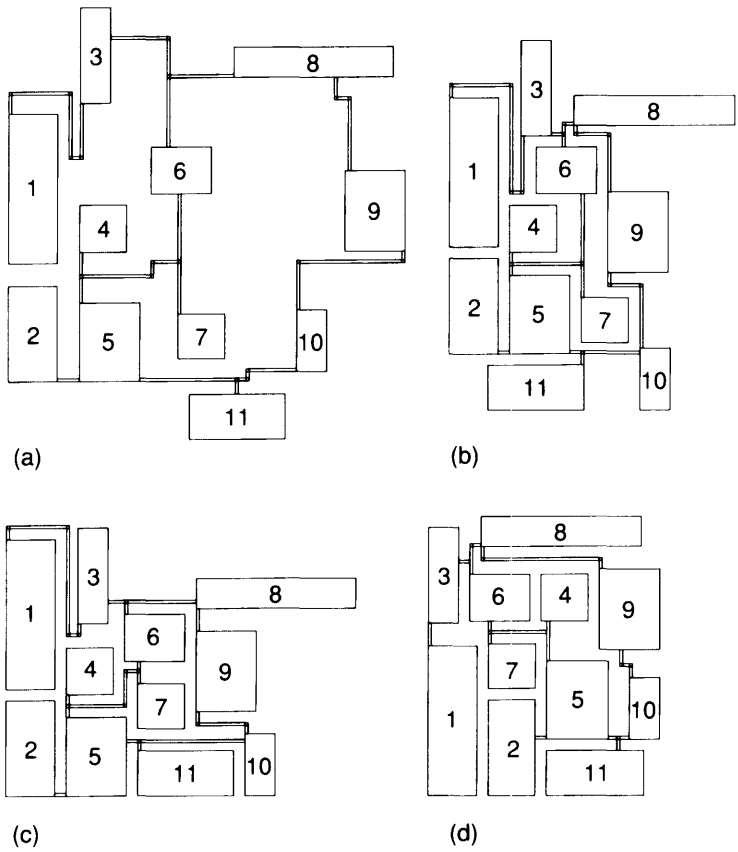

Figure 12.5 An instance of a compaction problem: (a) initial solution, (b) solution obtained after x-compaction followed by y-compaction, (c) solution obtained after y-compaction followed by x-compaction, (d) solution obtained after two-dimensional compaction

For component u there is a pair of variables x_u and y_u, which are the coordinates of either the center point of a module or one of the endpoints of the centerline of a wire. The topological relations can be expressed as (in)equalities to be satisfied by these variables. Following Schalg, Liao & Wong [1983], these inequalities are called the *base constraints*. For a more detailed description the reader is referred to Schalg, Liao & Wong [1983].

A pair of components is called a *free pair* if they are two modules not connected by wires (e.g., modules 1 and 2 in Figure 12.5), or a module and a wire that is not connected to that module. The topological relation between a free pair of components may vary from solution to solution; consider the relation between modules 1 and 2 in the initial and compacted layouts. For each free pair the variables have to satisfy certain inequalities in order to prevent modules and wires from overlapping each other. More precisely, for each free pair (u, v) one of four possible inequalities must be satisfied, so that u is either to the left, to the right, on top, or below v. We call these inequalities *distance constraints*. The number of free pairs can be substantially reduced by applying *pruning rules* [Hsieh, Leong & Liu, 1987]. For example, if modules u and v are connected by a single wire and modules v and w by a single wire in the same dimension, the relative positions of the pairs (u, v), (v, w), and (u, w) are fixed.

A solution of the problem can now be found by choosing a set of distance constraints, one for each free pair of components, and computing the area of the bounding rectangle of the resulting layout. Thus, after applying the pruning rules, if there are m free pairs, the number of solutions is 4^m. The number of *feasible* solutions is much smaller, since many choices will give rise to conflicts. The area of the bounding rectangle can be computed by constructing two directed graphs, one for the x-dimension and one for the y-dimension. The nodes of a graph correspond to the variables (in one dimension), the arcs to the inequalities (base and distance constraints) to be satisfied by the variables: An inequality $x_u + 2 \leqslant x_v$ produces an arc from node u to node v in the x-graph with weight 2. The length of the longest path in the x-graph (y-graph) then equals the dimension of the bounding rectangle in x-dimension (y-dimension). Furthermore, the feasibility of a solution is equivalent to the requirement that neither of the graphs contain positive cycles.

To illustrate these concepts, we return to Figure 12.5. In Figure 12.5(a) the distance constraint chosen for the free pair (1, 2) is apparently '1 on top of 2', creating arc (2, 1) in the y-graph. In the solution shown in Figure 12.5(d) the choice is '1 to the left of 2', creating arc (1, 2) in the x-graph. In the compacted layout, the longest path in the x-graph connects nodes 3, 6, 4, and 9; in the y-graph it connects nodes 11, 10, 9, and 8.

Simulated annealing can be applied as follows:

- The solution space is given by the 4^m possible choices of the m distance constraints.
- The cost of a solution is given by the product of the length of a longest path in the x-graph and the length of a longest path in the y-graph.

- To generate the neighborhood, Osman [1987] chooses a free pair (u, v) that is on a longest path in the current x-graph or y-graph, then replaces its current distance constraint by a new distance constraint.

This algorithm was applied to five problems, consisting of 6, 11, 15, 39, and 73 modules, and 12, 24, 31, 63, and 122 wires. The first two problems are adapted from Schalg, Liao & Wong [1983], the third problem is from Osman [1987], the fourth and fifth are adapted from Hsieh, Leong & Liu [1987]. Results were compared with those obtained by two other algorithms: an optimization algorithm, based on *branch-and-bound* techniques [Schalg, Liao & Wong, 1983], and an approximation algorithm based on *one-dimensional compaction*. For the second problem, whose initial layout is shown in Figure 12.5(a), the result of x-compaction followed by y-compaction is shown in Figure 12.5(b); y-compaction followed by x-compaction leads to the layout shown in Figure 12.5(c). The areas of the bounding rectangles of these layouts are 63.2% and 45.9% larger, respectively, than the area of the bounding rectangle of the optimally compacted layout shown in Figure 12.5(d). The performance of simulated annealing for each of the five problems is reported in Table 12.3. Column n gives the total number of layout components in each problem. The algorithms were implemented on a VAX 8650 computer.

Table 12.3 gives the average cost of the best solution $\bar{A}_{\text{best}}$, the average running time in seconds $\bar{t}$, and standard deviations σ_A and σ_t, respectively, obtained from five runs of the simulated annealing algorithm. The table also includes the results obtained by the branch-and-bound algorithm A_{opt} as well as its running time t, and the results obtained by applying two consecutive one-dimensional compactions: x followed by y, A_{x-y}, and y followed by x, A_{y-x}. No running times are mentioned for the approximation algorithm, since they are virtually negligible. For Problem 3 the branch-and-bound algorithm takes at least 3 days of CPU time, and 672 is the conjectured optimal value; Problems 4 and 5 are too large to be solved by the branch-and-bound algorithm.

From these results we conclude that, in the trade-off between CPU time and quality of final solutions, simulated annealing comes out very well. It was able to find high-quality solutions (as opposed to the one-dimensional approach) in

Table 12.3 Results of compaction by simulated annealing for five problems

		Simulated annealing				Branch-and-bound		1D compaction	
Problem	n	$\bar{A}_{\text{best}}$	σ_A	$\bar{t}$	σ_t	A_{opt}	t	A_{x-y}	A_{y-x}
1	18	114.0	0.0	4.8	3.7	114.0	0.2	126.0	142.5
2	35	3 557.8	0.0	164.9	19.8	3 557.8	643.0	5 808.0	5 192.3
3	46	685.2	10.8	36.4	4.6	672.0	> 3 days	1 015.0	1 200.0
4	102	1 736.0	0.0	100.4	16.4			2 016.0	2 067.0
5	195	3 372.0	24.0	440.2	158.4			4 032.0	4 187.0

reasonable amounts of CPU time (as opposed to the branch-and-bound approach). On the other hand, if one were to deal with instances consisting of say 500 modules and wires, one still would have to resort to one-dimensional compaction, because the running times for simulated annealing would be prohibitive.

For other compaction approaches based on simulated annealing, we refer the reader to Mosteller [1986] and Zeestraten [1985].

Simulated annealing has been applied to other layout problems. For applications to partitioning see Chatterjee & Hartley [1990], Chung & Rao [1986], Sugai & Hirata [1991], and Tao et al. [1992]. Wong, Leong & Liu [1986] proposed a PLA folding algorithm based on simulated annealing.

3 TABU SEARCH

Tabu search is another general local search procedure. Though some of the ideas in tabu search were proposed over a period of time, tabu search was introduced in its present form by Glover [1989, 1990] and has since been applied to solve combinatorial optimization problems arising in many different areas [Glover et al., 1993].

Unlike simulated annealing, which is a memoryless search method, tabu search accumulates information during the search process and stores the information in memory. This information is used either to encourage or discourage the generation of certain types of solutions. Like simulated annealing, tabu search may accept a solution inferior to the current one.

Though accepting inferior solutions is necessary for local search methods to jump out of local optima, it may lead to cycling if the search repeatedly returns to a local optimum previously visited. The most important feature of tabu search is to use information stored in memory to forbid the generation of certain types of solutions. The conditions that characterize the forbidden solutions are called *tabu conditions*. Yet a solution that violates one or more of the tabu conditions can be accepted if its aspiration value, computed according to an *aspiration function*, exceeds a certain value. Both the list of tabu conditions and the aspiration function are dynamically updated as the algorithm proceeds.

The main step in tabu search consists of generating the best solution in the neighborhood of the current solution subject to the tabu conditions (which can, however, be overridden according to the aspiration value of a solution), and replacing the current solution by the new solution. We now set up the general framework of tabu search. Let $\mathscr{S}$ denote the solution space and f the objective function of an optimization problem. Let $\mathscr{N}(i)$ denote the neighborhood of solution i. Let T denote the tabu list and A the aspiration function. Figure 12.6 shows the general framework of tabu search.

Application of tabu search to an optimization problem has four key components: neighborhood structure, tabu list, aspiration function, and stopping condition; see also Chapter 5.

```
begin TABU
  select an initial solution i∈𝒮;
  initialize tabu list T and aspiration function A;
  while the stopping condition is not met do
    compute the best i' in 𝒩(i)
      subject to tabu list T and aspiration function A;
    i ← i';
    update tabu list T;
    update aspiration function A;
  end while
end TABU
```

Figure 12.6 Tabu search paradigm

Tabu search has been applied to the solution of several layout problems. We will now discuss its application to two of them, placement and graph partitioning.

3.1 Partitioning

Let $G=(V, E)$ be a weighted graph with an even the number of vertices, where V is the set of modules and E is the set of nets. Let $w(u, v)$ denote the weight of edge $\{u, v\}$ in E. (As a convention, if $\{u, v\}$ is not in E, we let $w(u, v)=0$.) The bipartitioning problem asks for a division of V into two equal halves, U and W, so as to minimize the sum of the weights of the edges with one vertex in U and the other in W. This problem has various applications in design automation of VLSI circuits. Here we describe a tabu search approach to graph partitioning due to Tao et al. [1992].

The solution space consists of all equal-size bipartitions of V. This is,

$$\mathscr{S}=\{(U, W) \mid U \cup W=V, U \cap W=\varnothing, |U|=|W|\}.$$

For $i=(U, W)$, the value of the objective function is

$$f(i)=\sum_{u \in U,\, v \in W} w(u, v).$$

For $i=(U, W)$, $\mathscr{N}(i)$, the neighborhood of i is defined as follows. The vertices in U are sorted according to the linear order $\prec$ such that for $u_1, u_2 \in U$, $u_1 \prec u_2$ if

$$f((U-\{u_1\}, W \cup\{u_1\})) \leqslant f((U-\{u_2\}, W \cup\{u_2\})).$$

In other words, the decrement in the value of the cut when u_1 is moved from U to W is larger than or equal to the corresponding decrement when u_2 is moved from U to W. The vertices in W are sorted in a similar way.

Let $u_1, u_2, \ldots, u_K$ denote the K smallest vertices in U, according to the linear ordering $\prec$, and $v_1, v_2, \ldots, v_K$ the K smallest vertices in W. For $i=(U, W)$, we

define

$$\mathcal{N}(i) = \{(U', W') \mid U' = (U - \{u_p\}) \cup \{v_q\},\ W' = (W - \{v_q\}) \cup \{u_p\}, \text{ where } p, q \leqslant K\}.$$

In other words, (U', W') can be obtained from (U, W) by exchanging two vertices from the sets $\{u_1, u_2, \ldots, u_K\}$ and $\{v_1, v_2, \ldots, v_K\}$. Thus, the number of neighboring solutions of i is K^2.

The tabu list is simply a record of the r most recent exchanges for a positive integer r. To be specific, if $u_1 \in U$ and $v_1 \in W$ are selected to be exchanged to yield a new solution, the pair (v_1, u_1) will be added to the tabu list unless it is already there. (That u_1 and v_1 are selected when (v_1, u_1) is in the tabu list is possible because a tabu condition can be overridden by the aspiration function.) In the case that (v_1, u_1) is to be added to the tabu list, the rth exchange will be removed if there is one.

The aspiration function A is defined to be a function from $\mathbb{R}$ to $\mathbb{R}$. Specifically, $A(c)$ is one less than the cost of the best solution that has been reached from a solution of cost c. Intuitively, it means the tabu conditions will be overruled if a new solution has a cost lower than any other solution that has been reached from a solution with the current cost. Consequently, in each iteration, the aspiration function is updated as follows:

if $A(f(i')) > f(i)$ **then** $A(f(i')) \leftarrow f(i) - 1$
else
if $A(f(i)) > f(i')$ **then** $A(f(i)) \leftarrow f(i') - 1$.

The algorithm will stop if no improvement is observed for a certain number of iterations.

We now turn to multiway partitioning, which is the problem of partitioning the vertices of a graph into several mutually exclusive subsets so that the sum of the weights of the edges crossing the subsets is minimized under the condition that the vertex weights are distributed evenly among the subsets. Let $G = (V, E)$. To each vertex $v \in V$, a vertex weight $d(v)$ is assigned. To each edge $\{u, v\} \in E$, an edge weight $w(u, v)$ is assigned. Let $\mathcal{S}$ denote the set of all feasible solutions. For $i \in \mathcal{S}$, $i = (V_1, V_2, \ldots, V_m)$ where $V_1 \cup V_2 \cup \cdots \cup V_m = V$ such that $V_p \cap V_q = \varnothing$ for $p \neq q$.

The cost of i, $f(i)$, is the sum of the $w(u, v)$ such that $u \in V_p$ and $v \in V_q$ for $p \neq q$. We want to search for a solution which minimizes the value of $f(i)$ subject to the constraint that

$$\sum_{1 \leqslant p < q \leqslant m} \left(\sum_{u \in V_p} d(u) - \sum_{v \in V_q} d(v) \right)^2 \tag{3}$$

is minimum. We now transform the problem so that the weights on vertices in this constraint are removed. Given $G = (V, E)$, we construct a complete graph $G^* = (V, E^*)$ such that there is an edge $\{u, v\}$ between every two vertices u and v in

V. For each edge $\{u, v\}$ in E^*, its edge weight $w'(u, v)$ is defined as follows:

$$w'(u, v) = \begin{cases} -d(u)\,d(v)R + w(u, v) & \text{if } (u, v) \in E, \\ -d(u)\,d(v)R & \text{if } (u, v) \notin E, \end{cases}$$

where R is a positive real constant that is larger than the sum of the edge weights of all the edges in E. For a partition i of G^* (a partition of G as well), its cost, the sum of the weights of the edges crossing the subsets, is

$$f'(i) = -R \sum_{1 \leq p < q \leq m} \left(\sum_{u \in V_p} d(u) \right) \left(\sum_{v \in V_q} d(v) \right) + f(i).$$

It can be shown that minimizing $f'(i)$ is equivalent to minimizing $f(i)$ under the constraint that (3) is minimum. Namely, solving the transformed problem is the same as solving the original problem. Thus, f' is chosen to be the cost function.

The neighborhood of solution i consists of two parts, $S_{1,i}$ and $S_{2,i}$, which are defined as follows. Let $i = (V_1, V_2, \ldots, V_m)$. We define

$$S_{1,i} = \{(V_1, V_2, \ldots, V_t - \{u\}, \ldots, V_j \cup \{u\}, \ldots, V_m) \mid u \in V_t\}.$$

In other words, a solution is in $S_{1,i}$ if it can be obtained by moving a vertex from one subset to another subset.

Let u be a vertex in V_t. Suppose that moving u from V_t to any one of $V_1, V_2, \ldots, V_{t-1}, V_{t+1}, \ldots, V_j, \ldots, V_m$ will yield a solution of cost larger than $f(i)$. Let V_j be the subset into which u is moved that will lead to the smallest increase in the cost. We define

$$S_{2,i} = \{(V_1, V_2, \ldots, V_t - \{u\} \cup \{v\}, \ldots, V_j - \{v\} \cup \{u\}, \ldots, V_m) \mid u \in V_t, v \in V_j\}.$$

In other words, we shall only exchange u with every vertex v in V_j but not with other vertices in other subsets. $\mathcal{N}(i)$ is simply the union of $S_{1,i}$ and $S_{2,i}$.

The tabu list, aspiration function, and stopping condition are all the same as for bipartitioning.

For other partitioning algorithms based on tabu search, see Areibi & Vannelli [1993] and Lim & Chee [1991].

3.2 Placement

Tabu search has also been applied to placement [Lim, Chee & Wu, 1991; Song & Vannelli, 1992]. Here we briefly describe the placement algorithm proposed by Song & Vannelli [1992]. This algorithm is applicable to both sea-of-gates and gate array design styles.

An estimate of the total wire length is used as the cost of a placement. The wire length of a net is estimated by the half-perimeter of the bounding box of the terminals of the net in the placement. The estimate of the total wire length is simply the sum of the estimated wire lengths of all the nets.

Let i be a placement and u be a module. We define $N_i(\delta, u)$ to be the set of modules whose x- and y-coordinates are within distance δ from that of u, i.e.,

$$N_i(\delta, u) = \{ v \text{ is a module } | |x_i(u) - x_i(v)| \leqslant \delta \text{ and } |y_i(u) - y_i(v)| \leqslant \delta \},$$

where $x_i(u)$ and $y_i(u)$ are the respective x- and y-coordinates of module u in placement i. To define the neighboring placements of i, a module is selected randomly. Suppose that module u is selected. The neighborhood of i, $\mathcal{N}(i)$, is then defined to be the set of placements that can be obtained from i by exchanging module u with any one of the modules in $N_i(\delta, u)$.

The tabu list is the list of the r most recent module exchanges for some control parameter r which is a positive integer. The exchanges in the tabu list are ordered from the oldest to the most recent. The current exchange will be added to the tabu list if it is not there. The addition of the current exchange will cause the rth exchange (the oldest) in the list to be dropped. It was observed that the aspiration function has almost no influence on the behavior of the algorithm, so it was actually not used.

The stopping condition is as follows. An upper limit is set on the total number of iterations between improvements. If the limit is reached without encountering an improvement after the previous improvement, the algorithm stops.

By using a constructive placement algorithm to generate the initial placement, the results were comparable to those obtained using TimberWolfSC5.4, a placement package based on simulated annealing, but the running time was considerably shorter.

4 GENETIC ALGORITHMS

Genetic algorithms have their origin in biological processes. Species in the natural world undergo a process of slow evolution; they gradually adapt themselves to the environment through mutation and mixing of genes. Good characteristics gradually emerge and bad characteristics gradually disappear.

A genetic algorithm maintains a *population* of solutions to the problem at hand and allows them to evolve through successive generations. Crossover and mutation are used to create the next generation from the present generation. A *crossover* operation combines two (or sometimes more) solutions to yield a new solution; a *mutation* operation perturbs an individual solution. The solutions to be included in the next generation are then selected, according to their *fitness* values, from the set of solutions comprising of the current generation and the newly generated solutions. Crossover and the selection criteria form the main optimization step in genetic algorithms. The assumption here is that good partial solutions (genes) to a problem are combined during crossover to form better solutions. Mutation is the main operation for avoiding local optima. A good tutorial on genetic algorithms is Goldberg [1989].

A constant number of solutions with the highest fitness values are typically selected so that the population is maintained at a fixed size for each generation.

```
begin GEN
  set up the first generation;
  for (t ← 1 to max_of_generation) do
      select a mating pool from the current generation;
      crossover solutions in the mating pool to generate
      a children pool;
      mutate the solutions in the children pool and current
      generation;
      select solutions to form the new current generation;
  end for
end GEN
```

Figure 12.7 Genetic algorithm paradigm

The process is terminated after a certain number of generations. Either the best remaining solution or the best solution ever observed is reported. Figure 12.7 is a high-level description of a genetic algorithm; see also Chapter 6.

Genetic algorithms have been applied to many layout problems; we present two examples: floorplanning and improvement of layouts.

4.1 Floorplanning

We present a genetic algorithm for floorplan design due to Cohoon et al. [1991]. The solution space consists of all slicing floorplans with n modules. As in the simulated annealing algorithm of Wong & Liu [1987] (Section 2.2), Polish expressions are used to represent the solutions. Four crossover operations are introduced. They are defined as follows.

- CO_1: One child is produced. The child is constructed by carrying out a one-for-one replacement of the operators in parent 1 by the operators in parents 2 in the order they appear in parent 2. Figure 12.8 shows an example.
- CO_2: One child is produced. The child is constructed by carrying out a one-for-one replacement of the operands in parent 1 by the operands in parent 2 in the order they appear in parent 2. Figure 12.9 shows an example.
- CO_3: One child is produced. A subtree in parent 1 is first identified. The child is constructed by reordering the operands in parent 1 that are not in the subtree; the reordering is made according to the order of the operands in parent 2. Figure 12.10 shows an example.
- CO_4: Two children are produced. Two subtrees of the same size are identified in parent 1 and parent 2, respectively. Child 1 is constructed as follows. The subtree in parent 1 is replaced by the subtree in parent 2. The remaining operators of parent 1 are retained in their original positions. The operands in parent 1 that do not appear in the subtree from parent 2 will retain their order and occupy the positions for operands in parent 1 that are not occupied by

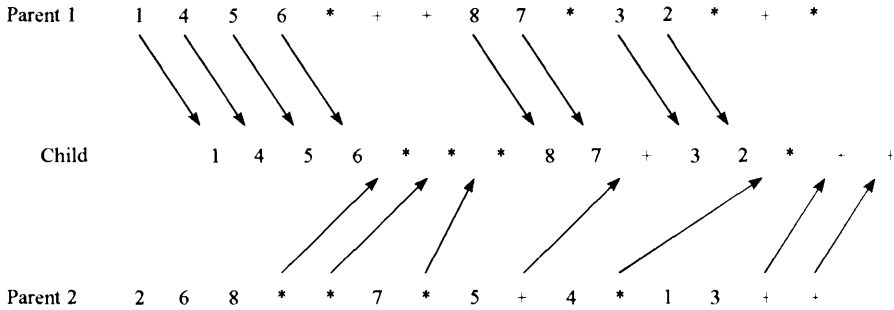

Figure 12.8 Illustration of crossover operator CO_1

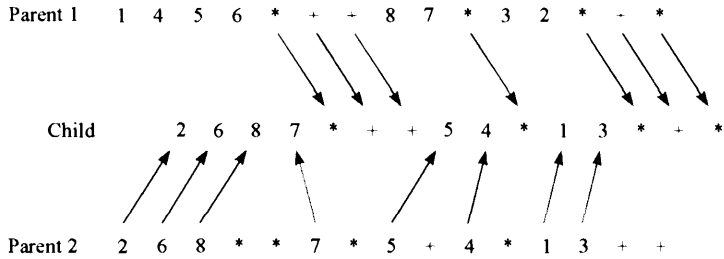

Figure 12.9 Illustrations of crossover operator CO_2

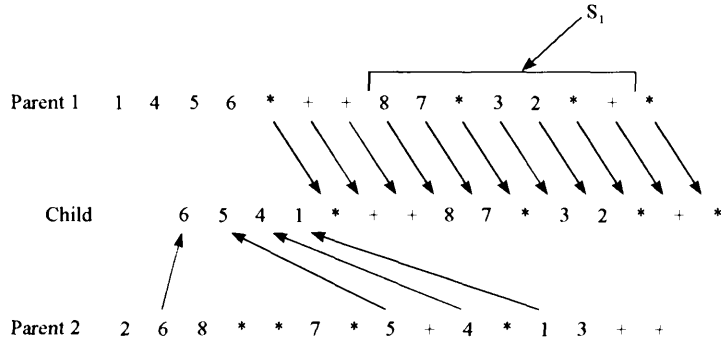

Figure 12.10 Illustration of crossover operator CO_3

operands in the subtree from parent 2. Child 2 is constructed in the same way except that the roles of parent 1 and parent 2 are interchanged. Figure 12.11 shows an example.

The mutation operators take a single individual and modify it in a localized manner. The operations defined by Wong & Liu [1987] in their simulated

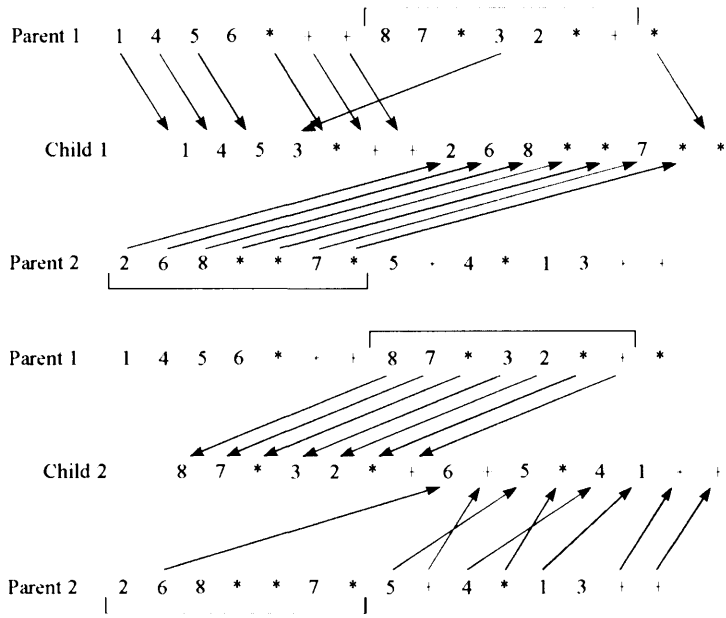

Figure 12.11 Illustration of crossover operator CO_4

annealing algorithm are chosen as the mutation operators, namely, M_1: swapping two adjacent operands, M_2: switching a sequence of adjacent operators, and M_3: swapping an operator and a neighboring operand.

A score is given for every solution in the current population. The score of a solution is the cost function defined in the simulated annealing floorplanning algorithm by Wong & Liu [1987]. Let μ denote the mean and σ denote the standard deviation of the distribution of the scores of the solutions in the current population. For each solution i in the population, its fitness value is determined according to the formula

$$fitness(i) = \frac{\mu - score(i) + \alpha\sigma}{2\alpha\sigma}$$

where α is a constant parameter. Since we shall keep the population size constant, the solutions with the n highest fitness values will survive. Experiments with the algorithm showed very encouraging results.

4.2 Global improvement of macro cell layouts

In most macro cell floorplanning systems the shapes (widths and heights) of the modules and the total wire length are used as control parameters, e.g. the algorithm of Wong & Liu [1987]. Since no actual routing is carried out during the floorplanning stage, total wire length is often estimated very roughly.

The area of a layout consists of two parts: the area occupied by the modules and the area needed for routing. Given a floorplan for a set of modules with fixed shapes, a placement can be obtained by placing the modules according to the floorplan and leaving enough routing area between every pair of neighboring modules. The routing area cannot be determined precisely without actually completing the routing. However, given the pin positions on the boundary of each module and the global routes, channel density gives a fairly good estimation of the channel width. The density of a channel can be changed by laterally shifting the modules on the boundaries of the channel [Tammassia & Tollis, 1988; Lengauer, 1990]. We now present a genetic algorithm for improving a layout generated by a floorplanning system due to Glasmacher, Hess & Zimmermann [1991].

The problem can be formally defined as follows. Given a set of rectangular modules $\{m_1, \ldots m_n\}$ with fixed width and height and fixed pin positions, a floorplan for the modules, and a global routing, determine a placement of the modules with minimum total area for a given aspect ratio such that the separation between any two neighboring modules is larger than or equal to the density of the channel formed by the two modules. We further assume that the floorplans are slicing. Figure 12.12 shows an example of the problem with six modules and the corresponding slicing floorplan.

A solution to the global improvement problem consists of one gene for each cutline and the genes are arranged in a fixed linear order as a table. A gene specifies which two modules on opposite sides of the cutline are chosen as well as the offset between the two modules in the corresponding placement. The offset of two modules may be measured by, say, the difference between the coordinates of

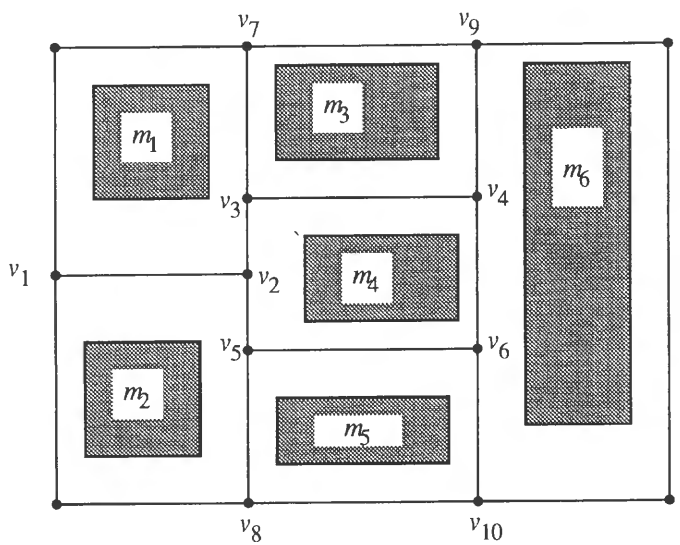

Figure 12.12 An example of the problem with six modules

gene	1	2	3	4	5
cutline	(v_1, v_2)	(v_3, v_4)	(v_5, v_6)	(v_7, v_8)	(v_9, v_{10})
module pair	m_1, m_2	m_3, m_4	m_4, m_5	m_1, m_3	m_3, m_6
offset	2	1	2	1	1

Figure 12.13 A solution to the example in Figure 12.12

the lower left corners of the two modules in the direction of the cutline. Figure 12.13 shows a solution to the problem in Figure 12.12. Gene 1 corresponds to the horizontal cutline (v_1, v_2). The only pair of modules on opposite sides of (v_1, v_2) is m_1 and m_2, so this pair is chosen. The difference in the horizontal coordinates of the lower left corners of m_1 and m_2 is 2 units as specified by the offset value in column 1. For cutline (v_7, v_8) (gene 4), on the left side either m_1 or m_2 can be chosen; on the right side either m_3, m_4, or m_5 can be chosen. For the solution in Figure 12.13 the chosen pair is m_1 and m_3. Since (v_7, v_8) is vertical, the difference in the vertical coordinates of the lower left corners of m_1 and m_3 is 1 unit, as specified by the offset value in column 5.

Given a solution, we now describe how to determine the corresponding placement. The algorithm works bottom-up in the slicing tree. A cutline is processed only after its two children have been processed. To process a cutline, shift the two partial placements on opposite sides of the cutline so that the two chosen modules on opposite sides have the offset specified by the solution. The offsets between any other pair of neighboring modules along the cutline are then determined as well. Next the minimum separation requirement between any pair of neighboring modules along the cutline is determined by computing the density of the channel formed by the two modules. The two partial placements are now moved in the direction perpendicular to the cutline until the minimum separation between any two modules on opposite sides of the cutline is satisified. The final placement is generated after the root of the slicing tree is processed.

The fitness value of a solution i is a combination of three measures:

- the aspect ratio $AR(i)$ of the enclosing rectangle;
- the area $AP(i)$ of the polygon enclosing all the modules, excluding the routing area on the boundary of the layout;
- the area $AB(i)$ of the bounding box.

Let i be a solution and ARg be the given aspect ratio. The following formula is used to compute the fitness value of a solution:

$$fitness(i) = x(i)\left(\frac{A}{AP(i)} + \frac{A}{AB(i)}\right),$$

Parent 1

gene	1	2	3	4	5
cutline	(v_1, v_2)	(v_3, v_4)	(v_5, v_6)	(v_7, v_8)	(v_9, v_{10})
module pair	m_1, m_2	m_3, m_4	m_4, m_5	m_1, m_3	m_3, m_6
offset	2	1	2	1	1

Parent 2

gene	1	2	3	4	5
cutline	(v_1, v_2)	(v_3, v_4)	(v_5, v_6)	(v_7, v_8)	(v_9, v_{10})
module pair	m_1, m_2	m_3, m_4	m_4, m_5	m_2, m_5	m_4, m_6
offset	2	1	1	1	2

Child 1

gene	1	2	3	4	5
cutline	(v_1, v_2)	(v_3, v_4)	(v_5, v_6)	(v_7, v_8)	(v_9, v_{10})
module pair	m_1, m_2	m_3, m_4	m_4, m_5	m_1, m_3	m_4, m_6
offset	2	1	2	1	2

Child 2

gene	1	2	3	4	5
cutline	(v_1, v_2)	(v_3, v_4)	(v_5, v_6)	(v_7, v_8)	(v_9, v_{10})
module pair	m_1, m_2	m_3, m_4	m_4, m_5	m_2, m_5	m_3, m_6
offset	2	1	1	1	1

Figure 12.14 Illustration of the crossover operation

where A is the total area of the modules and

$$x(i) = \begin{cases} ARg/AR(i) & \text{if } ARg < AR(i), \\ AR(i)/ARg & \text{otherwise.} \end{cases}$$

To cross two solutions, a cutline is selected randomly. Two children (solutions) are generated by concatenating the first segment of one solution and the second segment of the other, and vice versa. Figure 12.14 shows an example of the crossover operation. In this example, cut line (v_9, v_{10}) (gene 5) is selected to cross the two parents. Child 1 is formed by concatenating genes 1 to 4 of parent 1 and gene 5 of parent 2; similarly child 2 is formed by concatenating genes 1 to 4 of parent 2 and gene 5 of parent 1.

Mutation of a solution is performed by modifying the genes. Whether a gene will be modified or not is determined by a given probability. A gene is modified by randomly selecting two modules on opposite sides of the cutline together with a random offset for the two modules.

The initial population of solutions is generated by randomly selecting two adjacent modules on opposite sides of each cutline, and an offset is assigned to them randomly. The mating pool is set up by selecting the solutions from the current generation of solutions at random, favoring those with high fitness values. Note that a solution may be selected several times. The size of the mating pool is chosen to be the number of current solutions. To cross the solutions in the mating pool, two solutions are selected at random and removed from the pool. Crossover will take place with a given probability and the two resulting solutions are added to the child pool. If crossover did not take place, the two solutions themselves will be added to the child pool. The next generation is produced by choosing 25% of the fittest solutions in the current generation and another 25% of the remaining solutions randomly. And 25% of the fittest solutions in the mutated child pool are chosen together with another 25% randomly chosen solutions among the remaining ones. Therefore, the number of solutions in each generation is held fixed. An upper bound on the number of generations is set. The final placement is the one corresponding to the solution with the best fitness value in the last generation. Good experimental results were observed.

There are many other results on applying genetic algorithms to the solution of layout problems. Here we list some of them. For partitioning see Chandrasekharam, Subhramanian & Chaudhury [1993], Jin & Chan [1992], and Saab & Rao [1989]. For floorplanning/placement see Chan, Mazumder & Shahookar [1991], Cohoon & Paris [1987], Esbensen [1992], Glasmacher & Zimmermann [1992], Kling & Banerjee [1987, 1991], Mohan & Mazumder [1992], and Shahookar & Mazumder [1990]. For routing see Geraci et al. [1990], Lin, Hsu & Tsai [1989], and Liu, Sakamoto & Shimamoto [1993]. Fourman [1985] proposed a genetic algorithm for compaction. Hill [1993] applied genetic algorithms to the gate sizing problem.

Local Search in Combinatorial Optimization
Edited by E. Aarts and J. K. Lenstra
© 1997 John Wiley & Sons Ltd.

13

Code design

Iiro S. Honkala
University of Turku, Turku

Patric R. J. Östergård
Helsinki University of Technology, Espoo

1 INTRODUCTION 441
2 ERROR-CORRECTING CODES 442
2.1 Basic definitions 442
2.2 Applying local search methods 443
3 COVERING CODES 446
3.1 Basic definitions 446
3.2 Applying local search methods 447
3.3 A matrix method 448
3.4 The football pool problem 449
3.5 Multiple coverings and other problems 450
4 SOURCE CODES 452
5 LOW AUTOCORRELATION BINARY SEQUENCES 454
ACKNOWLEDGMENTS 456

1 INTRODUCTION

In information theory we wish to find fast and reliable methods for transmitting information. We may first wish to encode the information using a suitable *source code* – to convert the information from analog to digital – or to compress the data. To achieve reliable transmission over a noisy channel, the information is then encoded using a suitable *channel code*, i.e., an *error-correcting code*. Algebraic and combinatorial methods often provide good codes [e.g., Van Lint, 1982; MacWilliams & Sloane, 1977], codes having a rich structure that makes encoding and decoding easy. However, we usually obtain good codes only for certain code parameters. Therefore local search techniques, robust by nature and uniform in performance, constitute another valuable method for constructing good codes. On the other hand, using local search we often, but not always, find codes whose structure is more random; this may restrict their use in practice.

In return, these information theoretical problems provide us with good test benches for local search techniques, because there are extensive numerical tables of the best known codes for many of these problems.

Section 2 presents some basic concepts in coding theory and study how error-correcting codes, particularly constant weight codes, can be constructed using local search. Section 3 is devoted to a dual problem, the construction of covering codes. The covering radius of codes has been extensively studied in recent years. We study several different types of codes with good covering properties, e.g., multiple coverings and multiple coverings of the farthest-off points, and discuss a special case known as the football pool problem. Section 4 discusses source codes and vector quantization, and Section 5 looks at low-autocorrelation binary sequences.

2 ERROR-CORRECTING CODES

2.1 Basic definitions

Consider words (or vectors) of length n over a given q-element alphabet F. Usually we take as our alphabet F the set $\mathbb{Z}_q = \{0, 1, \ldots, q-1\}$ of integers modulo q. If q is a prime power, it is often convenient to consider the finite field $\mathbf{F}_q$ of q elements instead in order to be able to use results from linear algebra. The (Hamming) distance $d_H(x, y)$ between any two words $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$ of length n is defined to be the number of indices i such that $x_i \neq y_i$. The weight $wt(x)$ is the number of nonzero coordinates in x.

A q-ary *code* of length n is simply a nonempty subset of F^n. The minimum distance $d(C)$ of a code $C \subseteq F^n$ is the smallest of the pairwise distances between the codewords of C. The code C is *linear* if $F = \mathbf{F}_q$ and C is a linear subspace of F^n.

The set

$$B_r(x) = \{y \in F^n \mid d_H(x, y) \leqslant r\}$$

is called the *Hamming sphere* of radius r centered at x.

Assume that the code $C \subseteq \mathbb{Z}_q^n$ has minimum distance $2e + 1$. Suppose that one of the codewords $c \in C$ is transmitted over a noisy channel and that during the transmission at most e errors occur, i.e., at most e coordinates of the vector c are changed. If the receiver knows that one of the codewords was transmitted and always decodes the received word to the nearest codeword, the codeword c that was originally sent can actually be retrieved, simply because the Hamming spheres of radius e centered at the codewords are disjoint. Therefore the code C is called *e-error-correcting*. To be able to send as much information as possible in one time unit, one would like the cardinality of C to be as large as possible. Determining the values of the functions

$$A_q(n, d) = \text{the largest possible cardinality of a } q\text{-ary code of length } n \text{ and minimum distance at least } d$$

and

$$\alpha_q(\delta) = \limsup_{n \to \infty} n^{-1} \log_q A_q(n, \delta n)$$

are central problems in combinatorial coding theory. In general, the exact values

are not known, and we only have lower and upper bounds; see Van Lint [1982, Chapter 5] and MacWilliams & Sloane [1977, Chapter 17]. If $q = 2$ we usually use the notation $A(n, d)$ instead of $A_2(n, d)$.

If all the codewords in a code have the same weight, the code is called a *constant weight code*. A very important coding theoretical function is

$$A(n, d, w) = \text{the maximum possible cardinality of a binary code } C \subseteq \mathbb{Z}_2^n \text{ with constant weight } w \text{ and minimum distance at least } d.$$

This function is very important in estimating $A(n, d)$. And if we ask what is the cardinality of a largest possible set of w-element subsets of a given n-element set $\mathscr{S}$ such that any t-element subset of $\mathscr{S}$ is contained in at most one of these w-element subsets, it is easy to check that the answer is $A(n, 2(w - t + 1), w)$. For the most recent tables of lower bounds on $A(n, d)$ and $A(n, d, w)$, see Brouwer et al. [1990].

2.2 Applying local search methods

Lower bounds on $A_q(n, d)$ and $A(n, d, w)$ are improved by explicitly constructing error-correcting codes. It is interesting to see that, since El Gamal et al. [1987] presented the first results on using local search methods for finding error-correcting codes, most of the methods covered in this book have been applied to the problem. The literature mainly covers the problem of finding codes that improve the best known lower bounds on $A(n, d, w)$. One reason for this is that for binary error-correcting codes of length at most 16, all exact values of $A(n, d)$ are known, except $72 \leqslant A(10, 3) = A(11, 4) \leqslant 76$ and $144 \leqslant A(11, 3) = A(12, 4) \leqslant 152$ [Klein, Litsyn & Vardy, 1995; Litsyn & Vardy, 1994].

We now consider the problem of finding constant weight error-correcting codes. With slight modifications, the same approach can be used to obtain lower bounds on $A_q(n, d)$. Before the search for a code C, we fix the following parameters: the length of the code n, the weight w, the minimum distance d, and the number of codewords M. If the desired code is found, M is increased and the search procedure is repeated. The problem-specific parameters are usually chosen as follows:

- *Solution.* Any M binary words $c_1, c_2, \ldots, c_M$ of length n and weight w.
- *Neighborhood structure.* The neighborhood consists of all solutions that are obtained by replacing any one word c in the solution with another word c' of weight w with $d_H(c, c') = 2$.
- *Cost function.* The cost function is defined as

$$f(C) = \sum_{1 \leqslant i < j \leqslant M,\, d_H(c_i, c_j) < d} g(d_H(c_i, c_j)).$$

Several functions g have been proposed:

- Let $g(x) = (d - x)^k$; Koschnick [1991] uses $k = 1$ and Dueck & Scheuer [1990] use $k = 2$.

- Let $g(x) = x^{-k}$; used by El Gamal et al. [1987], where the summation in $f(C)$ is taken over all $1 \leqslant i < j \leqslant M$; often $k = 2$.
- Let $g(x) = 1/(x^2 + 1) - 1/(d^2 + 1)$; used by Comellas & Roca [1993].

We now take a look at local search methods that have been applied to this problem. Unfortunately, no thorough comparison of these methods has been published. Undoubtedly, it is not easy to compare the methods objectively, since it is often not even clear how to use one given method in the best possible way to find a certain code.

The seminal paper by El Gamal et al. [1987] uses *simulated annealing*. They manage to improve more than 10 bounds by finding corresponding constant weight codes. All the bounds found have been improved since the paper appeared, but its importance should not be underestimated; El Gamal et al. clearly demonstrate the potential of local search methods.

Dueck & Scheuer [1990] use *threshold accepting* to find constant weight codes. The performance of their algorithm is compared against the results of El Gamal et al. [1987]. For most instances they find better codes by using threshold accepting. Nevertheless, it is difficult to evaluate their comparison, as nothing is said about computation times. They also try to speed up the search by imposing the following structure on a code when $n = 2w$: if c is a codeword, then its complement $\bar{c}$ is also a codeword. In this way they managed to find one new bound (which has now been improved).

Koschnick [1991] presents an optimization algorithm of his own, which has the following neighborhood-generating mechanism:

- *Neighborhood structure.* We make a list of and go through all codewords that are at distance less than d from some other codeword, one at a time in a random order. The neighborhood with respect to such a codeword consists of codes where the codeword considered is replaced by another word with the same weight at distance 2 apart.

This part of the algorithm follows the steepest descent local search heuristic, i.e., a solution with least cost in the neighborhood is accepted if that cost is not higher than the cost of the previous solution. When a new solution has been accepted, a new list is constructed and the procedure is repeated. One method that starts in the same way is tabu search. However, in the continuation, Koschnick's algorithm has its own methods to escape local minima and start a new descent; see the paper for further details. Four of the bounds in the paper are still the best known [Brouwer et al., 1990].

Genetic algorithms are used by Comellas & Roca [1993], and by Vaessens, Aarts & Van Lint [1993], who calculate bounds for $A_3(n, d)$. In Comellas & Roca [1993] the results are comparable to those of El Gamal et al. [1987].Vaessens, Aarts & Van Lint [1993] obtain many new ternary codes. They do not keep the size of the code (M) constant, but start with a small code that has minimum distance d, trying to add codewords whenever this is possible without violating the minimum distance criterion; neither is it violated by recombination.

The principle of always keeping the minimum distance at least d is also adopted in the paper by Brouwer et al. [1990], where several powerful methods are used that lead to a vast number of best known bounds. One of their algorithms is as follows:

1. Start with a code with minimum distance d.
2. Perform a permutation on the coordinates of the code.
3. Perturb every codeword by 'pushing' it until it is lexicographically as small as possible.
4. Sort the (permuted and pushed) codewords into lexicographic order.
5. Remove the (lexicographically) last k words from the code, and attempt to replace them by more than k words, using some exhaustive or heuristic search method.
6. Go back to Step 2 (and repeat as many times as desired).

Here we can use in Step 5 one of the local search methods we have already discussed. A code is said to be k-optimal if we cannot first delete k codewords and then add more than k codewords while maintaining the minimum distance d. The codes in Brouwer et al. [1990] are k-optimal for values of k in the range 2–5. They discuss other heuristics besides this one.

We would like to suggest another natural choice for the cost function. We now consider $A_q(n,d)$ with $d = 2e + 1$. For a code $C \subseteq \mathbb{Z}_q^n$, let $c(x) = |B_e(x) \cap C|$. Then we have a packing if $c(x) \leqslant 1$ for all $x \in \mathbb{Z}_q^n$. A possible cost function is then $f(C) = -M|B_e(y)| + \sum_{x \in \mathbb{Z}_q^n} c(x)^k$ for $k > 1$. Alternatively, we know that we have a packing when $c(x) = 0$ for exactly $q^n - M|B_e(y)|$ words $x \in \mathbb{Z}_q^n$. This gives the following possible cost function:

- *Cost function.* The cost function is defined as

$$f(C) = M|B_e(y)| - q^n + |\{x \in \mathbb{Z}_q^n | c(x) = 0\}|.$$

The first part of the cost function is included so that the cost for a packing is 0. As shown in the next section, a computer program that finds covering codes can be used to find packings in this way. This approach is computationally more efficient than the approach considered earlier when M is large and $|B_e(x)|$ is small, i.e., d is small. It also works for constant weight codes.

Example 1. We made attempts to improve lower bounds on $A(n,4,w)$ tabulated by Brouwer et al. [1990]. No improvements could be made, but in the following cases we were able to find a code with one more codeword than the present record and the distances between codewords at least 3 (and hence at least 4), except for *one pair* of words at distance 2 from each other: $A(12,4,5) \geqslant 80$, $A(13,4,5) \geqslant 123$, $A(14,4,6) \geqslant 278$, and $A(14,4,7) \geqslant 325$.

For a partitioning construction that gives good constant weight codes with minimum distance 4, see Brouwer et al. [1990] and their references. In this construction, a code C consisting of all vectors of length n and weight w is partitioned into codes with minimum distance 4. Such partitions can then be

combined to obtain good codes with minimum distance 4. Local search has also been used to find good partitions of this kind; see Brouwer et al. [1990].

Said & Palazzo [1993] use *tabu search* to find good *unit-memory convolutional codes* and *linear* error-correcting codes. The codewords in a linear code with 2^k codewords and length n are linear combinations of the rows of a $k \times n$ generator matrix G. The authors fix the parameters n, k, and d of the desired code and randomly select the n initial columns of G. These are stored in a vector $c = (c_1, c_2, \ldots, c_{2^k-1})$ such that c_i is the number of columns in G whose decimal representation is i (all-zero columns are not allowed). A vector $w = (w_1, w_2, \ldots, w_{2^k-1})$ consisting of the weights of all codewords (except the all-zero word, which is always contained in a linear code) can then be obtained from c by matrix multiplication. The minimum distance of a linear code is the minimum weight of any nonzero codeword. If $w_i \geqslant d$ for all i a code is found.

The following parameters are used in the search:

- *Solution.* Any vector c such that $\sum_{i=1}^{2^k-1} c_i = n$, $c_i \geqslant 0$.
- *Neighborhood structure.* All solutions c' such that for some a and b, $c'_a - c_a = 1$, $c_b - c'_b = 1$, and $c_i = c'_i$ when $i \neq a, b$.
- *Cost function.* One possible cost function is

$$f(c) = \sum_{1 \leqslant i \leqslant 2^k-1,\, w_i < d} g(w_i),$$

  where any function g as defined in the beginning of this section can be used. Said & Palazzo [1993] use the following function with $k \geqslant 2$; at the end of the search it must be checked that $w_i \geqslant d$ for all i:

$$f(c) = \sum_{1 \leqslant i \leqslant 2^k-1,\, w_i \leqslant d} (d + 1 - w_i)^k.$$

Two tabu lists 'in' (I) and 'out' (O) are used; the indices a and b, used above, are added to I and O, respectively, when the corresponding neighbor is accepted.

For applications of local search to decoding, see Zhang [1992]. El Gamal et al. [1987] discuss the use of simulated annealing in constructing *spherical codes*. They consider the problem of maximizing the minimum Euclidean distance between k points on an n-dimensional unit sphere. Such codes are also considered by Nurmela [1995], who found several new codes using methods involving simulated annealing and tabu search. For example, he gives three new three-dimensional codes; these record-breaking codes have 43, 77, and 80 points.

3 COVERING CODES

3.1 Basic definitions

The problem discussed in the previous section can be viewed as a *packing* problem: we wish to pack as many nonintersecting Hamming spheres of a given radius as possible in $\mathbb{Z}_q^n$. Its dual is the *covering* problem: to find the smallest number of Hamming spheres of a given radius needed to cover $\mathbb{Z}_q^n$. So, one of

the main objects of study in this section is the function

$$K_q(n, R) = \text{the smallest possible cardinality of a } q\text{-ary code of length } n \text{ and covering radius at most } R,$$

where the *covering radius* is the smallest integer R such that every word in $\mathbb{Z}_q^n$ is within Hamming distance R from at least one codeword of C, i.e., the smallest integer R such that the Hamming spheres of radius R cover the whole space. If $q = 2$ we use the notation $K(n, R)$ instead of $K_2(n, R)$. When R is clear from the context, we say that a word x is covered by a codeword c or a code C, if $d_H(x, c) \leqslant R$ or $d_H(x, C) = \min_{c \in C} d_H(x, c) \leqslant R$, respectively.

As with the packing problem, we usually do not know the exact value of $K_q(n, R)$ but only lower and upper bounds. The upper bounds are constructive, and the lower bounds came from nonexistence proofs. Covering radius problems of codes have been discussed in a large number of papers since Taussky & Todd [1948], e.g., Brualdi, Litsyn & Pless [1998], Cohen et al. [1985], Cohen, Lobstein & Sloane [1986], Graham & Sloane [1985], Hämäläinen & Rankinen [1991], Van Lint [1989], and their references. For applications of this problem, see Cohen et al. [1985].

It is known that asymptotically

$$\lim_{n \to \infty} n^{-1} \log_q K_q(n, \rho n) = 1 - H_q(\rho)$$

for $\rho \in [0, (q-1)/q]$, where $H_q(x) = x \log_q(q-1) - x \log_q x - (1-x) \log_q(1-x)$ is the q-ary entropy function ($H_q(0) = 0$); see Cohen & Frankl [1985], Delsarte & Piret [1986], and their references.

3.2 Applying local search methods

Local search has been used in many papers to construct good covering codes. The method utilized is almost exclusively *simulated annealing*. The objective is to find a q-ary code C of length n with M codewords and covering radius at most R (these parameters are fixed before the search). In the simulated annealing algorithm, the following choices of problem-specific parameters have turned out to give a good and robust performance:

- *Solution.* Any M q-ary words $c_1, c_2, \ldots, c_M$ of length n.
- *Neighborhood structure.* The neighborhood is generated by changing i coordinates, $1 \leqslant i \leqslant R$, in one word in the solution. Alternatively, instead of considering all words in the solution when the neighborhood is determined, we may go through the words in the solution one by one in some fixed order.
- *Cost function.* The cost function is defined to be the number of words in $\mathbb{Z}_q^n$ that are not covered by the words in the solution, that is, $f(C) = |\{x \in \mathbb{Z}_q^n \mid d_H(x, C) > R\}|$.

If we find a solution C such that $f(C) = 0$, then M is decreased and the search procedure is repeated. This version is used by Wille [1987] to show that

$K_3(6, 1) \leqslant 74$, by Van Laarhoven et al. [1989] to prove that $K_3(6, 1) \leqslant 73$ and $K_3(7, 1) \leqslant 186$, by Östergard [1991a, 1991b, 1994a] and Wille [1990] to obtain a number of other record-holding codes. The bound $K_3(6, 1) \leqslant 73$ was proved independently by Bernasconi [1988]. Nice accounts of the method can also be found in Aarts & Van Laarhoven [1992], Van der Ham [1988], and Van Lint [1989].

Due to the fact that many local search methods are very time-consuming, any possibilities of speeding up the algorithms should be used. By using appropriate data structures much can be won. A one-dimensional integer array covered [ ] with q^n elements, one for each word in the space, can be used to tell how many times each word is covered. Now, the cost of a new solution can be calculated by considering only those words covered by the replaced codeword and the codeword that substitutes it. To find the neighbors of a codeword and the words covered by it, a two-dimensional array neighbor [ ][ ] with $|B_R(x)| \times q^n$ elements turns out to be very useful (as long as its size is within the memory limits of the computer). During the search this array is not altered, so its entries can be determined in advance.

A recent study by Östergard [1997] indicates that an increase in search speed can be obtained by using *tabu search*. The tabu list consists of indices of recently perturbed codewords (if c_i is perturbed, then i is added to the tabu list), and the following neighborhood function is used.

- *Neighborhood structure.* We go through the uncovered words in the space one by one, e.g., in the lexicographic order. The neighborhood with respect to an uncovered word x consists of those solutions that cover x and that can be obtained by perturbing exactly one codeword of the present solution.

3.3 A matrix method

When n grows, the required computation times become very long. One possible solution to this problem is to somehow restrict the search space, hopefully in a way that essentially reduces the computation times but does not too severely restrict the class of codes. One way would be to insist that the code has to be linear. Tables of the best currently known linear covering codes can be found in Lobstein & Pless [1994]. The following theorem describes a class of codes that is much larger than the class of linear codes and still possesses similar useful properties.

Let $F = \mathbb{Z}_q$ or $F = \mathbf{F}_q$, and let $A = (I_r, D)$ be an $r \times n$ matrix over F, where I_r is the $r \times r$ identity matrix. A set $S \subseteq F^r$ is said to R-cover F^r using A if every $x \in F^r$ can be represented as a sum of exactly one element of S and an F-linear combination of at most R of the columns of A, i.e., if given a word $x \in F^r$ we can find a word $y \in F^n$ of weight at most R and a word $s \in S$ such that $x = Ay + s$.

Theorem 1 *If S is an R-covering of F^r using A then the code*

$$C = \{x \in F^n \mid Ax \in S\}$$

has $|S|q^{n-r}$ codewords and covering radius at most R.

Proof Suppose $z \in F^n$. Then $Az \in F^r$, and we can find a word $y \in F^n$ of weight at most R such that $Az = Ay + s$ for some $s \in S$. Then $A(z - y) \in S$, and hence $z - y \in C$ and $d_H(z, z - y) \leqslant R$. The cardinality of the code is clear. □

The case $R = 1$ is due to Blokhuis & Lam [1984], who generalized a method of Kamps & Van Lint [1970]; the generalization to arbitrary R was independently presented by Van Lint Jr. [1988] and Carnielli [1990]. This theorem can also be generalized to many of the covering problems discussed at the end of this section.

We can now use local search methods, e.g., simulated annealing, to find A and S [Van Laarhoven et al., 1989; Östergard, 1994b]. First, the parameters q, n, R, r, and $|S|$ are fixed. A suitable cost function is in this case the number of words in F^r that *cannot* be written in the form $Ay + s$ for any $y \in F^n$ of weight at most R and $s \in S$. Three main approaches have been proposed to find words in S and columns in A that generate a covering. A neighbor can be obtained in the following ways during the search.

- Change both the words in S and the columns in the matrix A during the annealing process [Van Laarhoven et al., 1989].
- Change only the words in S, and consider a large number of matrices A, one at a time. For example, if A is an $(n - 1) \times n$ matrix it turns out that we need only consider $n - 2$ matrices where the D part (one column) consists of 2 to $n - 1$ ones [Östergard, 1994a, 1994b, 1997].
- Change only the words in S, and consider a matrix A that has some nice combinatorial properties; for example, it is obtained from a good linear code or otherwise has a special structure [Koschnick, 1993; Östergard, 1994b].

Furthermore, these alterations can be made in many ways. For example, in the first of these approaches, the columns of A can be changed by randomly choosing a column and replacing it by a randomly chosen column not in A [Van Laarhoven et al., 1989]. All of these approaches have been used to find record-breaking covering codes. Linear covering codes can be found by setting $S = \{00\ldots0\}$ and only changing the columns of A during the search.

3.4 The football pool problem

The problem of determining values of $K_3(n, 1)$ has intrigued combinatorialists for a long time. This is called the *football pool problem*. See Blokhuis & Lam [1984], Fernandes & Rechtschaffen [1983], Kamps & Van Lint [1967, 1970], Stanton [1969], Weber [1983], and their references for early upper and lower bounds on this function. In the football pools, three alternatives are given for each match: home win, draw, and away win. Assume that there are n matches on a football pool coupon or that you are 'sure' about the outcomes of all but n matches. Now if you want to get the first prize, whatever the outcomes of these matches, you have to fill in 3^n coupons. However, if you are satisfied with the second prize, $K_3(n, 1)$ coupons will do.

Exact values of $K_3(n, 1)$ are known only for $n \leqslant 5$ and for $n = (3^k - 1)/2$, $k \geqslant 3$. The cases $n \leqslant 3$ are easy. Kamps & Van Lint [1967] proved that $K_3(5, 1) = 27$. The others follow from the existence of perfect ternary Hamming codes [e.g., Van Lint, 1982, p. 36]. The fact that for codes of length $6 \leqslant n \leqslant 12$ only the upper bound $K_3(10, 1) \leqslant 3645$ [Blokhuis & Lam, 1984] has been able to resist attacks based on simulated annealing and tabu search demonstrates the importance of local search in the construction of covering codes.

Van Laarhoven et al. [1989] found a code proving $K_3(7, 1) \leqslant 186$. They did not impose any structure on the code, but it turns out that the bound can be proved using Theorem 1 and a code with $|S| = 62$, $r = 6$, $n = 7$, and $D = (111000)^T$ or $D = (111110)^T$ [Koschnick, 1993; Östergard, 1993]. This is interesting for two reasons. Firstly, codes can be found much faster in this way. Secondly, it is always desirable that the codes have some structure, which may permit a nice mathematical proof of the covering. Two such proofs can be found in Van Laarhoven et al. [1989] and Östergard [1994b], where $K_3(8, 1) \leqslant 486$ and $K_3(11, 1) \leqslant 9477$ are proved, respectively. Apart from these references, new upper bounds on $K_3(n, 1)$ are also presented by Östergard [1997] and Östergard & Hämäläinen [1997].

3.5 Multiple coverings and other problems

Together with the football pool problem, the problem of finding good *binary* covering codes is the most interesting. The best known lower and upper bounds for $K(n, R)$ with $n \leqslant 16$ and $R \leqslant 4$ are shown in Table 13.1. Upper bounds

Table 13.1 Bounds for $K(n, R)$; starred entries are upper bounds obtained by local search

n	$R = 1$	$R = 2$	$R = 3$	$R = 4$
1	1			
2	2	1		
3	2	2	1	
4	4	2	2	1
5	7	2	2	2
6	12	4	2	2
7	16	7	2	2
8	32	11–12	4	2
9	55–62*	15–16	7	2
10	105–120*	23–30*	9–12	4
11	178–192	36–44	12–16	7
12	342–380*	61–78*	18–28*	8–12
13	598–736*	97–128	28–42*	11–16
14	1172–1408*	157–256	44–64	15–28
15	2^{11}	309–384*	70–112	22–40*
16	2^{12}	512–768	114–192*	33–64

*Found by local search.

obtained by local search are starred. They have been reported by Hämäläinen et al. [1993], Hämäläinen et al. [1995a], Hämäläinen & Rankinen [1991], Östergard [1991a, 1994a, 1998], Östergard & Hämäläinen [1997], Östergard & Kaikkonen [1998], and Wille [1990, 1996]. For the lower bounds, see Li & Chen [1994] and their references, Habsieger [1994, 1997], and Honkala [1994]. Other types of covering codes to which simulated annealing has been applied are codes over mixed alphabets [Östergard, 1994a; Östergard & Hämäläinen, 1997].

Many constructions can be used to build new longer codes from shorter ones. It is often essential that the codewords of the initial codes can be partitioned in a certain way; see Cohen, Lobstein & Sloane [1986], Graham & Sloane [1985], Honkala [1991], and Östergard [1991b]. A code $C \subseteq \mathbb{Z}_q^n$ that has covering radius R is said to have (k, t)-subnorm S if there is a partition $C = C_1 \cup C_2 \cup \cdots \cup C_k$ such that $\min_a d_H(x, C_a) + \max_a d_H(x, C_a) \leqslant S$ whenever $R - t \leqslant d_H(x, C) \leqslant R$. For a given code C and subnorm S, local search can be used to find such a partition (if it exists). Östergard [1992] employs simulated annealing to solve this problem (especially for $t = 0$). Another closely related partitioning problem is also discussed in the same paper. See also Östergard & Kaikkonen [1998], where tabu search is used to solve the same problem.

There are several natural generalizations of the basic covering problem. Define

$K_q(n, r, \mu)$ = the smallest possible cardinality of a code $C \subseteq \mathbb{Z}_q^n$ such that every $x \in \mathbb{Z}_q^n$ is covered by at least μ codewords,

i.e., $|B_r(x) \cap C| \geqslant \mu$ for all $x \in \mathbb{Z}_q^n$. Such a code is called a *multiple covering* or, more precisely, a μ-fold covering. Numerical tables for $q = 2$, $n \leqslant 16$, $r \leqslant 4$, and $\mu \leqslant 4$ and references about this problem can be found in Hämäläinen et al. [1993]. Theorem 1 readily generalizes to multiple coverings, and many of the upper bounds of Hämäläinen et al. [1993] were indeed found using this method and simulated annealing. Some of the bounds have been further improved using tabu search [Östergard, 1995]. Computer programs that search for covering codes can very easily be modified to search for multiple coverings. We only need to change the cost function to $f(C) = \sum_{x \in \mathbb{Z}_q^n} \max\{0, \mu - |B_r(x) \cap C|\}$. The process of updating this function during the search is again facilitated by using the array covered [ ], defined earlier. Also, for the next few problems, only slight modifications of the basic search program are necessary.

A closely related problem is to study the function

$F_q(n, r, \mu)$ = the smallest possible cardinality of a code $C \subseteq \mathbb{Z}_q^n$ such that the covering radius of C is at most r and $|B_r(x) \cap C| \geqslant \mu$ whenever $d_H(x, C) = r$.

Such a code is called a *multiple covering of the farthest-off points*; see Hämäläinen et al. [1995a], where tables of upper bounds for $q = 2$, $n \leqslant 16$, $r \leqslant 4$, and $\mu \leqslant 4$ can also be found. Theorem 1 generalizes for this case, too and again many of the codes given by Hämäläinen et al. [1995a] were found using simulated annealing.

If we allow multiple coverings or multiple coverings of the farthest-off points to have the same codeword more than once, we obtain two very similar problems

[Hämäläinen et al., 1993; Hämäläinen et al., 1995a]. Local search could also be used in studying an even more general class called *weighted coverings*; see Cohen et al. [1995].

Local search techniques can also be applied if we have two sets $\mathscr{A}, \mathscr{B} \subseteq \mathbb{Z}_q^n$ and we wish to cover all the points in $\mathscr{A}$ using the points in $\mathscr{B}$. The most important special case is when $\mathscr{A} \subseteq \mathbb{Z}_2^n$ is the set of words of weight v and $\mathscr{B} \subseteq \mathbb{Z}_2^n$ is the set of words of weight u for some u, v. We then wish to determine the values of the function

$K(n, u, v, r)$ = the smallest possible cardinality of a binary code of length n and constant weight u such that every word of weight v is within distance r from at least one codeword.

For bounds on such codes, see Etzion, Wei & Zhang [1995]. Simulated annealing has been used in the search for such coverings by Nurmela [1993] and Nurmela & Östergård [1993a, 1993b]. In Honkala & Hämäläinen [1991] we are given a 258-element subset $\mathscr{B} \subseteq \mathbb{Z}_2^9$ and we need to find the smallest possible code C with covering radius 1 such that $C \subseteq \mathscr{B}$. Hämäläinen et al. [1995b] mention some other problems of this type related to the football pool problem.

4 SOURCE CODES

Before a signal from an (analog or digital) input source, such as a camera or a microphone, can be handled by a digital communication system, it has to be mapped to accurate digital representations. A widely utilized method for such *source coding* is *vector quantization* [Gersho & Gray, 1992]: vectors of input data are approximated by predetermined vectors in a codebook.

Data from the information source is denoted by x_i, $i \geqslant 0$. For binary digital sources $x_i \in \{0, 1\}$, for analog sources $x_i \in \mathbb{R}$. These signals are combined into n-tuples. The codebook (source code) consists of M n-tuples: $\mathbf{Y} = \{y_0, y_1, \ldots, y_{M-1}\}$. An n-dimensional signal x is now mapped into one of the vectors y_i in the codebook and the index i is transmitted. At the receiving end, the vector y_i can then be retrieved. The problem is (for a given M) to choose the vectors in the codebook so that the original signals are distorted as little as possible when they are mapped into these vectors. The average distortion per sample is

$$\delta = \frac{1}{n} \sum_{m=0}^{M-1} \int_{S_m} P(x) d(x, y_m) dx, \tag{1}$$

where $P(x)$ is the density function for x, S_m is the region consisting of the vectors that are mapped into y_m, and (most commonly) $d(x, y_m) = \| x - y_m \|^2$. Minimal possible distortion for a given codebook is obtained if the *nearest neighbor condition* is employed, that is, x is encoded into y_m only if $d(x, y_m) = d(x, \mathbf{Y}) = \min \{d(x, y) | y \in \mathbf{Y}\}$.

Lloyd [1982] and Max [1960] independently devised an algorithm for constructing fairly good source codes for scalar sources ($n = 1$). Linde, Buzo & Gray

[1980] presented an algorithm that is an extension of the Lloyd–Max algorithm to vector sources. Usually called the LBG algorithm, in the literature it is sometimes known as the GLA (generalized Lloyd algorithm). These deterministic algorithms iteratively improve a code until it cannot be further improved. The final code thus has locally minimal distortion. Even for very simple cases, this local minimum is not necessarily the global minimum [Cetin & Weerackody, 1988; Gray & Karnin, 1982]. It is then natural to try to apply local search methods that do not get stuck in local minima. The fairly long execution times of such methods do not affect the performance of the communication system, since the source code is constructed and stored in advance.

The first results in this direction were presented by El Gamal et al. [1987], who studied constructions of good source codes by simulated annealing. Only digital sources with evenly distributed signals are considered in their paper (for all $x \in \mathbb{Z}_2^n$, $P(x) = 1/2^n$). The distortion is *the average number of erroneous bits*, that is,

$$\delta = \frac{1}{2^n} \sum_{x \in \mathbb{Z}_2^n} d_H(x, \mathbf{Y}),$$

where $d_H(\cdot, \cdot)$ is the Hamming distance. The covering radius of Y gives an upper bound on the distortion. Implementation is quite straightforward. An appropriate cost function is $f(\mathbf{Y}) = \sum_{x \in \mathbb{Z}_2^n} d_H(x, \mathbf{Y})$ (which is an integer), and the neighborhood may be defined in a similar way as for covering codes.

For analog sources with the distortion given by (1), we have an optimization problem with continuous variables. The possibility of using simulated annealing and other local search methods to find good solutions to this problem has recently been discussed in a number of papers. The density function for the source signal is not known, in general, but is approximated through the use of a finite set of training vectors. A natural choice of cost function is then

$$f(\mathbf{Y}) = \sum_{m=0}^{M-1} \sum_{x \in S_m} d(x, y_m), \tag{2}$$

where x goes through the training vectors. Two main approaches for constructing good source codes from sets of training vectors have been proposed; Zeger, Vaisey & Gersho [1992] provide an extensive review. The difference between them lies in the definition of a solution. The approaches are as follows:

- Give the solution as a codebook $\mathbf{Y}$ (codebook approach).
- Give the solution as a partition $S_0, S_1, \ldots, S_{M-1}$ of the training vectors (clustering approach).

In the codebook approach, we start with a random initial code $\mathbf{Y}$. The cost of a solution $\mathbf{Y}$ is given by (2), and a neighbor is obtained by slightly perturbing the code. Now any local search method can be applied. Kodama, Wakasugi & Kasahara [1992] use simulated annealing and perturb a solution by replacing a randomly chosen vector (actually, two vectors are perturbed at a time)

$y_m = (y_{m,0}, y_{m,1}, \ldots, y_{m,n-1})$ by

$$y'_m = (y_{m,0}(1+\Delta_0), y_{m,1}(1+\Delta_1), \ldots, y_{m,n-1}(1+\Delta_{n-1})),$$

where each Δ_i is a uniform random variable in $[-\beta, \beta]$, with β fixed. A smoother convergence may be obtained in this case if the neighborhood function is altered (by decreasing β) during the optimization procedure [Zeger, Vaisey & Gersho, 1992]. A similar approach is also briefly considered by Cetin & Weerackody [1988].

In the clustering approach, we start with a random partition $S_0, S_1, \ldots, S_{M-1}$ of the training vectors. In the quantizer, the vectors in S_i are mapped into y_i, for which the optimal choice is the centroid $|S_i|^{-1} \sum_{x \in S_i} x$. We use the distortion as the cost function, and a neighbor is obtained by moving a randomly chosen training vector to a randomly chosen new set. The strength of this method compared to the previous one lies in the fact that only the training vectors of two sets S_i need to be considered in calculating the cost for a new solution. Applications of local search methods to this clustering problem are described, for example, by Bilbro, Hall & Ray [1992] and Flanagan et al. [1989].

Results have also been presented where local search methods and the LBG algorithm are combined [Vaisey & Gersho, 1988]. Some techniques, named stochastic relaxation, *unconditionally* perturb intermediate solutions of the LBG algorithm in a random way [Linde, Buzo & Gray, 1980; Zeger & Gersho, 1989; Zeger, Vaisey, & Gersho, 1992].

Research has recently been performed on combined source/channel coding, and on taking channel errors into account when the source code is designed and when indices are assigned to the vectors in the codebook. For example, Farvardin [1990], Farvardin & Vaishampayan [1991], Goodman & Moulsley [1988], and Kodama, Wakasugi & Kasahara [1992] have applied local search methods to these problems.

In the literature these new methods have been evaluated using Gauss–Markov, image and speech test sources [e.g., Huang & Harris, 1993].

5 LOW AUTOCORRELATION BINARY SEQUENCES

A *binary sequence* of length N is a vector $s = (s_1, s_2, \ldots, s_N)$, where each s_i is either -1 or 1, and its (aperiodic) autocorrelations are defined by

$$R_k = \sum_{i=1}^{N-k} s_i s_{i+k}, \; k = 0, 1, \ldots, N-1.$$

Binary sequences with low autocorrelations have important applications in communication [e.g., Bernasconi, 1987]. The quality of a sequence can be measured using the *merit factor*, introduced by Golay [1972], which is defined as

$$F = \frac{N^2}{2\sum_{k=1}^{N-1} R_k^2}.$$

The problem is to find the sequence of each length with the largest merit factors. Denote by F_n the largest possible merit factor of a sequence of length n. Using the ergodicity postulate [Golay, 1977], Golay [1982] showed that

$$\lim_{n \to \infty} F_n = 12.32\ldots.$$

The maximum merit factors of all sequences of length up to 32 have been determined by Turyn by exhaustive search and can be found in Golay [1982].

A binary sequence of length $N = 2n - 1$ is called *skew-symmetric* [Golay, 1977] if

$$s_{n-i} = (-1)^i s_{n+i}, \text{ for all } i = 1, 2, \ldots n - 1.$$

For skew-symmetric sequences all the autocorrelations R_k with k odd are zero. For this and other reasons [Golay, 1977, 1982; De Groot & Würtz, 1991], these sequences are good candidates for large merit factors. Restricting the search space to skew-symmetric sequences reduces the problem size by a factor of 2. By exhaustive search the largest merit factors of skew-symmetric binary sequences have been found for lengths up to 71 [Golay, 1977; Golay & Harris, 1990; De Groot, Würtz & Hoffmann, 1989]. Other limited searches are discussed by Golay [1977]. Golay & Harris [1990] found many best known skew-symmetric sequences of length at most 117 using a limited search. Golay [1983] uses quadratic residues to construct good binary sequences of length p, where p is an odd prime; see also Jensen, Jensen & Høholdt [1991].

Beenker, Claasen & Hermens [1985] have done computer searches for binary skew-symmetric sequences of length $N \leqslant 199$ using local search techniques. They discuss several methods that are based on iterative improvement and simulated annealing. The cost of the initial solution is required to be higher than a given threshold F_{th} (usually $F_{th} = 2$ or $F_{th} = 1.5$) before the iterative improvement procedure is applied. In one method an initial solution is obtained by randomly choosing $s_1, s_2, \ldots, s_{n-1}$ and taking $s_n = 1$ or -1, depending on which value yields the larger merit factor. In another method, it is defined as

$$s_k = \operatorname{sgn}\{\sin((k-1)k\pi/2N + \varphi) + c \operatorname{rand}(k)\},\ k = 1, 2, \ldots, n-1,$$

where $\operatorname{sgn}\{x\}$ is 1 if $x \geqslant 0$ and -1 otherwise, φ is a random real number between 0 and π (independent of k), c is a constant, and $\operatorname{rand}(k)$ is a random real number between -1 and 1. Iterative improvement is carried out by consecutively checking if changing s_i for $i = 1, \ldots, n-1$ gives a better sequence. If a better sequence is found, it is accepted, and the process is continued until no further improvement can be found. Simulated annealing was also implemented with the same generation mechanism. It turned out to yield merit factors that are comparable with the methods based on iterative improvement. Beenker, Claasen & Hermens [1985] give tables of the best found merit factors for all odd $N \leqslant 199$; see also De Groot, Würtz & Hoffmann [1989]. The most recent tables can be found in Mühlenbein [1991b], where results up to 201 are reported.

Bernasconi [1987] studied the distribution of merit factor values of long binary sequences and its characteristics by comparing approximate expressions based

on Golay's ergodicity postulate and numerical results obtained using iterative improvement and simulated annealing. The results indicate that the highest merit factor configurations are extremely isolated in the configuration space, like the holes in the golf course. See also Golay & Harris [1990]. Bernasconi [1988] also uses his results to present an optimized cooling schedule.

Evolutionary and genetic algorithms have also given new results in the search for low autocorrelation binary skew-symmetric sequences. These algorithms have recently been used by De Groot, Würtz & Hoffmann [1989], Mühlenbein [1992a], and Wang [1987]. See the references for further details on these methods and their computer implementations.

ACKNOWLEDGMENTS

We would like to thank all those people who have sent us copies of their papers. The second author gratefully acknowledges financial support from the Ella and Georg Ehrnrooth Foundation and Helsingin Sanomat Centenary Foundation.

Bibliography

E.H.L. AARTS, F.M.J. DE BONT, E.H.A. HABERS, P.J.M. VAN LAARHOVEN (1985). Statistical cooling: a general approach to combinatiorial optimization problems. *Philips Journal of Research* **4**, 193–226.

E.H.L. AARTS, F.M.J. DE BONT, E.H.A. HABERS, P.J.M. VAN LAARHOVEN (1986). Parallel implementations of the statistical cooling algorithm. *Integration, The VLSI Journal* **3**, 209–238.

E.H.L. AARTS, J.H.M. KORST (1989a). *Simulated Annealing and Boltzmann Machines*, Wiley, Chichester.

E.H.L. AARTS, J.H.M. KORST (1989b). Boltzmann machines for traveling salesman problems. *European Journal of Operational Research* **39**, 79–95.

E.H.L. AARTS, J.H.M. KORST (1991). Boltzmann machines as a model for massively parallel annealing. *Algorithmica* **6**, 437–465.

E.H.L. AARTS, J.H.M. KORST, P.J.M. VAN LAARHOVEN (1988). A quantitative analysis of the simulated annealing algorithm: a case study for the traveling salesman problem. *Journal of Statistical Physics* **50**, 189–206.

E.H.L. AARTS, P.J.M. VAN LAARHOVEN (1985a). Simulated annealing: an introduction. *Statistica Neerlandica* **43**, 31–52.

E.H.L. AARTS, P.J.M. VAN LAARHOVEN (1985b). Statistical cooling: a general approach to combinatorial optimization problems. *Philips Journal of Research* **40**, 193–226.

E.H.L. AARTS, P.J.M. VAN LAARHOVEN (1992). Local search in coding theory. *Discrete Mathematics* **106/107**, 11–18.

E.H.L. AARTS, P.J.M. VAN LAARHOVEN, J.K. LENSTRA, N.L.J. ULDER (1994). A computational study of local search algorithms for job shop scheduling. *ORSA Journal on Computing* **6**, 118–125.

E.H.L. AARTS, J.K. LENSTRA (EDS.) (1997). *Local Search in Combinatorial Optimization*, Wiley, Chichester.

E.H.L. AARTS, H.P. STEHOUWER (1993). Neural networks and the travelling salesman problem. S. GIELEN, B. KAPPEN (EDS.). *Proceedings of the International Conference on Artificial Neural Networks*, Springer, Berlin, 950–955.

T.S. ABDUL-RAZAQ, C.N. POTTS, L.N. VAN WASSENHOVE (1990). A survey of algorithms for the single machine total weighted tardiness scheduling problem. *Discrete Applied Mathematics* **26**, 235–253.

M. ABRAMOWITZ, I.A. STEGUN (EDS.) (1970). *Handbook of Mathematical Functions*, 9th ed., Dover, New York.

A. ACAN, Z. UNVER (1992). Switchbox routing by simulated annealing: SAR. *1992 IEEE International Symposium on Circuits and Systems*, IEEE, New York, 1985–1988.

J.O. ACHUGBUE, F.Y. CHIN (1982). Scheduling the open shop to minimize mean flow time. *SIAM Journal on Computing* **11**, 709–720.

D.H. ACKLEY, G.E. HINTON, T.J. SEJNOWSKI (1985). A learning algorithm for Boltzmann machines. *Cognitive Science* **9**, 147–169.

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

B. ADENSO-DÍAS (1992). Restricted neighborhood in the tabu search for the flowshop problem. *European Journal of Operational Research* **62**, 27–37.

I. ADLER, G.B. DANTZIG (1974). Maximum diameter of abstract polytopes. *Mathematical Programming Studies* **1**, 20–40.

I. ADLER, R. SAIGAL (1976). Long monotone paths in abstract polytopes. *Mathematics of Operations Research* **1**, 89–95.

B.-H. AHN, J.H. HYUN (1990). Single facility multi-class job scheduling. *Computers & Operations Research* **17**, 265–272.

A.V. AHO, J.E. HOPCROFT, J.D. ULLMAN (1974). *The Design and Analysis of Algorithms*, Addison-Wesley, Reading, MA.

D. ALDOUS (1983). Minimization algorithms and random walk on the *d*-cube. *Annals of Probability* **11**, 403–413.

D. ALDOUS, J. PITMAN (1983). The asymptotic speed and shape of a particle system. J.F.C. KINGMAN, G.E.H. REUTER (EDS.). *Probability, Statistics and Analysis*, London Mathematical Society Lecture Notes 79, Cambridge University Press, Cambridge, UK, 1–23.

A.S. ALFA, S.S. HERAGU, M. CHEN (1991). A 3-opt based simulated annealing algorithm for vehicle routing problems. *Computers & Industrial Engineering* **21**, 635–639.

J.R.A. ALLWRIGHT, D.B. CARPENTER (1989). A distributed implementation of simulated annealing for the travelling salesman problem. *Parallel Computing* **10**, 335–338.

N. ALON, J.H. SPENCER, P. ERDÖS (1992). *The Probabilistic Method*, Wiley, New York.

H. ALT, T. HAGERUP, K. MEHLHORN, F.P. PREPARATA (1987). Deterministic simulation of idealized parallel computers on more realistic ones. *SIAM Journal on Computing* **16**, 808–835.

I. ALTHÖFER, K.-U. KOSCHNICK (1991). On the convergence of 'threshold accepting'. *Applied Mathematics and Optimization* **24**, 183–195.

I. ALTHÖFER, K.-U. KOSCHNICK (1993). On the deterministic complexity of searching local maxima. *Discrete Applied Mathematics* **43**, 111–113.

S. AMIN (1994). A self-organized travelling salesman. *Neural Computing and Applications* **2**, 129–133.

B. ANGÉNIOL, G.D.L.C. VAUBOIS, J.-Y. LE TEXIER (1988). Self-organizing feature maps and the travelling salesman problem. *Neural Networks* **1**, 289–293.

S. ANILY, A. FEDERGRUEN (1987a). Ergodicity in parametric nonstationary Markov chains: an application to simulated annealing methods. *Operations Research* **35**, 867–874.

S. ANILY, A. FEDERGRUEN (1987b). Simulated annealing methods with general acceptance probabilities. *Journal of Applied Probability* **24**, 657–667.

D. APPLEGATE, R.E. BIXBY, V. CHVÁTAL, W. COOK (1994). *Private communication.*

D. APPLEGATE, R.E. BIXBY, V. CHVÁTAL, W. COOK (1995). *Finding cuts in the TSP: a preliminary report*, Report 95–05, DIMACS, Rutgers University, New Brunswick, NJ.

D. APPLEGATE, V. CHVÁTAL, W. COOK (1990). *Data structures for the Lin–Kernighan heuristic*, Talk presented at the CRPC TSP Workshop, April 22–24, 1990, Rice University, Houston, TX.

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

D. APPLEGATE, W. COOK (1993). Solving large-scale matching problems. D.S. JOHNSON, C.C. MCGEOCH (EDS.). *Network Flows and Matching: First DIMACS Implementation Challenge*, AMS, Providence, RI, 557–576.

D. APPLEGATE, W. COOK (1994). *Private communication.*

J.R. ARAQUE, G. KUDVA, T.L. MORIN, J.F. PEKNY (1994). A branch-and-cut algorithm for the vehicle routing problem. C.C. RIBEIRO, N. MACULAN (EDS.). *Applications of Combinatorial Optimization*, Annals of Operations Research 50, Baltzer, Basel, 37–59.

S. AREIBI, A. VANNELLI (1993). Circuit partitioning using a tabu search approach. *1993 IEEE International Symposium on Circuits and Systems*, IEEE, New York, 1643–1646.

I. ARIZONO, A. YAMAMOTO, H. OHTA (1992). Scheduling for minimizing total actual flow time by neural networks. *International Journal of Production Research* **30**, 503–511.

S. ARORA (1996). Polynomial time approximation schemes for Euclidean TSP and other geometric problems. *Proceedings 37th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society Press, Los Alamitos, CA, 2–11.

S. ARORA, C. LUND, R. MOTWANI, M. SUDAN, M. SZEGEDY (1992). Proof verification and hardness of approximation problems. *Proceedings 33rd Annual Symposium on Foundations of Computer Science*, IEEE Computer Society Press, Los Alamitos, CA, 14–23.

C. ARTHUR (1994). Smart ants solve travelling salesman problem. *New Scientist* **142**, 1928 (June 4, 1994), 6.

H. ASOH, H. MÜHLENBEIN (1994a). Estimating the heritability by decomposing the genetic variance. Y. DAVIDOR, H.-P. SCHWEFEL, R. MÄNNER (EDS.). *Parallel Problem Solving from Nature III*, Lecture Notes in Computer Science 866, Springer, Berlin, 98–107.

H. ASOH, H. MÜHLENBEIN (1994b). On the mean convergence time of evolutionary algorithms without selection and mutation. Y. DAVIDOR, H.-P. SCHWEFEL, R. MÄNNER (EDS.). *Parallel Problem Solving from Nature III*, Lecture Notes in Computer Science 866, Springer, Berlin, 88–97.

A.A. ASSAD (1988). Modeling and implementation issues in vehicle routing. B.L. GOLDEN, A.A. ASSAD (EDS.). *Vehicle Routing: Methods and Studies*, North-Holland, Amsterdam, 7–45.

G. AUSIELLO, M. PROTASI (1995). Local search, reducibility and approximability of NP-optimization problems. *Information Processing Letters* **54**, 73–79.

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

T. BÄCK (1993). Optimal mutation rates in genetic search. S. FORREST (ED.). *Proceedings of the Fifth International Conference on Genetic Algorithms*, Morgan Kaufmann, San Mateo, CA, 2–9.

T. BÄCK, F. HOFFMEISTER, H.-P. SCHWEFEL (1991). A survey of evolution strategies. R.K. BELEW, L.B. BOOKER (EDS.). *Proceedings of the Fourth International Conference on Genetic Algorithms*, Morgan Kaufmann, San Mateo, CA, 2–9.

T. BÄCK, H.-P. SCHWEFEL (1993). An overview of evolutionary algorithms for parameter optimization. *Evolutionary Computation* **1**, 1–24.

A. BAGCHI, A.C. WILLIAMS (1989). *k-Order local optima and recognition problems for certain classes of pseudo-Boolean functions*, Research report 43–89, RUTCOR, Rutgers University, New Brunswick, NJ.

S. BAGCHI, S. UCKIN, Y. MIYABE, K. KAWAMURA (1991). Exploring problem-specific recombination operators for job shop scheduling. R.K. BELEW, L.B. BOOKER (EDS.). *Proceedings of the Fourth International Conference on Genetic Algorithms*, Morgan Kaufmann, San Mateo, CA, 10–17.

K.R. BAKER (1974). *Introduction to Sequencing and Scheduling*, Wiley, New York.

K.R. BAKER, G.D. SCUDDER (1990). Sequencing with earliness and tardiness penalties: a review. *Operations Research* **38**, 22–36.

E. BALAS, J.K. LENSTRA, A. VAZACOPOULOS (1995). The one-machine problem with delayed precedence constraints and its use in job shop scheduling. *Management Science* **41**, 94–109.

E. BALAS, A. VAZACOPOULOS (1998). Guided local search with shifting bottleneck for job shop scheduling, *Management Science* **44**, 262–275.

H.-J. BANDELT (1989). *NP-completeness for non-global subgraph medians*. Manuscript.

P. BANERJEE, M. JONES (1986). A parallel simulated annealing algorithm for standard cell placement on a hypercube computer. *IEEE International Conference on Computer-Aided Design: ICCAD-86*, IEEE Computer Society Press, Washington, DC, 34–37.

J.W. BARNES, J.B. CHAMBERS (1995). Solving the job shop scheduling problem with tabu search. *IIE Transactions* **27**, 257–263.

J.W. BARNES, M. LAGUNA (1993). Solving the multiple-machine weighted flow time problem using tabu search. *IIE Transactions* **25**, 121–128.

E.R. BARNES, A. VANNELLI, J.Q. WALKER (1988). A new heuristic for partitioning the nodes of a graph. *SIAM Journal on Discrete Mathematics* **1**, 299–305.

R. BATTITI, G. TECCHIOLLI (1992). *The reactive tabu search*, Preprint, Department of Mathematics, University of Trento, Trento.

R. BATTITI, G. TECCHIOLLI (1994). Simulated annealing and tabu search in the long run: a comparison on QAP tasks. *Computers & Mathematics with Applications* **28**, 1–8.

E.B. BAUM (1986a). *Iterated descent: a better algorithm for local search in combinatorial optimization problems*, Manuscript.

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

J. BEARDWOOD, J.H. HALTON, J.M. HAMMERSLEY (1959). The shortest path through many points. *Proceedings of the Cambridge Philosophical Society* **55**, 299–327.

J.E. BEASLEY (1983). Route first–cluster second methods for vehicle routing. *Omega* **11**, 403–408.

G.F.M. BEENKER, T.A.C.M. CLAASEN, P.W.C. HERMENS (1985). Binary sequences with a maximally flat amplitude spectrum. *Philips Journal of Research* **40**, 289–304.

L.W. BEINEKE, F. HARARY (1965). The genus of the n-cube. *Canadian Journal of Mathematics* **17**, 494–496.

R.K. BELEW, L.B. BOOKER (EDS.) (1991). *Proceedings of the Fourth International Conference on Genetic Algorithms*, Morgan Kaufmann, San Mateo, CA.

C.J.P. BÉLISLE, H.E. ROMEIJN, R.L. SMITH (1990). *Hide-and-seek: a simulated annealing algorithm for global optimization*, Technical report 90–25, Department of Industrial and Operations Engineering, University of Michigan, Ann Arbor, MI.

H. BELOUADAH, M.E. POSNER, C.N. POTTS (1992). Scheduling with release dates on a single machine to minimize total weighted completion time. *Discrete Applied Mathematics* **36**, 213–231.

H. BELOUADAH, C.N. POTTS (1994). Scheduling identical parallel machines to minimize total weighted completion time. *Discrete Applied Mathematics* **48**, 201–218.

J.L. BENTLEY (1975). Multidimensional binary search trees used for associative search. *Communications of the ACM* **18**, 509–517.

J.L. BENTLEY (1990a). Experiments on traveling salesman heuristics. *Proceedings of the First Annual ACM–SIAM Symposium on Discrete Algorithms*, ACM, New York, and SIAM, Philadelphia, PA, 91–99.

J.L. BENTLEY (1990b). K-d trees for semidynamic point sets. *Proceedings of the Sixth Annual ACM Symposium on Computational Geometry*, ACM, New York, 187–197.

J.L. BENTLEY (1992). Fast algorithms for geometric traveling salesman problems. *ORSA Journal on Computing* **4**, 387–411.

J. BERNASCONI (1987). Low autocorrelation binary sequences: statistical mechanics and configuration space analysis. *Journal de Physique* **48**, 559–567.

J. BERNASCONI (1988). Optimization problems and statistical mechanics. *Proceedings of the Workshop on Chaos and Complexity*, World Scientific, Singapore, 245–259.

G.L. BILBRO, L.C. HALL, L.A. RAY (1992). A provably convergent inhomogeneous genetic annealing algorithm. *Proceedings of the SPIE–Conference on Neural and Stochastic*

Methods in Image and Signal Processing, SPIE The International Society for Optical Engineering, Bellingham, WA, Volume 1766, 50–60.

K. BINDER (1978). *Monte Carlo Methods in Statistical Physics*, Springer, Berlin.

R.G. BLAND, D.F. SHALLCROSS (1989). Large traveling salesman problems arising from experiments in X-ray crystallography: a preliminary report on computation. *Operations Research Letters* **8**, 125–128.

J.L. BLANTON, R.L. WAINWRIGHT (1992). Vehicle routing with time windows using genetic algorithms. *Proceedings of the Sixth Oklahoma Symposium on Artificial Intelligence*, Tulsa, OK, 242–251.

J.L. BLANTON, R.L. WAINWRIGHT (1993). Multiple vehicle routing with time and capacity constraints using genetic algorithms. S. FORREST (ED.). *Proceedings of the Fifth International Conference on Genetic Algorithms*, Morgan Kaufmann, San Mateo, CA, 452–459.

J. BŁAZEWICZ, K.H. ECKER, G. SCHMIDT, J. WĘGLARZ (1994). *Scheduling in Computer and Manufacturing Systems*, 2nd ed., Springer, Berlin.

T. BLICKLE, L. THIELE (1995). A mathematical analysis of tournament selection. L.J. ESHELMAN (ED.). *Proceedings of the Sixth International Conference on Genetic Algorithms*, Morgan Kaufmann, San Francisco, CA, 9–16.

A. BLOKHUIS, C.W.H. LAM (1984). More coverings by rook domains. *Journal of Combinatorial Theory, Series A* **36**, 240–244.

F. BOCK (1958a). An algorithm for solving 'travelling-salesman' and related network optimization problems: abstract. *Bulletin Fourteenth National Meeting of the Operations Research Society of America*, 897.

F. BOCK (1958b). *An algorithm for solving 'traveling-salesman' and related network optimization problems*, Manuscript associated with talk presented at the Fourteenth National Meeting of the Operations Research Society of America, St. Louis, MO.

L. BODIN, B.L. GOLDEN, A. ASSAD, M.O. BALL (1983). Routing and scheduling of vehicles and crews: the state of the art. *Computers & Operations Research* **10**, 63–211.

M.C.S. BOERES, L.A.V. DE CARVALHO, V.C. BARBOSA (1992). A faster elastic-net algorithm for the traveling salesman problem. *IJCNN: International Joint Conference on Neural Networks*, IEEE, New York, II 215–220.

K.D. BOESE, A.B. KAHNG (1994). Best-so-far vs. where-you-are: implications for optimal finite-time annealing. *Systems & Control Letters* **22**, 71–78.

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

N. BOISSIN, J.-L. LUTTON (1993). A parallel simulated annealing algorithm. *Parallel Computing* **19**, 859–872.

B. BOLLOBÁS (1985). *Random Graphs*, Academic Press, Orlando, FL.

J.A. BONDY, U.S.R. MURTY (1976). *Graph Theory with application*, Macmillan, London.

E. BONOMI, J.-L. LUTTON (1984). The N-city travelling salesman problem: statistical mechanics and the Metropolis algorithm. *SIAM Review* **26**, 551–568.

F.M.J. DE BONT, E.H.L. AARTS, P. MEEHAN, C.G. O'BRIEN (1988). Placement of shapeable blocks. *Philips Journal of Research* **43**, 1–19.

J.-P. BOUCHAUD, M. MÉZARD, G. PARISI, J.S. YEDIDA (1991). Polymers with long-ranged self-repulsion: a variational approach. *Journal of Physics A* **24**, L1025–1030.

D.E. VAN DEN BOUT, T.K. MILLER III (1989). Improving the performance of the Hopfield Tank neural network through normalization and annealing. *Biological Cybernetics* **62**, 129–139.

R.M. BRADY (1985). Optimization strategies gleaned from biological evolution. *Nature* **317**, 804–806.

R.D. BRANDT, Y. WANG, A.J. LAUB, S.K. MITRA (1988). Alternative networks for solving the traveling salesman problem and the list-matching problem. *IEEE International Conference on Neural Networks*, IEEE, New York II 333–340.

H. BRAUN (1991). On solving travelling salesman problems by genetic algorithms. H.-P. SCHWEFEL, R. MÄNNER (EDS.). *Parallel Problem Solving from Nature*, Lecture Notes in Computer Science 496, Springer, Berlin, 129–133.

H.J. BREMERMANN, J. ROGHSON, S. SALAFF (1966). Global properties of evolution processes. H.H. PATTEE, E.A. EDELSACK, L. FEIN, A.B. CALLAHAN (EDS.). *Natural Automata and Useful Simulations*, Macmillan, London, 3–42.

A. BRONSTED (1983). *An Introduction to Convex Polytopes*, Springer, New York.

A.E. BROUWER, J.B. SHEARER, N.J.A. SLOANE, W.D. SMITH (1990). A new table of constant weight codes. *IEEE Transactions on Information Theory* **36**, 1334–1380.

R.A. BRUALDI, S.N. LITSYN, V.S. PLESS (1998). Covering radius. R.A. BRUALDI, W.C. HUFFMAN, V.S. PLESS (EDS.). *Handbook of Coding Theory*, North-Holland, Amsterdam, forthcoming.

J. BRUCK, J.W. GOODMAN (1988). A generalized convergence theorem for neural networks. *IEEE Transactions on Information Theory* **34**, 1089–1092.

P. BRUCKER, B. JURISCH, B. SIEVERS (1994). A branch & bound algorithm for the job-shop scheduling problem. *Discrete Applied Mathematics* **49**, 107–129.

L.J.J. VAN DER BRUGGEN, J.K. LENSTRA, P.C. SCHUUR (1993). Variable-depth search for the single-vehicle pickup and delivery problem with time windows. *Transportation Science* **27**, 298–311.

J.L. BRUNO, E.G. COFFMAN, JR., R. SETHI (1974). Scheduling independent tasks to reduce mean finishing time. *Communications of the ACM* **17**, 382–387.

J.L. BRUNO, P.J. DOWNEY (1978). Complexity of task sequencing with deadlines, set-up times and changeover costs. *SIAM Journal on Computing* **7**, 393–404.

T.N. BUI, B.R. MOON (1994). A new genetic approach for the traveling salesman problem. *ICEC'94: Proceedings of the First IEEE Conference on Evolutionary Computation*, IEEE, New York, 7–12.

M.G. BULMER (1980). *The Mathematical Theory of Quantitative Genetics*, Clarendon Press, Oxford.

L.I. BURKE, P. DAMANY (1992). The guilty net for the traveling salesman problem. *Computers & Operations Research* **19**, 255–265.

X. CAI (1995). Minimization of agreeably weighted variance in single machine systems. *European Journal of Operational Research* **85**, 576–592.

J. CARLIER (1982). The one-machine sequencing problem. *European Journal of Operational Research* **11**, 42–47.

J. CARLIER, E. PINSON (1989). An algorithm for solving the job-shop problem. *Management Science* **35**, 164–176.

J. CARLIER, P. VILLON (1990). A new heuristic for the traveling salesman problem. *Recherche Opérationnelle/Operations Research* **24**, 245–253.

W.A. CARNIELLI (1990). Hyper-rook domain inequalities. *Studies in Applied Mathematics* **82**, 59–69.

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

V. CERNÝ (1985). Thermodynamical approach to the traveling salesman problem: an efficient simulation algorithm. *Journal of Optimization Theory and Applications* **45**, 41–51.

A.E. CETIN, V. WEERACKODY (1988). Design vector quantizers using simulated annealing. *IEEE Transactions on Circuits and Systems* **35**, 1550.

H. CHAN, P. MAZUMDER, K. SHAHOOKAR (1991). Macro-cell and module placement by genetic adaptive search with bitmap-represented chromosome. *Integration, The VLSI Journal* **12**, 49–77.

B. CHANDRA, H. KARLOFF, C.A. TOVEY (1997). New results on the old k-opt algorithm for the TSP. *SIAM Journal on Computing*, forthcoming.

R. CHANDRASEKHARAM, S. SUBHRAMANIAN, S. CHAUDHURY (1993). Genetic algorithm for node partitioning problem and applications in VLSI design. *IEE Proceedings, Part E* **140**, 255–260.

S. CHANG, H. MATSUO, G. TANG (1990). Worst-case analysis of local search heuristics for the one-machine total tardiness problem. *Naval Research Logistics* **37**, 111–121.

A. CHATTERJEE, R. HARTLEY (1990). A new simultaneous circuit partitioning and chip placement approach based on simulated annealing. *27th ACM/IEEE Design Automation Conference: Proceedings 1990*, IEEE Computer Society Press, Los Alamitos, CA, 36–39.

N. CHRISTOFIDES (1976). *Worst-case analysis of a new heuristic for the travelling salesman problem*, Report 388, Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, PA.

N. CHRISTOFIDES (1985). Vehicle routing. E.L. LAWLER, J.K. LENSTRA, A.H.G. RINNOOY KAN, D.B. SHMOYS (EDS.). *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, Wiley, Chichester, 431–448.

N. CHRISTOFIDES, E. HADJICONSTANTINOU, A. MINGOZZI (1993). *A new exact algorithm for the vehicle routing problem based on q-path and k-shortest path relaxations*, Working paper, Management School, Imperial College, London.

N. CHRISTOFIDES, A. MINGOZZI, P. TOTH (1979). The vehicle routing problem. N. CHRISTOFIDES, A. MINGOZZI, P. TOTH, C. SANDI (EDS.). *Combinatorial Optimization*, Wiley, Chichester, 315–338.

M. CHRISTOPH, K.H. HOFFMANN (1993). Scaling behaviour of optimal simulated annealing schedules. *Journal of Physics A* **26**, 3267–3277.

C. CHU (1992). A branch-and-bound algorithm to minimize total flow time with unequal release dates. *Naval Research Logistics* **39**, 859–875.

M.J. CHUNG, K.K. RAO (1986). Parallel simulated annealing for partitioning and routing. *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers; ICCD'86*, IEEE Computer Society Press, Washington, DC, 238–242.

V. CHVÁTAL (1983). *Linear Programming*, Freeman, San Francisco, CA.

G. CLARKE, J.W. WRIGHT (1964). Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research* **12**, 568–581.

G.A. CLEVELAND, S.F. SMITH (1989). Using genetic algorithms to schedule flow-shop releases. J.D. SCHAFFER (ED.). *Proceedings of the Third International Conference on Genetic Algorithms*, Morgan Kaufmann, San Mateo, CA, 160–169.

B. CODENOTTI, G. MANZINI, L. MARGARA, G. RESTA (1993). Global strategies for augmenting the efficiency of TSP heuristics. F. DEHNE, J.-R. SACK, N. SANTORO, S. WHITESIDES (EDS.). *Algorithms and Data Structures*, Lecture Notes in Computer Science 709, Springer, Berlin, 253–264.

B. CODENOTTI, G. MANZINI, L. MARGARA, G. RESTA (1996). Perturbation: an efficient technique for the solution of very large instances of the Euclidean TSP. *INFORMS Journal on Computing* **8**, 125–133.

G.D. COHEN, P. FRANKL (1985). Good coverings of Hamming spaces with spheres. *Discrete Mathematics* **56**, 125–131.

G.D. COHEN, I.S. HONKALA, S.N. LITSYN, H.F. MATTSON, JR. (1995). Weighted coverings and packings. *IEEE Transactions on Information Theory* **41**, 1856–1867.

G.D. COHEN, M.R. KARPOVSKY, H.F. MATTSON, JR., J.R. SCHATZ (1985). Covering radius: survey and recent results. *IEEE Transactions on Information Theory* **31**, 328–343.

G.D. COHEN, A.C. LOBSTEIN, N.J.A. SLOANE (1986). Further results on the covering radius of codes. *IEEE Transactions on Information Theory* **32**, 680–694.

J.P. COHOON, S.U. HEGDE, W.N. MARTIN, D.S. RICHARDS (1991). Distributed genetic algorithms for the floorplan design problem. *IEEE Transactions on Computer-Aided Design* **10**, 483–492.

J.P. COHOON, W.D. PARIS (1987). Genetic placement. *IEEE Transactions on Computer-Aided Design* **6**, 956–964.

N.E. COLLINS, R.W. EGLESE, B.L. GOLDEN (1988). Simulated annealing: an annotated bibliography. *American Journal of Mathematical and Management Sciences* **8**, 209–307.

A. COLORNI, M. DORIGO, V. MANNIEZZO (1992). An investigation of some properties of an 'ant algorithm'. R. MÄNNER, B. MANDERICK (EDS.). *Parallel Problem Solving from Nature 2*, North-Holland, Amsterdam, 509–520.

F. COMELLAS, R. ROCA (1993). Using genetic algorithms to design constant weight codes. J. ALSPECTOR, R. GOODMAN, T.X. BROWN (EDS.). *Proceedings of the International Workshop on Applications of Neural Networks to Telecommunication*, Lawrence Erlbaum, Hillsdale, NJ, 119–124.

A. CONDON (1989). *Computational Models of Games*, MIT Press, Cambridge, MA.

D. CONNOLLY (1990). An improved annealing scheme for the QAP. *European Journal of Operational Research* **46**, 93–100.

D.P. CONNORS, P.R. KUMAR (1987). Simulated annealing and balance of recurrence order in time-inhomogeneous Markov chains. *Proceedings of the 26th IEEE Conference on Decision and Control*, IEEE, New York, 2261–2263.

R.W. CONWAY, W.L. MAXWELL, L.W. MILLER (1967). *Theory of Scheduling*, Addison-Wesley, Reading, MA.

W. COOK, P.D. SEYMOUR (1993). *Private communication.*

G. CORNUÉJOLS, F. HARCHE (1993). Polyhedral study of the capacitated vehicle routing problem. *Mathematical Programming* **60**, 21–52.

G. CORNUÉJOLS, G.L. NEMHAUSER (1978). Tight bounds for Christofides' traveling salesman heuristic. *Mathematical Programming* **14**, 116–121.

D. COSTA (1994). A tabu search algorithm for computing an operational time table. *European Journal of Operational Research* **76**, 98–110.

R.W. COTTLE (1980). Observations on a nasty class of linear complementarity problems. *Discrete Applied Mathematics* **2**, 89–111.

Y. CRAMA (1989). Recognition problems for special classes of polynomials in 0–1 variables. *Mathematical Programming* **44**, 139–155.

H.A.J. CRAUWELS, C.N. POTTS, L.N. VAN WASSENHOVE (1997). Local search heuristics for single machine scheduling with batch set-up times to minimize total weighted completion time. C.-Y. LEE, L. LEI (EDS.). *Scheduling: Theory and Applications*, Annals of Operations Research 70, Baltzer, Amsterdam, 261–79.

H.A.J. CRAUWELS, C.N. POTTS, L.N. VAN WASSENHOVE (1998). Local search heuristics for the single machine total weighted tardiness scheduling problem. *INFORMS Journal on Computing*, to appear.

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

J.F. CROW (1986). *Basic Concepts in Population, Quantitative and Evolutionary Genetics*, Freeman, New York.

J.F. CROW, M. KIMURA (1970). *An Introduction to Population Genetics Theory*, Harper and Row, New York.

H. CROWDER, M. PADBERG (1980). Solving large-scale symmetric travelling salesman problems to optimality. *Management Science* **26**, 495–509.

M. DAM, M. ZACHARIASEN (1994). *Tabu Search on the Geometric Traveling Salesman Problem*, MSc thesis, Department of Computer Science, University of Copenhagen, Copenhagen.

D.G. DANNENBRING (1977). An evaluation of flow-shop sequencing heuristics. *Management Science* **23**, 1174–1182.

G.B. DANTZIG (1951). Maximization of a linear function of variables subject to linear inequalities. T.C. KOOPMANS (ED.). *Activity Analysis of Production and Allocation*, Wiley, New York, 339–347.

G.B. DANTZIG (1963). *Linear Programming and Extensions*, Princeton University Press, Princeton, NJ.

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

S. DAUZÈRE-PÉRÈS, J.-B. LASSERRE (1993). A modified shifting bottleneck procedure for job-shop scheduling. *International Journal of Production Research* **31**, 923–932.

J.S. DAVIS, J.J. KANET (1993). Single-machine scheduling with early and tardy completion costs. *Naval Research Logistics* **40**, 85–101.

L. DAVIS (1985). Job shop scheduling with genetic algorithms. J.J. GREFENSTETTE (ED.). *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, Lawrence Erlbaum, Hillsdale, NJ, 136–140.

E. DEKEL, S. SAHNI (1983). Binary trees and parallel scheduling algorithms. *IEEE Transactions on Computers* **32**, 307–315.

F. DELLA CROCE, R. TADEI, G. VOLTA (1995). A genetic algorithm for the job shop problem. *Computers & Operations Research* **22**, 15–24.

M. DELL'AMICO, F. MAFFIOLI, S. MARTELLO (EDS.) (1997). *Annotated Bibliographies in Combinatorial Optimization*, Wiley, Chichester.

M. DELL'AMICO, M. TRUBIAN (1993). Applying tabu-search to the job-shop scheduling problem. F. GLOVER, E. TAILLARD, M. LAGUNA, D. DE WERRA (EDS.). *Tabu Search*, Annals of Operations Research 41, Baltzer, Basel, 231–252.

P. DELSARTE, P. PIRET (1986). Do most binary linear codes achieve the Goblick bound on the covering radius? *IEEE Transactions on Information Theory* **32**, 826–828.

B. DENBY (1988). Neural networks and cellular automata in experimental high energy physics. *Computer Physics Communications* **49**, 429–448.

M. DESROCHERS, J.K. LENSTRA, M.W.P. SAVELSBERGH, F. SOUMIS (1988). Vehicle routing with time windows: optimization and approximation. B.L. GOLDEN, A.A. ASSAD (EDS.). *Vehicle Routing: Methods and Studies*, North-Holland, Amsterdam, 65–84.

J. DESROSIERS, Y. DUMAS, M.M. SOLOMON, F. SOUMIS (1995). Time constrained routing and scheduling. M.O. BALL, T.L. MAGNANTI, C.L. MONMA, G.L. NEMHAUSER (EDS.). *Network Routing*, Handbooks in Operations Research and Management Science, Volume 8, North-Holland, Amsterdam, 35–139.

D.N. DEUTSCH (1976). A 'dogleg' channel router. *13th Design Automation Conference: Proceedings*, IEEE, New York, 425–433.

H. DJIDJEV (1982). On the problem of partitioning planar graphs. *SIAM Journal on Algebraic and Discrete Methods* **3**, 229–240.

U. DORNDORF, E. PESCH (1995). Evolution based learning in a job shop scheduling environment. *Computers & Operations Research* **22**, 25–40.

K.A. DOWSLAND (1993). Simulated annealing. C.R. REEVES (ED.). *Modern Heuristic Techniques for Combinatorial Problems*, Blackwell, Oxford, 20–69.

J. DU, J.Y.-T. LEUNG (1990). Minimizing total tardiness on one processor is *NP*-hard. *Mathematics of Operations Research* **15**, 483–495.

R.O. DUDA, P.E. HART (1973). *Pattern Classification and Scene Analysis*, Wiley, New York.

G. DUECK (1993). New optimization heuristics: the great-deluge algorithm and the record-to-record-travel. *Journal of Computational Physics* **104**, 86–92.

G. DUECK, T. SCHEUER (1990). Threshold accepting: a general purpose optimization algorithm appearing superior to simulated annealing. *Journal of Computational Physics* **90**, 161–175.

A.E. DUNLOP, B.W. KERNIGHAN (1985). A procedure for placement of standard cell VLSI circuits. *IEEE Transactions on Computer-Aided Design* **4**, 92–98.

R. DURBIN, R. SZELISKI, A.L. YUILLE (1989). An analysis of the elastic net approach to the traveling salesman problem. *Neural Computation* **1**, 348–358.

R. DURBIN, D. WILLSHAW (1987). An analogue approach to the travelling salesman problem using an elastic net method. *Nature* **326**, 689–691.

J. EDMONDS (1965). Matching and a polyhedron with 0–1 vertices. *Journal of Research of the National Bureau of Standards* **69B**, 125–130.

J. EDMONDS, E.L. JOHNSON (1970). Matching: a well-solved class of integer linear programs. R. GUY, H. HANANI, N. SAUER, J. SCHONHEIM (EDS.). *Combinatorial Structures and Their Applications*, Gordon and Breach, New York, 89–92.

R.W. EGLESE (1990). Simulated annealing: a tool for operational research. *European Journal of Operational Research* **46**, 271–281.

A.E. EIBEN, E.H.L. AARTS, K.M. VAN HEE (1991). Global convergence of genetic algorithms: a Markov chain analysis. H.-P. SCHWEFEL, R. MÄNNER (EDS.). *Parallel Problem Solving from Nature*, Lecture Notes in Computer Science 496, Springer, Berlin, 4–12.

A.E. EIBEN, P.-E. RAUE, ZS. RUTTKAY (1994). Genetic algorithms with multiparent recombination. Y. DAVIDOR, H.-P. SCHWEFEL, R. MÄNNER (EDS.). *Parallel Problem Solving from Nature III*, Lecture Notes in Computer Science 866, Springer, Berlin, 78–87.

S. EILON, I.G. CHOWDHURY (1977). Minimizing waiting time variance in the single machine problem. *Management Science* **23**, 567–575.

A.A. EL GAMAL, L.A. HEMACHANDRA, I. SHPERLING, V.K. WEI (1987). Using simulated annealing to design good codes. *IEEE Transactions on Information Theory* **33**, 116–123.

H. EL GHAZIRI (1991). Solving routing problems by a self-organizing map. T. KOHONEN, K. MÄKISARA, O. SIMULA, J. KANGAS (EDS.). *Artificial Neural Networks*, North-Holland, Amsterdam, 829–834.

H. EL GHAZIRI (1993). *Algorithmes connexionnistes pour l' optimisation combinatoire*, PhD thesis 1167, Ecole Polytechnique Fédérale de Lausanne, Lausanne (in French).

H. EMMONS (1969). One-machine sequencing to minimize certain functions of job tardiness. *Operations Research* **17**, 701–715.

H. ESBENSEN (1992). A genetic algorithm for macro cell placement. *EURO-DAC'92: European Design Automation Conference; EURO-VHDL'92*, IEEE Computer Society Press, Los Alamitos, CA, 52–57.

L.J. ESHELMAN (1991). The CHC adaptive search algorithm: how to have safe search when engaging in nontraditional genetic recombination. G. RAWLINS (ED.). *Foundations of Genetic Algorithms*, Morgan Kaufmann, San Mateo, CA, 265–283.

L.J. ESHELMAN, J.D. SCHAFFER (1993). Crossover's niche. S. FORREST (ED.). *Proceedings of the Fifth International Conference on Genetic Algorithms*, Morgan Kaufmann, San Mateo, CA, 9–14.

T. ETZION, V. WEI, Z. ZHANG (1995). Bounds on the sizes of constant weight covering codes. *Design, Codes and Cryptography* **5**, 217–239.

A.A. EVDOKIMOV (1969). Maximal length of circuit in a unitrary n-dimensional cube. *Mathematical Notes* **6**, 642–648.

U. FAIGLE, W. KERN (1992). Some convergence results for probabilistic tabu search. *ORSA Journal on Computing* **4**, 32–37.

U. FAIGLE, R. SCHRADER (1988). On the convergence of stationary distributions in simulated annealing algorithms. *Information Processing Letters* **27**, 189–194.

D.S. FALCONER (1981). *Introduction to Quantitative Genetics*, Longman, London.

E. FALKENAUER, S. BOUFFOUIX (1991). A genetic algorithm for job shop. *Proceedings 1991 IEEE International Conference on Robotics and Automation*, IEEE Computer Society Press, Los Alamitos, CA, 824–829.

N. FARVARDIN (1990). A study of vector quantization for noisy channels. *IEEE Transactions on Information Theory* **36**, 799–809.

N. FARVARDIN, V. VAISHAMPAYAN (1991). On the performance and complexity of channel-optimized vector quantizers. *IEEE Transactions on Information Theory* **37**, 155–160.

Y. FATHI (1979). Computational complexity of LCPs associated with positive definite symmetric matrices. *Mathematical Programming* **17**, 335–344.

Y. FATHI, C.A. TOVEY (1986). Affirmative action algorithms. *Mathematical Programming* **34**, 292–301.

F. FAVATA, R. WALKER (1991). A study of the application of Kohonen-type neural networks to the travelling salesman problem. *Biological Cybernetics* **64**, 463–468.

J.A. FELDMAN, D.H. BALLARD (1982). Connectionist models and their properties. *Cognitive Science* **6**, 205–254.

W. FELLER (1950). *An Introduction to Probability Theory and Its Applications, Volume 1*, Wiley, New York, NY.

H. FERNANDES, E. RECHTSCHAFFEN (1983). The football pool problem for 7 and 8 matches. *Journal of Combinatorial Theory, Series A* **35**, 109–114.

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

C.M. FIDUCCIA, R.M. MATTHEYSES (1982). A linear-time heuristic for improving network partitions. *ACM IEEE Nineteenth Design Automation Conference: Proceedings*, IEEE Computer Society Press, Los Alamitos, CA, 175–181.

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

G. FINN, E. HOROWITZ (1979). A linear time approximation algorithm for multiprocessor scheduling. *BIT* **19**, 312–320.

S.T. FISCHER (1995). A note on the complexity of local search problems. *Information Processing Letters* **53**, 69–75.

H. FISHER, G.L. THOMPSON (1963). Probabilistic learning combinations of local job-shop scheduling rules. J.F. MUTH, G.L. THOMPSON (EDS.). *Industrial Scheduling*, Prentice Hall, Englewood Cliffs, NJ, 225–251.

M.L. FISHER (1995). Vehicle routing. M.O. BALL, T.L. MAGNANTI, C.L. MONMA, G.L. NEMHAUSER (EDS.). *Network Routing*, Handbooks in Operations Research and Management Science, Volume 8, North-Holland, Amsterdam, 1–33.

R.A. FISHER (1958). *The Genetical Theory of Natural Selection*, Dover, New York.

J.K. FLANAGAN, D.R. MORRELL, R.L. FROST, C.J. READ, B.E. NELSON (1989). Vector quantization codebook generation using simulated annealing. *ICASSP 89: 1989 International Conference on Acoustics, Speech, and Signal Processing*, IEEE, New York, 1759–1762.

M.M. FLOOD (1956). The traveling-salesman problem. *Operations Research* **4**, 61–75.

L.J. FOGEL, A.J. OWENS, M.J. WALSH (1966). *Artificial Intelligence Through Simulated Evolution*, Wiley, New York.

Y.-P.S. FOO, H. SZU (1989). Solving large-scale optimization problems by divide-and-conquer neural networks. *IJCNN: International Joint Conference on Neural Networks*, IEEE, New York, I 507–511.

Y.-P.S. FOO, Y. TAKEFUJI (1988a). Stochastic neural networks for solving job-shop scheduling: part 1, problem representation. *IEEE International Conference on Neural Networks*, IEEE, New York, II 275–282.

Y.-P.S. FOO, Y. TAKEFUJI (1988b). Stochastic neural networks for solving job-shop scheduling: part 2, architecture and simulations. *IEEE International Conference on Neural Networks*, IEEE, New York, II 283–290.

Y.-P.S. FOO, Y. TAKEFUJI (1988c). Integer linear programming neural networks for job-shop scheduling. *IEEE International Conference on Neural Networks*, IEEE, New York, II 341–348.

S. FORREST (ED.) (1993). *Proceedings of the Fifth International Conference on Genetic Algorithms*, Morgan Kaufmann, San Mateo, CA.

J.C. FORT (1988). Solving a combinatorial problem via self-organizing process: an application of the Kohonen algorithm to the traveling salesman problem. *Biological Cybernetics* **59**, 33–40.

M.P. FOURMAN (1985). Compaction of symbolic layout using genetic algorithms. J.J. GREFENSTETTE (ED.). *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, Lawrence Erlbaum, Hillsdale, NJ, 141–153.

B.L. FOX (1993). Integrating and accelerating tabu search, simulated annealing, and genetic algorithms. F. GLOVER, E. TAILLARD, M. LAGUNA, D. DE WERRA (EDS.). *Tabu Search*, Annals of Operations Research 41, Baltzer, Basel, 47–67.

B.L. FOX (1994). Random restart versus simulated annealing. *Computers & Mathematics with Applications* **27**, 33–35.

P.M. FRANCA, M. GENDREAU, G. LAPORTE, F.M. MÜLLER (1994). A composite heuristic for the identical parallel machine scheduling problem with minimum makespan objective. *Computers & Operations Research* **21**, 205–210.

P.M. FRANCA, M. GENDREAU, G. LAPORTE, F.M. MÜLLER (1995). The m-traveling salesman problem with minmax objective. *Transportation Science* **29**, 267–275.

M.L. FREDMAN, D.S. JOHNSON, L.A. MCGEOCH, G. OSTHEIMER (1995). Data structures for traveling salesmen. *Journal of Algorithms* **18**, 432–479.

S. FRENCH (1982). *Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop*, Ellis Horwood, Chichester.

C. FRIDEN, A. HERTZ, D. DE WERRA (1989). STABULUS: a technique for finding stable sets in large graphs with tabu search. *Computing* **42**, 35–44.

C. FRIDEN, A. HERTZ, D. DE WERRA (1990). TABARIS: an exact algorithm based on tabu search for finding a maximum independent set in a graph. *Computers & Operations Research* **17**, 437–445.

J. FRIEDMAN, J. KAHN, E. SZEMERÉDI (1989). On the second eigenvalue in random regular graphs. *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, ACM, New York, 587–598.

A.M. FRIEZE (1979). Worst-case analysis of algorithms for travelling salesman problems. *Methods of Operations Research* **32**, 97–112.

B. FRITZKE, P. WILKE (1991). FLEXMAP: a neural network for the traveling salesman problem with linear time and space complexity. *1991 IEEE International Joint Conference on Neural Networks*, IEEE, New York, 929–934.

H.N. GABOW (1973). *Implementations of algorithms for maximum matching on nonbipartite graphs*, PhD thesis, Department of Computer Science, Stanford University, Stanford, CA.

H.N. GABOW, R.E. TARJAN (1991). Faster scaling algorithms for general graph-matching problems. *Journal of the Association for Computing Machinery* **38**, 815–853.

B.L. GARCIA (1993). *Développement de techniques de recherche tabou pour le problème de tournées de véhicules avec fenêtres de temps*, Publication CRT-931, Centre de recherche sur les transports, Université de Montréal, Montréal (in French).

B.L. GARCIA, J.-Y. POTVIN, J.-M. ROUSSEAU (1994). A parallel implementation of the tabu search heuristic for vehicle routing problems with time window constraints. *Computers & Operations Research* **21**, 1025–1033.

M.R. GAREY, D.S. JOHNSON (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, CA.

M.R. GAREY, D.S. JOHNSON, R. SETHI (1976). The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research* **1**, 117–129.

M.R. GAREY, R.E. TARJAN, G.T. WILFONG (1988). One-processor scheduling with symmetric earliness and tardiness penalties. *Mathematics of Operations Research* **13**, 330–348.

B. GÄRTNER, G.M. ZIEGLER (1994). Randomized simplex algorithms on Klee Minty cubes. *Proceedings 35th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society Press, Los Alamitos, CA, 502–510.

S.B. GELFAND, S.K. MITTER (1985). Analysis of simulated annealing for optimization. *Proceedings of the 24th IEEE Conference on Decision & Control*, IEEE, New York, 779–786.

S. GEMAN, D. GEMAN (1984). Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **6**, 721–741.

M. GENDREAU, A. HERTZ, G. LAPORTE (1992). New insertion and post optimization procedures for the traveling salesman problem. *Operations Research* **40**, 1086–1094.

M. GENDREAU, A. HERTZ, G. LAPORTE (1994). A tabu search heuristic for the vehicle routing problem. *Management Science* **40**, 1276–1290.

M. GERACI, P. ORLANDO, F. SORBELLO, G. VASSALLO (1990). A genetic algorithm for the routing of VLSI circuits. *EDAC: Proceedings of the European Design Automation Conference*, IEEE Computer Society Press, Washington, DC, 218–223.

A. GERSHO, R.M. GRAY (1992). *Vector Quantization and Signal Compression*, Kluwer, Boston, MA.

J.B. GHOSH (1994). Batch scheduling to minimize total completion time. *Operations Research Letters* **16**, 271–275.

B. GIDAS (1985). Nonstationary Markov chains and convergence of the annealing algorithm. *Journal of Statistical Physics* **39**, 73–131.

B. GIFFLER, G.L. THOMPSON (1960). Algorithms for solving production-scheduling problems. *Operations Research* **8**, 487–503.

J.R. GILBERT, J.P. HUTCHINSON, R.E. TARJAN (1984). A separator theorem for graphs of bounded genus. *Journal of Algorithms* **3**, 391–407.

J.R. GILBERT, E. ZMIJEWSKI (1987). A parallel graph partitioning algorithm for a message-passing multiprocessor. *International Journal of Parallel Programming* **16**, 427–449.

B.E. GILLETT, L.R. MILLER (1974). A heuristic algorithm for the vehicle-dispatch problem. *Operations Research* **22**, 340–349.

L. GISLÉN, B. SÖDERBERG, C. PETERSON (1989). Teachers and classes with neural networks. *International Journal of Neural Systems* **1**, 167–176.

L. GISLÉN, B. SÖDERBERG, C. PETERSON (1992a). Rotor neurons: formalism and dynamics. *Neural Computation* **4**, 737–745.

L. GISLÉN, B. SÖDERBERG, C. PETERSON (1992b). Complex scheduling with Potts neural networks. *Neural Computation* **4**, 805–831.

K. GLASMACHER, A. HESS, G. ZIMMERMANN (1991). A genetic algorithm for global improvement of macrocell layouts. *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, IEEE Computer Society Press, Los Alamitos, CA, 306–313.

K. GLASMACHER, G. ZIMMERMANN (1992). Chip assembly in the PLAYOUT VLSI design system. *EURO-DAC '92: European Design Automation Conference; EURO-VHDL '92*, IEEE Computer Society Press, Los Alamitos, CA, 215–221.

C.A. GLASS, C.N. POTTS (1996). A comparison of local search methods for flow shop scheduling. G. LAPORTE, I.H. OSMAN (EDS.). *Metaheuristics in Combinatorial Optimization*, Annals of Operations Research 63, Baltzer, Amsterdam, 489–509.

C.A. GLASS, C.N. POTTS, P. SHADE (1994). Unrelated parallel machine scheduling using local search. *Mathematical and Computer Modelling* **20**, 41–52.

F. GLOVER (1986). Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research* **13**, 533–549.

F. GLOVER (1989). Tabu search: part I. *ORSA Journal on Computing* **1**, 190–206.

F. GLOVER (1990). Tabu search: part II. *ORSA Journal on Computing* **2**, 4–32.

F. GLOVER (1991). *Multilevel tabu search and embedded search neighborhoods for the traveling salesman problem*, Manuscript, School of Business, University of Colorado, Boulder, CO.

F. GLOVER (1992). *Private communication.*

F. GLOVER (1996). Ejection chains, reference structures and alternating path methods for traveling salesman problems. *Discrete Applied Mathematics* **65**, 223–253.

F. GLOVER, M. LAGUNA (1991). Target analysis to improve a tabu search method for machine scheduling. *Arabian Journal for Science and Engineering* **16**, 239–253.

F. GLOVER, M. LAGUNA (1993). Tabu search. C.R. REEVES (ED.). *Modern Heuristic Techniques for Combinatorial Problems*, Blackwell, Oxford, 70–150.

F. GLOVER, E. TAILLARD, M. LAGUNA, D. DE WERRA (EDS.) (1993). *Tabu Search*, Annals of Operations Research 41, Baltzer, Basel.

G. GODBEER (1987). *On the computational complexity of the stable configuration problem for connectionist models*, MSc thesis, Department of Computer Science, University of Toronto, Toronto.

M.X. GOEMANS, D. BERTSIMAS (1991). Probabilistic analysis of the Held Karp lower bound for the Euclidean traveling salesman problem. *Mathematics of Operations Research* **16**, 72–89.

M.X. GOEMANS, D.P. WILLIAMSON (1994). .878-Approximation algorithms for MAX CUT and MAX 2SAT. *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing*, ACM, New York, 422–431.

M.J.E. GOLAY (1972). A class of finite binary sequences with alternate autocorrelation values equal to zero. *IEEE Transactions on Information Theory* **18**, 449–450.

M.J.E. GOLAY (1977). Sieves for low autocorrelation binary sequences. *IEEE Transactions on Information Theory* **23**, 43–51.

M.J.E. GOLAY (1982). The merit factor of long low autocorrelation binary sequences. *IEEE Transactions on Information Theory* **28**, 543–549.

M.J.E. GOLAY (1983). The merit factor of Legendre sequences. *IEEE Transaction on Information Theory* **29**, 934–936.

M.J.E. GOLAY, D.B. HARRIS (1990). A new search for skewsymmetric binary sequences with optimal merit factors. *IEEE Transactions on Information Theory* **36**, 1163–1166.

D.E. GOLDBERG (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Reading, MA.

D.E. GOLDBERG, K. DEB (1991). A comparative analysis of selection schemes used in genetic algorithms. G. RAWLINS (ED.). *Foundations of Genetic Algorithms*, Morgan Kaufmann, San Mateo, CA, 69–93.

D.E. GOLDBERG, R. LINGLE (1985). Alleles, loci, and the traveling salesman problem. J.J. GREFENSTETTE (ED.). *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, Lawrence Erlbaum, Hillsdale, NJ, 154–159.

B.L. GOLDEN, A.A. ASSAD (EDS.) (1988). *Vehicle Routing: Methods and Studies*, North-Holland, Amsterdam.

B.L. GOLDEN, C.C. SKISCIM (1986). Using simulated annealing to solve routing and location problems. *Naval Research Logistics Quarterly* **33**, 261–279.

B.L. GOLDEN, W.R. STEWART (1985). Empirical analysis of heuristics. E.L. LAWLER, J.K. LENSTRA, A.H.G. RINNOOY KAN, D.B. SHMOYS (EDS.). *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, Wiley, Chichester, 207–249.

L.M. GOLDSCHLAGER, R.A. SHAW, J. STAPLES (1982). The maximum flow problem is logspace complete for P. *Theoretical Computer Science* **21**, 105–111.

E. GOLES CHACC, J. OLIVOS (1981). The convergence of symmetric threshold automata. *Information and Control* **51**, 98–104.

E. GOLES CHACC, F. FOGELMAN-SOULIE, D. PELLEGRIN (1985). Decreasing energy functions as a tool for studying threshold networks. *Discrete Applied Mathematics* **12**, 261–277.

T. GONZALEZ, S. SAHNI (1976). Open shop scheduling to minimize finish time. *Journal of the Association for Computing Machinery* **23**, 665–679.

D.J. GOODMAN, T.J. MOULSLEY (1988). Using simulated annealing to design digital transmission codes for analogue sources. *Electronics Letters* **24**, 617–619.

M. GORGES-SCHLEUTER (1989). Asparagos: an asynchronous parallel genetic optimization strategy. J.D. SCHAFFER (ED.). *Proceedings of the Third International Conference on Genetic Algorithms*, Morgan Kaufmann, San Mateo, CA, 422–427.

M. GORGES-SCHLEUTER (1991). *Genetic algorithms and population structures: a massively parallel algorithm*, PhD thesis, Fachbereich Informatik, Universität Dortmund, Dortmund.

R.L. GRAHAM (1966). Bounds for certain multiprocessing anomalies. *Bell System Technical Journal* **45**, 1563–1581.

R.L. GRAHAM (1969). Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics* **17**, 416–429.

R.L. GRAHAM, E.L. LAWLER, J.K. LENSTRA, A.H.G. RINNOOY KAN (1979). Optimization and approximation in deterministic sequencing and scheduling: a survey. P.L. HAMMER, E.L. JOHNSON, B.H. KORTE (EDS.). *Discrete Optimization*, Annals of Discrete Mathematics 5, North-Holland, Amsterdam, 287–326.

R.L. GRAHAM, N.J.A. SLOANE (1985). On the covering radius of codes. *IEEE Transactions on Information Theory* **31**, 385–401.

R.M. GRAY, E.D. KARNIN (1982). Multiple local optima in vector quantizers. *IEEE Transactions on Information Theory* **28**, 256–261.

J.W. GREENE, K.J. SUPOWIT (1986). Simulated annealing without rejecting moves. *IEEE Transactions on Computer-Aided Design* **5**, 221–228.

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

J.J. GREFENSTETTE (1986). Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics* **16**, 122–128.

J.J. GREFENSTETTE, R. GOPAL, B.J. ROSMAITA, D. VAN GUCHT (1985). Genetic algorithms for the traveling salesman problem. J.J. GREFENSTETTE (ED.). *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, Lawrence Erlbaum, Hillsdale, NJ, 160–168.

C. DE GROOT, D. WÜRTZ (1991). Statistical mechanics of low autocorrelation skew-symmetric binary sequences. *Helvetica Physica Acta* **64**, 86–91.

C. DE GROOT, D. WÜRTZ, K.H. HOFFMANN (1989). *Low autocorrelation binary sequences: exact enumeration and optimization by evolutionary strategies*, Technical report IPS 89-09, ETH Zürich, Zürich.

M. GRÖTSCHEL, O. HOLLAND (1991). Solution of large-scale symmetric travelling salesman problems. *Mathematical Programming* **51**, 141–202.

M. GRÖTSCHEL, L. LOVÁSZ, A. SCHRIJVER (1988). *Geometric Algorithms and Combinatorial Optimization*, Springer, Berlin.

L.K. GROVER (1986). A new simulated annealing algorithm for standard cell placement. *IEEE International Conference on Computer-Aided Design: ICCAD-86*, IEEE Computer Society Press, Washington, DC, 378–380.

L.K. GROVER (1992). Local search and the local structure of NP-complete problems. *Operations Research Letters* **12**, 235–243.

J. GU (1994). Efficient local search with search space smoothing: a case study of the traveling salesman problem (TSP). *IEEE Transactions on Systems, Man, and Cybernetics* **24**, 728–735.

M.C. GUPTA, Y.P. GUPTA, A. KUMAR (1993). Minimizing flow time variance in a single machine system using genetic algorithms. *European Journal of Operational Research* **70**, 289–303.

M. GYULASSY, H. HARLANDER (1991). Elastic tracking and neural network algorithms for complex pattern recognition. *Computer Physics Communications* **66**, 31–46.

L. HABSIEGER (1994). Lower bounds for q-ary coverings by spheres of radius one. *Journal of Combinatorial Theory, Series A* **67**, 199–222.

L. HABSIEGER (1997). Binary codes with covering radius one: some new lower bounds. *Discrete Mathematics*, **176**, 115–130

M. HAIMOVICH, A.H.G. RINNOOY KAN (1985). Bounds and heuristics for capacitated routing problems. *Mathematics of Operations Research* **10**, 527–542.

B. HAJEK (1988). Cooling schedules for optimal annealing. *Mathematics of Operations Research* **13**, 311–329.

B. HAJEK, G. SASAKI (1989). Simulated annealing: to cool it or not. *Systems Control Letters* **12**, 443–447.

A. HAKEN (1989). *Connectionist networks that need exponential time to stabilize*, Manuscript.

A. HAKEN, M. LUBY (1988). Steepest descent can take exponential time for symmetric connection networks. *Complex Systems* **2**, 191–196.

N.G. HALL, W. KUBIAK, S.P. SETHI (1991). Earliness–tardiness scheduling problems, II: deviation of completion times about a restrictive common due date. *Operations Research* **39**, 847–856.

M. HALLDORSSON (1995). Approximating discrete collections via local improvements. *Proceedings of the Sixth Annual ACM–SIAM Symposium on Discrete Algorithms*, ACM, New York, and SIAM, Philadelphia, PA, 160–169.

M.W. VAN DER HAM (1988). *Simulated annealing applied in coding theory*, MSc thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven.

H.O. HÄMÄLÄINEN, I.S. HONKALA, M.K. KAIKKONEN, S.N. LITSYN (1993). Bounds for binary multiple covering codes. *Designs, Codes and Cryptography* **3**, 251–275.

H.O. HÄMÄLÄINEN, I.S. HONKALA, S.N. LITSYN, P.R.J. ÖSTERGÅRD (1995a). Bounds for binary codes that are multiple coverings of the farthest-off points. *SIAM Journal on Discrete Mathematics* **8**, 196–207.

H.O. HÄMÄLÄINEN, I.S. HONKALA, S.N. LITSYN, P.R.J. ÖSTERGÅRD (1995b). Football pools: a game for mathematicians. *American Mathematical Monthly* **102**, 579–588.

H.O. HÄMÄLÄINEN, S. RANKINEN (1991). Upper bounds for football pool problems and mixed covering codes. *Journal of Combinatorial Theory, Series A* **56**, 84–95.

P. HAMMER, B. SIMEONE, T. LIEBLING, D. DE WERRA (1988). From linear separability to unimodality: a hierarchy of pseudo-Boolean functions. *SIAM Journal on Algebraic and Discrete Methods* **1**, 174–184.

P. HANSEN (1986). *The steepest ascent mildest descent heuristic for combinatorial programming*, Talk presented at the Congress on Numerical Methods in Combinatorial Optimization, Capri.

F. HARARY (1972). *Graph Theory*, Addison-Wesley, Reading, MA.

A.M.A. HARIRI, C.N. POTTS (1991). Heuristics for scheduling unrelated parallel machines. *Computers & Operations Research* **18**, 323–331.

L.H. HARPER (1966). Optimal numberings and isoperimetric problems on graphs. *Journal of Combinatorial Theory* **1**, 385–393.

R. HAUPT (1989). A survey of priority rule-based scheduling. *OR Spektrum* **11**, 3–16.

S. HAYKIN (1994). *Neural Networks: A Comprehensive Foundation*, Macmillan, New York.

M. HEAP, R. KAPUR, A. MOURAD (1989). *A fault tolerant implementation of the traveling salesman problem*, Technical report, Department of Electrical Engineering and Computer Science, University of Texas, Austin, TX.

D. HEBB (1949). *Organization of Behaviour*, Wiley, New York.

R. HECHT-NIELSEN (1990). *Neurocomputing*, Addison-Wesley, New York.

K. H. HELBIG-HANSEN, J. KRARUP (1974). Improvements of the Held–Karp algorithm for the symmetric traveling salesman problem. *Mathematical Programming* **7**, 87–96.

M. HELD, R.M. KARP (1970). The traveling-salesman problem and minimum spanning trees. *Operations Research* **18**, 1138–1162.

M. HELD, R.M. KARP (1971). The traveling-salesman problem and minimum spanning trees: part II. *Mathematical Programming* **1**, 6–25.

M. HELD, P. WOLFE, H.P. CROWDER (1974). Validation of subgradient optimization. *Mathematical Programming* **6**, 62–88.

B.J. HELLSTROM, L.N. KANAL (1992). Asymmetric mean-field neural networks for multiprocessor scheduling. *Neural Networks* **5**, 671–686.

M. HERDY (1992). Reproductive isolation as strategy parameter in hierarchical organized evolution strategies. R. MÄNNER, B. MANDERICK (EDS.). *Parallel Problem Solving from Nature 2*, North-Holland, Amsterdam, 207–217.

J.W. HERRMANN, C.-Y. LEE (1995). Solving a class scheduling problem with a genetic algorithm. *ORSA Journal on Computing* **7**, 443–452.

A. HERTZ (1991). Tabu search for large scale timetabling problems. *European Journal of Operational Research* **54**, 39–47.

A. HERTZ (1992). Finding a feasible course schedule using tabu search. *Discrete Applied Mathematics* **35**, 255–270.

A. HERTZ, D. DE WERRA (1987). Using tabu search techniques for graph coloring. *Computing* **39**, 345–351.

A. HERTZ, D. DE WERRA (1991). The tabu search metaheuristic: how we used it. *Annals of Mathematics and Artificial Intelligence* **1**, 111–121.

J. HERTZ, A. KROGH, R.G. PALMER (1991). *Introduction to the Theory of Neural Computation*, Addison-Wesley, Redwood City, CA.

A.M. HILL (1993). *Application of genetic algorithms to performance-deriven standard cell gate sizing*, MSc thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL.

G.E. HINTON, T.J. SEJNOWSKI (1986). Learning and relearning in Boltzmann machines. D.E. RUMELHART, J.L. MCCLELLAND, THE PDP RESEARCH GROUP (EDS.). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition 1*, MIT Press, Cambridge, MA.

J.H. HOLLAND (1975). *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, MI.

R. HOLLEY, D. STROOCK (1988). Simulated annealing via Sobolev inequalities. *Communications in Mathematical Physics* **115**, 553–569.

A. HOMAIFAR, S. GUAN, G. LIEPINS (1993). A new approach on the traveling salesman problem. S. FORREST (ED.). *Proceedings of the Fifth International Conference on Genetic Algorithms*, Morgan Kaufmann, San Mateo, CA, 460–466.

I. HONG, A.B. KAHNG, B.-R. MOON (1995). *Improved large-step Markov chain variants for the symmetric TSP*, Report UCLA CSD TR-950035, Computer Science Department, University of Califorina, Los Angeles, CA.

I.S. HONKALA (1991). On (k, t)-subnormal covering codes. *IEEE Transactions on Information Theory* **37**, 1203–1206.

I.S. HONKALA (1994). A new lower bound on codes with covering radius one. *Proceedings of the International Symposium on Information Theory & Its Applications, Part 1*, Institute of Engineers Australia, Crows Nest, NSW, 39–41.

I.S. HONKALA, H.O. HÄMÄLÄINEN (1991). Bounds for abnormal binary codes with covering radius one. *IEEE Transactions on Information Theory* **37**, 372–375.

J.J. HOPFIELD (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences of the United States of America* **79**, 2554–2558.

J.J. HOPFIELD, D.W. TANK (1985). 'Neural' computation of decisions in optimization problems. *Biological Cybernetics* **52**, 141–152.

W.A. HORN (1973). Minimizing average flow time with parallel machines. *Operations Research* **21**, 846–847.

E. HOROWITZ, S. SAHNI (1976). Exact and approximate algorithms for scheduling nonidentical processors. *Journal of the Association for Computing Machinery* **23**, 317–327.

E.S.H. HOU, H.-Y. LI (1991). Task scheduling for flexible manufacturing systems based on genetic algorithms. *Conference Proceedings 1991: IEEE International Conference on Systems, Man, and Cybernetics*, IEEE, New York, 397–402.

E.S.H. HOU, H. REN, N. ANSARI (1990). Efficient multiprocessor scheduling based on genetic algorithms. *16th Annual IEEE Conference on Signal Processing and System Control*, IEEE, New York, 1239–1243.

T.M. HSIEH, H.W. LEONG, C.L. LIU (1987). *Two-dimensional compaction by simulated annealing*, Preprint. Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL.

C.-M. HUANG, R.W. HARRIS (1993). A comparison of several vector quantization codebook generation approaches. *IEEE Transactions on Image Processing* **2**, 108–112.

M.D. HUANG, F. ROMEO, A.L. SANGIOVANNI-VINCENTELLI (1986). An efficient general cooling schedule for simulated annealing. *IEEE International Conference on Computer-Aided Design: ICCAD-86*, IEEE Computer Society Press, Washington, DC, 381–384.

P.J. HUBER (1981). *Robust Statistics*, Wiley, New York.

R. HÜBSCHER, F. GLOVER (1994). Applying tabu search with influential diversification to multiprocessor scheduling. *Computers & Operations Research* **21**, 877–884.

P. HUSBANDS, F. MILL, S. WARRINGTON (1991). Genetic algorithms, production plan optimization and scheduling. H.-P. SCHWEFEL, R. MÄNNER (EDS.). *Parallel Problem Solving from Nature*, Lecture Notes in Computer Science 496, Springer, Berlin, 80–84.

O.H. IBARRA, C.E. KIM (1977). Heuristic algorithms for scheduling independent tasks on nonidentical processors. *Journal of the Association for Computing Machinery* **24**, 280–289.

L. INGBER (1989). Very fast simulated re-annealing. *Mathematical and Computer Modelling* **12**, 967–973.

L. INGBER (1993). Simulated annealing: practice versus theory. *Mathematical and Computer Modelling* **18**, 29–57.

D. ISAACSON, R. MADSEN (1976). *Markov Chains*, Wiley, New York.

J.R. JACKSON (1955). *Scheduling a production line to minimize maximum tardiness*, Research report 43, Management Science Research Project, University of California, Los Angeles, CA.

J.M. JENSEN, H.E. JENSEN, T. HØHOLDT (1991). The merit factor of binary sequences related to difference sets. *IEEE Transactions on Information Theory* **37**, 617–626.

D.W. JEPSEN, C.D. GELATT, JR. (1983). Marco placement by Monte Carlo annealing. *Proceedings IEEE International Conference on Computer Design: VLSI in Computers; ICCD'83*, IEEE Computer Society Press, Silver Spring, MD, 495–498.

R.J. JEROSLOW (1973). The simplex algorithm with the pivot rule of maximizing criterion improvement. *Discrete Mathematics* **4**, 367–378.

L.-M. JIN, S.-P. CHAN (1992). A genetic approach for network partioning. *International Journal of Computer Mathematics* **42**, 47–60.

P. JOG, J.Y. SUH, D. VAN GUCHT (1990). *Parallel genetic algorithms applied to the traveling salesman problem*, Report 314, Computer Science Department, Indiana University, Bloomington, IN.

D.S. JOHNSON (1974). Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences* **9**, 256–278.

D.S. JOHNSON (1990). Local optimization and the traveling salesman problem. M.S. PATERSON (ED.). *Automata, Languages, and Programming*, Lecture Notes in Computer Science 443, Springer, Berlin, 446–461.

D.S. JOHNSON, C.R. ARAGON, L.A. MCGEOCH, C. SCHEVON (1989). Optimization by simulated annealing: an experimental evaluation; part I, graph partitioning. *Operations Research* **37**, 865–892.

D.S. JOHNSON, C.R. ARAGON, L.A. MCGEOCH, C. SCHEVON (1991). Optimization by simulated annealing: an experimental evaluation; part II, graph coloring and number partitioning. *Operations Research* **39**, 378–406.

D.S. JOHNSON, C.R. ARAGON, L.A. MCGEOCH, C. SCHEVON (1997a). *Optimization by simulated annealing: an experimental evaluation; part III, the travelling salesman problem*, forthcoming.

D.S. JOHNSON, J.L. BENTLEY, L.A. MCGEOCH, E.E. ROTHBERG (1997b). *Near-optimal solutions to very large traveling salesman problems*, forthcoming.

D.S. JOHNSON, L.A. MCGEOCH, E.E. ROTHBERG (1996). Asymptotic experimental analysis for the Held–Karp traveling salesman bound. *Proceedings of the Seventh Annual ACM–SIAM Symposium on Discrete Algorithms*, ACM, New York, and SIAM, Philadelphia, PA, 341–350.

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

S.M. JOHNSON (1954). Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly* **1**, 61–68.

A. JOPPE, H.R.A. CARDON, J.C. BIOCH (1990). A neural network for solving the traveling salesman problem on the basis of city adjacency in the tour. *IJCNN: International Joint Conference on Neural Networks*, IEEE, New York, III 961–964.

M. JÜNGER, G. REINELT, G. RINALDI (1995). The traveling salesman problem. M.O. BALL, T.L. MAGNANTI, C.L. MONMA, G.L. NEMHAUSER (EDS.). *Network Models*, Handbooks in Operations Research and Management Science, Volume 7, North-Holland, Amsterdam, 225–330.

G. KALAI (1992). A subexponential randomized simplex algorithm. *Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing*, ACM, New York, 475–482.

H.J.L. KAMPS, J.H. VAN LINT (1967). The football pool problem for 5 matches. *Journal of Combinatorial Theory* **3**, 315–325.

H.J.L. KAMPS, J.H. VAN LINT (1970). A covering problem. *Colloquia Mathematica Societatis János Bolyai* **4**, 679–685.

J.J. KANET (1981). Minimizing the average deviation of jobs completion times about a common due date. *Naval Research Logistics Quarterly* **28**, 643–651.

J.J. KANET, V. SRIDHARAN (1991). Progenitor: a genetic algorithm for production scheduling. *Wirtschaftsinformatik* **33**, 332–336.

D.R. KARGER (1993). Global min-cuts in RNC and other ramifications of a simple mincut algorithm. *Proceedings of the Fourth Annual ACM–SIAM Symposium on Discrete Algorithms*, ACM, New York, and SIAM, Philadelphia, PA, 21–30.

D.R. KARGER, R. MOTWANI (1994). Derandomization through approximation. *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing*, ACM, New York, 497–506.

A.R. KARLIN, E. UPFAL (1988). Parallel hashing: an efficient implementation of shared memory. *Journal of the Association for Computing Machinery* **35**, 876–892.

N. KARMARKAR (1984). A new polynomial time algorithm for linear programming. *Combinatorica* **4**, 373–395.

R.M. KARP (1972). Reducibility among combinatorial problems. R.E. MILLER, J.W. THATCHER (EDS.). *Complexity of Computer Computations*, Plenum Press, New York, 85–103.

R.M. KARP (1977). Probabilistic analysis of partitioning algorithms for the traveling-salesman problem in the plane. *Mathematics of Operations Research* **2**, 209–224.

R.M. KARP, V. RAMACHANDRAN (1990). Parallel algorithms for shared memory machines. J. VAN LEEUWEN (ED.). *Algorithms and Complexity*, Handbook of Theoretical Computer Science, Volume A, Elsevier, Amsterdam, and MIT Press, Cambridge, MA, 869–941.

R.M. KARP, E. UPFAL, A. WIGDERSON (1986). Constructing a perfect matching is in random NC. *Combinatorica* **6**, 35–48.

R.M. KARP, A. WIGDERSON (1985). A fast parallel algorithm for the maximal independent set problem. *Journal of the Association for Computing Machinery* **32**, 762–773.

D.G. KELLY (1981). Some results on random linear programs. *Methods of Operations Research* **40**, 351–355.

W. KERN (1989). A probabilistic analysis of the switching algorithm for the Euclidean TSP. *Mathematical Programming* **44**, 213–219.

W. KERN (1993). On the depth of combinatorial optimization problems. *Discrete Applied Mathematics* **43**, 115–129.

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

G. KESIDIS (1990). Optimal acceptance probability for simulated annealing. *Stochastics and Stochastics Reports* **29**, 221–226.

L.G. KHACHIYAN (1979). A polynomial algorithm for linear programming. *Soviet Mathematics Doklady* **20**, 191–194.

S. KHANNA, R. MOTWANI, M. SUDAN, U. VAZIRANI (1994). On syntactic versus computational views of approximability. *Proceedings 35th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society Press, Los Alamitos, CA, 14–23.

C. KIM, W. KIM, H. SHIN, K. RHEE, H. CHUNG, J. KIM (1993). Combined hierarchical placement algorithm for row-based layouts. *Electronics Letters* **29**, 1508–1509.

Y.-D. KIM (1993). Heuristic for flowshop scheduling problems minimizing mean tardiness. *Journal of the Operational Research Society* **44**, 19–28.

G.A.P. KINDERVATER, J.K. LENSTRA, M.W.P. SAVELSBERGH (1993). Sequential and parallel local search for the time-constrained traveling salesman problem. *Discrete Applied Mathematics* **42**, 211–225.

S. KIRKPATRICK (1982). *Private communication.*

S. KIRKPATRICK (1984). Optimization by simulated annealing: quantitative studies. *Journal of Statistical Physics* **34**, 976–986.

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

S. KIRKPATRICK, G. TOULOUSE (1985). Configuration space and the travelling salesman problem. *Journal de Physique* **46**, 1277–1292.

V. KLEE, P. KLEINSCHMIDT (1987). The *d*-step conjecture and its relatives. *Mathematics of Operations Research* **12**, 718–755.

V. KLEE, G.J. MINTY (1972). How good is the simplex algorithm? O. SHISHA (ED.). *Inequalities III*, Academic Press, New York, 159–175.

Y. KLEIN, S.N. LITSYN, A. VARDY (1995). Two new bounds on the size of binary codes with a minimum distance of three. *Design, Codes and Cryptography* **6**, 219–227.

U. KLEINAU (1993). Two-machine shop scheduling problems with batch processing. *Mathematical and Computer Modelling* **17**, 55–66.

R.M. KLING, P. BANERJEE (1987). ESP: a new standard cell placement package using simulated evolution. *24th ACM/IEEE Design Automation Conference: Proceedings 1987*, IEEE Computer Society Press, Los Alamitos, CA, 60–66.

R.M. KLING, P. BANERJEE (1991). Empirical and theoretical studies of the simulated evolution method applied to standard cell placement. *IEEE Transactions on Computer-Aided Design* **10**, 1303–1315.

M. KNOPS (1993). *Locale zoekmethoden voor het handelsreizigersprobleem met tijdvensters*, MSc thesis, Department of Economics, Erasmus University, Rotterdam (in Dutch).

J. KNOX (1989). *The application of tabu search to the symmetric traveling salesman problem*, PhD thesis, College of Business and Administration, University of Colorado, Boulder, CO.

J. KNOX (1994). Tabu search performance on the symmetric traveling salesman problem. *Computers & Operations Research* **21**, 867–876.

J. KNOX, F. GLOVER (1989). *Comparative testing of traveling salesman heuristics derived from tabu search, genetic algorithms and simulated annealing*, Technical report, Center for Applied Artificial Intelligence, University of Colorado, Boulder, CO.

D.E. KNUTH (1983). *The Art of Computer Programming; Volume 3, Sorting and Searching*, Addison-Wesley, Reading, MA.

S. KOAKUTSU, Y. SUGAI, H. HIRATA (1992). Floorplanning by simulated annealing based on genetic algorithm. *Transactions of the Institute of Electrical Engineers of Japan* **112-C**, 411–416.

H. KODAMA, K. WAKASUGI, M. KASAHARA (1992). A construction of optimum vector quantizers by simulated annealing algorithm. *Electronics & Communications in Japan, Part I: Communications* **75**, 58–65.

W.H. KOHLER, K. STEIGLITZ (1975). Exact, approximate and guaranteed accuracy algorithms for the flow-shop problem $n/2/F/\bar{F}$. *Journal of the Association for Computing Machinery* **22**, 106–114.

T. KOHONEN (1988a). Self-organization via competition, cooperation and categorization applied to extended vehicle routing problems. *IJCNN: International Joint Conference on Neural Networks*, IEEE, New York, I 385–390.

T. KOHONEN (1988b). *Self-Organization and Associative Memory*, Springer, Berlin.

B.H. KORTE (1989). Applications of combinatorial optimization. M. IRI, K. TANABE (EDS.). *Mathematical Programming: Recent Developments and Applications*, Kluwer, Dordrecht, 1–55.

K.-U. KOSCHNICK (1991). Some new constant weight codes. *IEEE Transactions on Information Theory* **37**, 370–371.

K.-U. KOSCHNICK (1993). A new upper bound for the football pool problem for nine matches. *Journal of Combinatorial Theory, Series A* **62**, 162–167.

C. KOULAMAS, S.R. ANTONY, R. JAEN (1994). A survey of simulated annealing applications to operations research problems. *Omega* **22**, 41–56.

P.V. KRAUS, D.A. MLYNSKI (1991). A new annealing strategy for the placement of macrocells. *1991 IEEE International Symposium on Circuits and Systems*, IEEE, New York, 3130–3133.

W. KRAUTH, M. MÉZARD (1989). The cavity method and the travelling-salesman problem. *Europhysics Letters* **8**, 213–218.

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

M.W. KRENTEL (1989). Structure in locally optimal solutions. *30th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society Press, Los Alamitos, CA, 216–222.

M.W. KRENTEL (1990). On finding and verifying locally optimal solutions. *SIAM Journal on Computing* **19**, 742–751.

M.W. KRENTEL (1994). *Private communication.*

M.J. KRONE, K. STEIGLITZ (1974). Heuristic-programming solution of a flowshop-scheduling problem. *Operations Research* **22**, 629–638.

W. KUBIAK (1993). Completion time variance minimization on a single machine is difficult. *Operations Research Letters* **14**, 49–59.

E.S. KUH, T. OHTSUKI (1990). Recent advances in VLSI layout. *Proceedings of the IEEE* **78**, 237–263.

P.J.M. VAN LAARHOVEN (1988). *Theoretical and computational aspects of simulated annealing*, PhD thesis, Econometric Institute, Erasmus University, Rotterdam.

P.J.M. VAN LAARHOVEN, E.H.L. AARTS (1987). *Simulated Annealing: Theory and Applications*, Reidel, Dordrecht.

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

P.J.M. VAN LAARHOVEN, E.H.L. AARTS, J.H. VAN LINT, L.T. WILLE (1989). New upper bounds for the football pool problem for 6, 7 and 8 matches. *Journal of Combinatorial Theory, Series A* **52**, 304–312.

B.J. LAGEWEG, J.K. LENSTRA, A.H.G. RINNOOY KAN (1978). A general bounding scheme for the permutation flow-shop problem. *Operations Research* **26**, 53–67.

M. LAGUNA, J.W. BARNES, F. GLOVER (1991). Tabu search methods for a single-machine scheduling problem. *Journal of Intelligent Manufacturing* **2**, 63–74.

M. LAGUNA, J.W. BARNES, F. GLOVER (1993). Intelligent scheduling with tabu search: an application to jobs with linear delay penalties and sequence-dependent setup costs and times. *Applied Intelligence* **3**, 159–172.

M. LAGUNA, F. GLOVER (1993). Integrating target analysis and tabu search for improved scheduling systems. *Export Systems with Applications* **6**, 287–297.

M. LAGUNA, J.L. GONZÁLEZ VELARDE (1991). A search heuristic for just-in-time scheduling in parallel machines. *Journal of Intelligent Manufacturing* **2**, 253–260.

P. LALANNE, J.C. RODIER, P. CHAVEL, E. BELHAIRE, P.F. GARDA (1993). Optoelectronic devices for Boltzmann machines and simulated annealing. *Optical Engineering* **32**, 1904–1914.

J. LAM (1988). *An efficient simulated annealing schedule*, PhD thesis, Department of Computer Science, Yale University, New Haven, CT.

J. LAM, J.-M. DELOSME (1986). Logic minimization using simulated annealing. *IEEE International Conference on Computer-Aided Design: ICCAD-86*, IEEE Computer Society Press, Washington, DC, 348–351.

J. LAM, J.-M. DELOSME (1988a). *An efficient simulated annealing schedule: implementation and evolution*, Manuscript.

J. LAM, J.-M. DELOSME (1988b). Performance of a new annealing schedule. *25th ACM/IEEE Design Automation Conference: Proceedings 1988*, IEEE Computer Society Press, Los Alamitos, CA, 306–311.

A. LAPEDES, R. FARBER (1987). *Nonlinear signal processing using neural networks: prediction and system modeling*, Los Alamos Report LA-UR 87–2662.

G. LAPORTE (1992). The vehicle routing problem: an overview of exact and approximate algorithms. *European Journal of Operational Research* **59**, 345–358.

G. LAPORTE, Y. NOBERT (1987). Exact algorithms for the vehicle routing problems. S. MARTELLO, G. LAPORTE, M. MINOUX, C.C. RIBEIRO (EDS.). *Surveys in Combinatorial Optimization*, Annals of Discrete Mathematics 31, North-Holland, Amsterdam, 147–184.

E.L. LAWLER (1973). Optimal sequencing of a single machine subject to precedence constraints. *Management Science* **19**, 544–546.

E.L. LAWLER (1976). *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart, and Winston, New York.

E.L. LAWLER (1977). A 'pseudopolynomial' algorithm for sequencing jobs to minimize total tardiness. P.L. HAMMER, E.L. JOHNSON, B.H. KORTE, G.L. NEMHAUSER (EDS.). *Studies in Integer Programming*, Annals of Discrete Mathematics 1, North-Holland, Amsterdam, 331–342.

E.L. LAWLER (1978). Sequencing jobs to minimize total weighted completion time subject to precedence constraints. B. ALSPACH, P. HELL, D.J. MILLER (EDS.). *Algorithmic Aspects of Combinatorics*, Annals of Discrete Mathematics 2, North-Holland, Amsterdam, 75–90.

E.L. LAWLER, J.K. LENSTRA, A.H.G. RINNOOY KAN (1981, 1982). Minimizing maximum lateness in a two-machine open shop. *Mathematics of Operations Research* **6**, 153–158. Erratum. *Mathematics of Operations Research* **7**, 635.

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

E.L. LAWLER, J.K. LENSTRA, A.H.G. RINNOOY KAN, D.B. SHMOYS (1993). Sequencing and scheduling: algorithms and complexity. S.C. GRAVES, A.H.G. RINNOOY KAN, P.H. ZIPKIN (EDS.). *Logistics of Production and Inventory*, Handbooks in Operations Research and Management Science, Volume 4, North-Holland, Amsterdam, 445–522.

E.L. LAWLER, J.M. MOORE (1969). A functional equation and its application to resource allocation and sequencing problems. *Management Science* **16**, 77–84.

S. LAWRENCE (1984). *Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques (supplement)*, Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, PA.

B.W. LEE, B.J. SHEU (1991). Hardware annealing in electronic neural networks. *IEEE Transactions on Circuits and Systems* **38**, 134–141.

C.Y. LEE (1961). An algorithm for path connection and its applications. *IRE Transactions on Electronic Computers* **10**, 346–365.

C.Y. LEE, S.J. KIM (1995). Parallel genetic algorithms for the earliness–tardiness job scheduling problem with general penalty weights. *Computers & Industrial Engineering* **28**, 231–243.

J. LEE (1993). New Monte Carlo algorithm: entropic sampling. *Physical Review Letters* **71**, 211–214.

J. LEE (1995). *Private communication.*

J. LEE, M.Y. CHOI (1994). Optimization by multicanonical annealing and the traveling salesman problem. *Physical Review E* **50**, R651–R654.

J. VAN LEEUWEN, A.A. SCHOONE (1980). *Untangling a traveling salesman tour in the plane*, Report RUU-CS-80-11, Department of Computer Science, Utrecht University, Utrecht.

T. LENGAUER (1990). *Combinatorial Algorithms for Integrated Circuit Layout*, Teubner, Stuttgart, and Wiley, Chichester.

J.K. LENSTRA, A.H.G. RINNOOY KAN (1978). Complexity of scheduling under precedence constraints. *Operations Research* **26**, 22–35.

J.K. LENSTRA, A.H.G. RINNOOY KAN, P. BRUCKER (1977). Complexity of machine scheduling problems. P.L. HAMMER, E.L. JOHNSON, B.H. KORTE, G.L. NEMHAUSER (EDS.). *Studies in Integer Programming*, Annals of Discrete Mathematics 1, North-Holland, Amsterdam, 343–362.

J.K. LENSTRA, D.B. SHMOYS, É. TARDOS (1990). Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming* **46**, 259–271.

H.W. LEONG, C.L. LIU (1987). Algorithms for permutation channel routing. *Integration, The VLSI Journal* **4**, 17–45.

H.W. LEONG, D.F. WONG, C.L. LIU (1985). A simulated-annealing channel router. *IEEE International Conference on Computer-Aided Design: ICCAD-85*, IEEE Computer Society Press, Washington, DC, 226–229.

D. LI, W. CHEN (1994). New lower bounds for binary covering codes. *IEEE Transactions on Information Theory* **40**, 1122–1129.

T.M. LIEBLING (1981). *Note on a vertex following algorithm with a Poisson distributed number of iterations*, Technical report, Department of Mathematics, Ecole Polytechnique Fédérale de Lausanne, Lausanne.

L.W. LIGHT, P. ANDERSON (1993). Designing better keyboards via simulated annealing. *AI Expert* **9**, 20–27.

A. LIM, Y.M. CHEE (1991). Graph partitioning using tabu search. *1991 IEEE International Symposium on Circutis and Systems*, IEEE, New York, 1164–1167.

A. LIM, Y.M. CHEE, C.-T. WU (1991). Performance driven placement with global routing for macro cells. *Proceedings of the Second Great Lakes Symposium on VLSI*, 35–41.

F.-T. LIN, C.-Y. KAO, C.-C. HSU (1994). Applying the genetic approach to simulated annealing in solving some NP-hard problems. *IEEE Transactions on Systems, Man, and Cybernetics* **23**, 1752–1767.

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

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

Y.-L. LIN, Y.-C. HSU, F.-S. TSAI (1989). SILK: a simulated evolution router. *IEEE Transactions on Computer-Aided Design* **8**, 1108–1114.

P.O. LINDBERG, S. ÓLAFSSON (1984). On the length of simplex paths: the assignment case. *Mathematical Programming* **30**, 243–260.

Y. LINDE, A. BUZO, R.M. GRAY (1980). An algorithm for vector quantizer design. *IEEE Transactions on Communications* **28**, 84–95.

J.H. VAN LINT (1982). *Introduction to Coding Theory*, Springer, Berlin.

J.H. VAN LINT (1989). Recent results on covering problems. T. MORA (ED.). *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, Lecture Notes in Computer Science 357, Springer, Berlin, 7–21.

J.H. VAN LINT, JR. (1988). *Covering radius problems*, MSc thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven.

J. LIPSCOMB (1987). *On the computational complexity of finding a connectionist model's stable state of vectors*, MSc thesis, Department of Computer Science, University of Toronto, Toronto.

S.N. LITSYN, A. VARDY (1994). The uniqueness of the Best code. *IEEE Transactions on Information Theory* **40**, 1693–1698.

X.Z. LIU, A. SAKAMOTO, T. SHIMAMOTO (1993). Restrictive channel routing with evolution programs. *Transactions on Fundamentals of Electronics Communication and Computer Science* **76A**, 1738–1745.

D.C. LLEWELLYN, C.A. TOVEY (1993). Dividing and conquering the square. *Discrete Applied Mathematics* **43**, 131–153.

D.C. LLEWELLYN, C.A. TOVEY, M.A. TRICK (1989, 1993). Local optimization on graphs. *Discrete Applied Mathematics* **23**, 157–178. Erratum. *Discrete Applied Mathematics* **46**, 93–94.

S.P. LLOYD (1982). Least squares quantization in PCM. *IEEE Transactions on Information Theory* **28**, 129–136.

Z.-P. LO, B. BAVARIAN (1992). Optimization of job scheduling on parallel machines by simulated annealing algorithms. *Expert Systems with Applications* **4**, 323–328.

A. LOBSTEIN, V. PLESS (1994). The length function: a revised table. G. COHEN, S. N. LITSYN A. LOBSTEIN, G. ZÉMOR (EDS.). *Algebraic Coding*, Lecture Notes in Computer Science 781, Springer, Berlin, 51–55.

M. LOEBL (1991). *Maximal cuts*, Technical report 91–065, University of Bielefeld, Bielefeld.

C.-K. LOOI (1992). Neural network methods in combinatorial optimization. *Computers & Operations Research* **19**, 191–208.

M. LUBY (1986). A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing* **15**, 1036–1053.

G. LUEKER (1975). *Two polynomial complete problems in nonnegative integer programming*, Manuscript TR-178, Department of Computer Science, Princeton University, Princeton, NJ.

G. LUEKER (1976). Manuscript, Department of Computer Science, Princeton University, Princeton, NJ.

M. LUNDY, A. MEES (1986). Convergence of an annealing algorithm. *Mathematical Programming* **34**, 111–124.

F.J. MACWILLIAMS, N.J.A. SLOANE (1977). *The Theory of Error-Correcting Codes*, North-Holland, Amsterdam.

K.-T. MAK, A.J. MORTON (1993). A modified Lin–Kernighan traveling salesman heuristic. *Operations Research Letters* **13**, 127–132.

M. MALEK, M. GURUSWAMY, M. PANDYA (1989). Serial and parallel simulated annealing and tabu search algorithms for the traveling salesman problem. F. GLOVER, H.J. GREENBERG (EDS.). *Linkages with Artificial Intelligence*, Annals of Operations Research 21, Baltzer, Basel, 59–84.

S. MALLELA, L.K. GROVER (1988). Clustering based simulated annealing for standard cell placement. *25th ACM/IEEE Design Automation Conference: Proceedings 1988*, IEEE Computer Society Press, Los Alamitos, CA 312–317.

O.C. MARTIN, S.W. OTTO (1996). Combining simulated annealing with local search heuristics. G. LAPORTE, I.H. OSMAN (EDS.). *Metaheuristics in Combinatorial Optimization*, Annals of Operations Research 63, Baltzer, Amsterdam, 57–75.

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

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

A.J. MASON (1992). *Genetic algorithms and scheduling problems*, PhD thesis, Department of Management Sciences, University of Cambridge, Cambridge, UK.

A.J. MASON, E.J. ANDERSON (1991). Minimizing flow time on a single machine with job classes and setup times. *Naval Research Logistics* **38**, 333–350.

J. MATOUSEK, M. SHARIR, E. WELZL (1992). A subexponential bound for linear programming. *Proceedings of the Eighth Annual ACM Symposium on Computational Geometry*, ACM, New York, 1–8.

H. MATSUO, C.J. SUH, R.S. SULLIVAN (1987). *A controlled search simulated annealing method for the single machine weighted tardiness problem*, Working paper 87-12-2, Department of Management, University of Texas, Austin, TX.

H. MATSUO, C.J. SUH, R.S. SULLIVAN (1988). *A controlled search simulated annealing method for the general jobshop scheduling problem*, Working paper 03-04-88, Graduate School of Business, University of Texas, Austin, TX.

Y. MATSUYAMA (1991). Self-organization via competition, cooperation and categorization applied to extended vehicle routing problems. *IJCNN-91-Seattle: International Joint Conference on Neural Networks*, IEEE, New York, I 385–390.

J. MAX (1960). Quantizing for minimum distortion. *IEEE Transactions on Information Theory* **6**, 7–12.

N. METROPOLIS, A. ROSENBLUTH, M. ROSENBLUTH, A. TELLER, E. TELLER (1953). Equation of state calculations by fast computing machines. *Journal of Chemical Physics* **21**, 1087–1092.

Z. MICHALEWICZ, C.Z. JANIKOW (1991). Handling constraints in genetic algorithms. R.K. BELEW, L.B. BOOKER (EDS.). *Proceedings of the Fourth International Conference on Genetic Algorithms*, Morgan Kaufmann, San Mateo, CA, 151–157.

D. MITRA, F. ROMEO, A.L. SANGIOVANNI-VINCENTELLI (1986). Convergence and finite-time behavior of simulated annealing. *Advances in Applied Probability* **18**, 747–771.

J. MITTENTHAL, M. RAGHAVACHARI, A.I. RANA (1993). A hybrid simulated annealing approach for single machine scheduling problems with non-regular penalty functions. *Computers & Operations Research* **20**, 103–111.

S. MOHAN, P. MAZUMDER (1992). Wolverines: standard cell placement on a network of workstations. *EURO-DAC'92: European Design Automation Conference; EURO-VHDL'92*, IEEE Computer Society Press, Los Alamitos, CA, 46–51.

C.L. MONMA, C.N. POTTS (1989). On the complexity of scheduling with batch setup times. *Operations Research* **37**, 798–804.

J.M. MOORE (1968). An *n* job, one machine sequencing algorithm for minimizing the number of late jobs. *Management Science* **15**, 102–109.

R.C. MOSTELLER (1986). *Monte Carlo methods for 2-D compaction*, PhD thesis, Califorina Institute of Technology, Pasadena, CA.

H. MÜHLENBEIN (1989). Parallel genetic algorithm, population dynamics and combinatorial optimization. J.D. SCHAFFER (ED.). *Proceedings of the Third International Conference on Genetic Algorithms*, Morgan Kaufmann, San Mateo, CA, 416–421.

H. MÜHLENBEIN (1991a). Evolution in time and space: the parallel genetic algorithm. G. RAWLINS (ED.). *Foundation of Genetic Algorithms*, Morgan Kaufmann, San Mateo, CA, 316–337.

H. MÜHLENBEIN (1991b). Asynchronous parallel search by the parallel genetic algorithm. *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, IEEE Computer Society Press, Los Alamitos, CA, 526–533.

H. MÜHLENBEIN (1992a). Parallel genetic algorithms in combinatorial optimization. O. BALCI, R. SHARDA, S. ZENIOS (EDS.). *Computer Science and Operations Research*, Pergamon Press, New York, 441–456.

H. MÜHLENBEIN (1992b). How genetic algorithms really work: mutation and hill-climbing. R. MÄNNER, B. MANDERICK (EDS.). *Parallel Problem Solving form Nature 2*, North-Holland, Amsterdam, 15–26.

H. MÜHLENBEIN (1995). *Private communication.*

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

H. MÜHLENBEIN, J. KINDERMANN (1989). The dynamics of evolution and learning: towards genetic neural networks. R. PFEIFFER (ED.). *Connectionism in Perspective*, North-Holland, Amsterdam, 173–198.

H. MÜHLENBEIN, D. SCHLIERKAMP-VOOSEN (1993). Predictive models for the breeder genetic algorithm I: continuous parameter optimization. *Evolutionary Computation* **1**, 25–49.

H. MÜHLENBEIN, D. SCHLIERKAMP-VOOSEN (1994a). The science of breeding and its application to the breeder genetic algorithm. *Evolutionary Computation* **1**, 335–360.

H. MÜHLENBEIN, D. SCHLIERKAMP-VOOSEN (1994b). The theory of breeding and the breeder genetic algorithm. J. STENDER, E. HILLEBRAND, J. KINGDON (EDS.). *Genetic Algorithms in Optimization, Simulation and Modelling*, Frontiers in Artificial Intelligence Applications, IOS Press, Amsterdam, 27–64.

I. NAGAHARA, T. GAO, Y. KOMADA, Y. GAO, C.L. LIU (1989). Cooperative simulated annealing algorithm for VLSI placement problems. *Proceedings of the International Conference on Computer-Aided Design & Computer Graphics*, Beijing, 512–518.

H.N. NAGARAJA (1982). Selection differentials. E. KOTZ, N.L. JOHNSON, C.B. READ (EDS.). *Encyclopedia of Statistical Science*, Wiley, New York, 334–336.

T. NAGLYAKI (1992). *Introduction to Theoretical Population in Genetics*, Springer, Berlin.

S. NAHAR, S. SAHNI, E. SHRAGOWITZ (1985). Experiments with simulated annealing. *22nd ACM/IEEE Design Automation Conference: Proceedings 1985*, IEEE Computer Society Press, Los Alamitos, CA, 748–752.

R. NAKANO, T. YAMADA (1991). Conventional genetic algorithm for job shop problems. R.K. BELEW, L.B. BOOKER (EDS.). *Proceedings of the Fourth International Conference on Genetic Algorithms*, Morgan Kaufmann, San Mateo, CA, 474–479.

M. NAWAZ, E.E. ENSCORE, JR., I. HAM (1983). A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. *Omega* **11**, 91–95.

G.L. NEMHAUSER, L.A. WOLSEY (1988). *Integer and Combinatorial Optimization*, Wiley, New York.

J. VON NEUMANN, O. MORGENSTERN (1994). *Theory of Games and Economic Behavior*, Princeton University Press, Princeton, NJ.

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

T.A.J. NICHOLSON (1971). A method for optimizing permutation problems and its industrial applications. D.J.A. WELSH (ED.). *Combinatorial Mathematics and Its Applications*, Academic Press, London, 201–217.

E. NOWICKI, C. SMUTNICKI (1996a). A fast taboo search algorithm for the job shop problem. *Management Science* **42**, 797–813.

E. NOWICKI, C. SMUTNICKI (1996b). A fast tabu search algorithm for the permutation flow-shop problem. *European Journal of Operational Research* **91**, 160–175.

K.J. NURMELA (1993). *Constructing combinatorial designs by local search*, MSc thesis, Research report A27, Digital Systems Laboratory, Helsinki University of Technology, Espoo.

K.J. NURMELA (1995). *Constructing spherical codes by global optimization methods*, Research report A32, Digital Systems Laboratory, Helsinki University of Technology, Espoo.

K.J. NURMELA, P.R.J. ÖSTERGÅRD (1993a). *Constructing covering designs by simulated annealing*, Technical report B10, Digital Systems Laboratory, Helsinki University of Technology, Espoo.

K.J. NURMELA, P.R.J. ÖSTERGÅRD (1993b). Upper bounds for covering designs by simulated annealing. *Congressus Numerantium* **96**, 93–111.

F.A. OGBU, D.K. SMITH (1990). The application of the simulated annealing algorithm to the solution of the $n|m|C_{max}$ flowshop problem. *Computers & Operations Research* **17**, 243–253.

F.A. OGBU, D.K. SMITH (1991). Simulated annealing for the permutation flowshop problem. *Omega* **19**, 64–67.

M. O'HEIGEARTAIGH, J.K. LENSTRA, A.H.G. RINNOOY KAN (EDS.) (1985). *Combinatorial Optimization: Annotated Bibliographies*, Wiley, Chichester.

M. OHLSSON, C. PETERSON, B. SÖDERBERG (1993). Neural networks for optimization problems with inequality constraints: the knapsack problem. *Neural Computation* **5**, 331–339.

M. OHLSSON, C. PETERSON, A.L. YUILLE (1992). Track finding with deformable templates: the elastic arms approach. *Computer Physics Communications* **71**, 77–98.

I.M. OLIVER, D.J. SMITH, J.R.C. HOLLAND (1987). A study of permutation crossover operators on the traveling salesman problem. J.J. GREFENSTETTE (ED.). *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, Lawrence Erlbaum, Hillsdale, NJ, 224–230.

H.L. ONG, J.B. MOORE (1984). Worst-case analysis of two travelling salesman heuristics. *Operations Research Letters* **2**, 273–277.

I. OR (1976). *Traveling salesman-type combinatorial problems and their relation to the logistics of regional blood banking*, PhD thesis, Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, IL.

J.B. ORLIN (1985). On the simplex algorithm for networks and generalized networks. *Mathematical Programming Studies* **24**, 166–178.

I.H. OSMAN (1991). *Metastrategy simulated annealing and tabu search algorithms for combinatorial optimization problems*, PhD thesis, Management School, Imperial College, London.

I.H. OSMAN (1993). Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem. F. GLOVER, E. TAILLARD, M. LAGUNA, D. DE WERRA (EDS.). *Tabu Search*, Annals of Operations Research 41, Baltzer, Basel, 421–451.

I.H. OSMAN, G. LAPORTE (1996). Metaheuristics: a bibliography. G. LAPORTE, I.H. OSMAN (EDS.). *Metaheuristics in Combinatorial Optimization*, Annals of Operations Research 63, Baltzer, Amsterdam, 513–623.

I.H. OSMAN, C.N. POTTS (1989). Simulated annealing for permutation flow-shop scheduling. *Omega* **17**, 551–557.

W. OSMAN (1987). *Two-dimensional compaction of abstract layouts with statical cooling*, MSc thesis, Philips Research Laboratories, Eindhoven.

P.R.J. ÖSTERGÅRD (1991a). A new binary code of length 10 and covering radius 1. *IEEE Transaction on Information Theory* **37**, 179–180.

P.R.J. ÖSTERGÅRD (1991b). Upper bounds for q-ary covering codes. *IEEE Transactions on Information Theory* **37**, 660–664, 1738.

P.R.J. ÖSTERGÅRD (1992). Further results on (k, t)-subnormal covering codes. *IEEE Transactions on Information Theory* **38**, 206–210.

P.R.J. ÖSTERGÅRD (1993). *Construction methods for covering codes*, PhD thesis, Research report A25, Digital Systems Laboratory, Helsinki University of Technology, Espoo.

P.R.J. ÖSTERGARD (1994a). Construction methods for mixed covering codes. M. GYLLENBERG, L.E. PERSSON (EDS.). *Analysis, Algebra, and Computers in Mathematical Research*, Marcel Dekker, New York, 387–408.

P.R.J. ÖSTERGARD (1994b). New upper bounds for the football pool problem for 11 and 12 matches. *Journal of Combinatorial Theory, Series A* **67**, 161–168.

P.R.J. ÖSTERGARD (1995). New multiple covering codes by tabu search. *Australasian Journal of Combinatorics* **12**, 145–155.

P.R.J. ÖSTERGARD (1997). Constructing covering codes by tabu search. *Journal of Combinatorial Designs* **5**, 71–80.

P.R.J. ÖSTERGARD (1998). New constructions for q-ary covering codes. *Ars Combinatoria*, in press.

P.R.J. ÖSTERGARD, H.O. HÄMÄLÄINEN (1997). A new table of binary/ternary mixed covering codes. *Design, Codes and Cryptography* **11**, 151–178.

P.R.J. ÖSTERGARD, M.K. KAIKKONEN (1998). New upper bounds for binary covering codes. *Discrete Mathematics* **178**, 165–179.

R.H.J.M. OTTEN (1982). Automatic floorplan design. *ACM IEEE Nineteenth Design Automation Conference: Proceedings*, IEEE Computer Society Press, Los Alamitos, CA 261–267.

R.H.J.M. OTTEN, L.P.P.P. VAN GINNEKEN (1984). Floorplan design using simulated annealing. *IEEE International Conference on Computer-Aided Design: ICCAD-84*, IEEE, New York, 96–98.

M. PADBERG, G. RINALDI (1987). Optimization of a 532-city symmetric traveling salesman problem by branch and cut. *Operations Research Letters* **6**, 1–7.

M. PADBERG, G. RINALDI (1991). A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review* **33**, 60–100.

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

C.H. PAPADIMITRIOU (1992). The complexity of the Lin–Kernighan heuristic for the traveling salesman problem. *SIAM Journal on Computing* **21**, 450–465.

C.H. PAPADIMITRIOU, A.A. SCHÄFFER, M. YANNAKAKIS (1990). On the complexity of local search. *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing*, ACM, New York, 438–445.

C.H. PAPADIMITRIOU, K. STEIGLITZ (1977). On the complexity of local search for the traveling salesman problem. *SIAM Journal on Computing* **6**, 76–83.

C.H. PAPADIMITRIOU, K. STEIGLITZ (1978). Some examples of difficult traveling salesman problems. *Operations Research* **26**, 434–443.

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

C.H. PAPADIMITRIOU, U.V. VAZIRANI (1984). On two geometric problems related to the travelling salesman problem. *Journal of Algorithms* **5**, 231–246.

C.H. PAPADIMITRIOU, M. YANNAKAKIS (1991). Optimization, approximation and complexity classes. *Journal of Computer and System Sciences* **43**, 425–450.

C.H. PAPADIMITRIOU, M. YANNAKAKIS (1993). The traveling salesman problem with distances one and two. *Mathematics of Operations Research* **18**, 1–11.

I. PARBERRI (1990). A primer on the complexity theory of neural networks. R.B. BANERJI (ED.). *A Sourcebook on Formal Techniques in Artificial Intelligence*, North-Holland, Amsterdam, 217–268.

P.M. PARDALOS, K.A. MURTY, T.P. HARRISON (1993). A computational comparison of local search heuristics for solving quadratic assignment problems. *Informatica* **4**, 172–187.

A.G. PERCUS, O.C. MARTIN (1996). Finite size and dimensional dependence in the Euclidean traveling salesman problem. *Physical Review Letters* **76**, 1188–1191.

E. PESCH (1993). *Machine learning by schedule decomposition*, Working paper, Faculty of Economics and Business Administration, University of Limburg, Maastricht.

C. PETERSON (1989). Track finding with neural networks. *Nuclear Instruments and Methods A* **279**, 537–545.

C. PETERSON (1990a). Parallel distributed approaches to combinatorial optimization: benchmark studies on traveling salesman problem. *Neural Computation* **2**, 261–269.

C. PETERSON (1990b). Neural networks and high energy physics. D. PERRET-GALLIX, W. WOIJCIK (EDS.). *Proceedings of the International Workshop on Software Engineering, Artificial Intelligence and Expert Systems for High Energy and Nuclear Physics, Lyon Villeurbanne, March 1990*, Editions du CRNS, Paris, 465–480.

C. PETERSON, J.R. ANDERSON (1988). Neural networks and NP-complete optimization problems: a performance study on the graph bisection problem. *Complex Systems* **2**, 59–89.

C. PETERSON, E. HARTMAN (1989). Explorations of the mean field theory learning algorithm. *Neural Networks* **2**, 475–494.

C. PETERSON, B. SÖDERBERG (1989). A new method for mapping optimization problems onto neural networks. *International Journal of Neural Systems* **1**, 3–22.

M. PIRLOT (1992). General local search heuristics in combinatorial optimization: a tutorial. *Belgian Journal of Operations Research, Statistics and Computer Science* **32**, 8–67.

L.K. PLATZMAN, J.J. BARTHOLDI III (1989). Spacefilling curves and the planar travelling salesman problem. *Journal of the Association for Computing Machinery* **36**, 719–737.

J. PLESNIK (1986). Bad examples of the metric traveling salesman problem. *Acta Mathematica Universitatis Comenianae* **48/49**, 203–207.

S. POLJAK (1995). Integer linear programs and local search for max-cut. *SIAM Journal on Computing* **24**, 822–839.

M.E. POSNER (1985). Minimizing weighted completion times with deadlines. *Operations Research* **33**, 562–574.

C.N. POTTS (1980). An adaptive branching rule for the permutation flow-shop problem. *European Journal of Operational Research* **5**, 19–25.

C.N. POTTS (1985a). A Lagrangean based branch and bound algorithm for single machine sequencing with precedence constraints to minimize total weighted completion time. *Management Science* **31**, 1300–1311.

C.N. POTTS (1985b). Analysis of a linear programming heuristic for scheduling unrelated parallel machines. *Discrete Applied Mathematics* **10**, 155–164.

C.N. POTTS, S.L. VAN DE VELDE (1995). *Dynasearch: iterative local improvement by dynamic programming; part I, the traveling salesman problem*, Working paper LPOM-9511, Faculty of Mechanical Engineering, University of Twente, Enschede.

C.N. POTTS, L.N. VAN WASSENHOVE (1982). A decomposition algorithm for the single machine total tardiness problem. *Operations Research Letters* **1**, 177–181.

C.N. POTTS, L.N. VAN WASSENHOVE (1983). An algorithm for single machine sequencing with deadlines to minimize total weighted completion time. *European Journal of Operational Research* **12**, 379–387.

C.N. POTTS, L.N. VAN WASSENHOVE (1985). A branch and bound algorithm for the total weighted tardiness problem. *Operations Research* **33**, 363–377.

C.N. POTTS, L.N. VAN WASSENHOVE (1988). Algorithms for scheduling a single machine to minimize the weighted number of late jobs. *Management Science* **34**, 843–858.

C.N. POTTS, L.N. VAN WASSENHOVE (1991). Single machine tardiness sequencing heuristics. *IIE Transactions* **23**, 346–354.

J.-Y. POTVIN (1993). The traveling salesman problem: a neural network perspective. *ORSA Journal on Computing* **5**, 328–347.

J.-Y. POTVIN (1996). Genetic algorithms for the traveling salesman problem. G. LAPORTE, I.H. OSMAN (EDS.). *Metaheuristics in Combinatorial Optimization*, Annals of Operations Research 63, Baltzer, Amsterdam, 339–370.

J.-Y. POTVIN, S. BENGIO (1996). The vehicle routing problem with time windows; part II: genetic search. *INFORMS Journal on Computing* **8**, 165–172.

J.-Y. POTVIN, T. KERVAHUT, B.-L. GARCIA, J.-M. ROUSSEAU (1996). The vehicle routing problem with time windows; part I: tabu search. *INFORMS Journal on Computing* **8**, 158–164.

B. PREAS, M. LORENZETTI (1988). *Physical Design Automation of VLSI Systems*, Benjamin Cummings, Menlo Park, CA.

W.P. PRESS, B.P. FLANNERY, S.A. TEUKOLSKY, W.T. VETTERING (1986). *Numerical Recipes: The Art of Scientific Computing*, Cambridge University Press, Cambridge.

H.N. PSARAFTIS (1983). *k*-Interchange procedures for local search in a precedence-constrained routing problem. *European Journal of Operational Research* **13**, 391–402.

V.M. PUREZA, P.M. FRANÇA (1991). *Vehicle routing problems via tabu search metaheuristic*, Publication CRT-747, Centre de recherche sur les transports, Université de Montréal, Montréal.

S. RAJASEKARAN, J.H. REIF (1992). Nested annealing: a provable improvement to simulated annealing. *Theoretical Computer Science* **99**, 157–176.

C.R. RAO (1973). *Linear Statistical Inference and Its Application*, Wiley, New York.

C.P. RAVIKUMAR, L.M. PATNAIK (1987). Parallel placement by simulated annealing. *Proceedings 1987 IEEE International Conference on Computer Design: VLSI in Computers & Processors; ICCD '87*, IEEE Computer Society Press, Los Alamitos, 91–94.

I. RECHENBERG (1973). *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Information*, Fromman, Freiburg.

C.R. REEVES (ED.) (1993a). *Modern Heuristic Techniques for Combinatorial Optimization*, Blackwell, Oxford.

C.R. REEVES (1993b). Improving the efficiency of tabu search for machine sequencing problems. *Journal of the Operational Research Society* **44**, 375–382.

C.R. REEVES (1995). A genetic algorithm for flowshop sequencing. *Computers & Operations Research* **22**, 5–13.

G. REINELT (1991). TSPLIB: a traveling salesman problem library. *ORSA Journal on Computing* **3**, 376–384.

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

G. REINELT (1994). *The Traveling Salesman: Computational Solutions for TSP Applications*, Lecture Notes in Computer Science 840, Springer, Berlin.

S. REITER, G. SHERMAN (1965). Discrete optimizing. *Journal of the Society for Industrial and Applied Mathematics* **13**, 864–889.

W.T. RHEE, M. TALAGRAND (1988). A sharp deviation inequality for the stochastic traveling salesman problem. *Annals of Probability* **17**, 1–8.

J.T. RICHARDSON, M. PALMER, G.E. LIEPINS, M. HILLIARD (1989). Some guidelines for genetic algorithms with penalty functions. J.D. SCHAFFER (ED.). *Proceedings of the Third International Conference on Genetic Algorithms*, Morgan Kaufmann, San Mateo, CA, 191–197.

G. RINGEL (1965). Das Geschlecht des vollständigen Paaren Graphen. *Abhandlungen aus dem mathematisches Seminar der Universität Hamburg 28*, 139–150 (in German).

A.H.G. RINNOOY KAN, B.J. LAGEWEG, J.K. LENSTRA (1975). Minimizing total costs in one-machine scheduling. *Operations Research* **23**, 908–927.

R.L. RIVEST, C.M. FIDUCCIA (1982). A 'greedy' channel router. *ACM IEEE Nineteenth Design Automation Conference: Proceedings*, IEEE Computer Society Press, Los Alamitos, CA, 418–424.

N. ROBERTSON, P.D. SEYMOUR (1991). Graph minors X: obstructions to tree-decomposition. *Journal of Combinatorial Theory, Series B* **52**, 153–190.

Y. ROCHAT, F. SEMET (1994). A tabu search approach for delivering pet food and flour in Switzerland. *Journal of the Operational Research Society* **45**, 1233–1246.

V. RÖDL, C.A. TOVEY (1987). Multiple optima in local search. *Journal of Algorithms* **8**, 250–259.

A. ROHE (1995). *Private communication.*

F. ROMEO, A.L. SANGIOVANNI-VINCENTELLI (1991). A theoretical framework for simulated annealing. *Algorithmica* **6**, 302–345.

R. RÖPLING-LENHART (1988). *Über die Länge von Kreiscodierungen*, PhD thesis, Abteilung Mathematik, Universität Dortmund, Dortmund (in German).

J.S. ROSE, D.R. BLYTHE, W.M. SNELGROVE, Z.G. VRANESIC (1986). Fast, high quality VLSI placement on an MIMD multiprocessor. *IEEE International Conference on Computer-Aided Design: ICCAD-86*, IEEE Computer Society Press, Washington, DC, 42–45.

D.J. ROSENKRANTZ, R.E. STEARNS, P.M. LEWIS II (1977). An analysis of several heuristics for the traveling salesman problem. *SIAM Journal on Computing* **6**, 563–581.

S. ROSS (1982). A simple heuristic approach to simplex efficiency. *European Journal of Operational Research* **9**, 344–346.

A. ROSSI (1990). A simulated annealing channel routing algorithm. *Calcolo* **27**, 279–290.

Y. ROSSIER, M. TROYON, T.M. LIEBLING (1986). Probabilistic exchange algorithms and Euclidean traveling salesman problems. *OR Spektrum* **8**, 151–164.

B. ROY, B. SUSSMANN (1964). *Les problèmes d' ordonnancement avec contraintes disjonctives*, Note DS No. 9 bis, SEMA, Paris (in French).

D.E. RUMELHART, J.L. MCCLELLAND (EDS.) (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Volume 1, MIT Press, Cambridge, MA.

Y. SAAB, V. RAO (1989). An evolution-based approach to partitioning ASIC systems. *26th ACM/IEEE Design Automation Conference: Proceedings 1989*, IEEE Computer Society Press, Los Alamitos, CA, 767–770.

S.N. SAHA, S. MOHAN, K.S. RAGHUNATHAN (1989). An efficient and fast approach to floorplanning. *Proceedings of the International Conference on Computer-Aided Design & Computer Graphics*, IEEE, New York, 546–549.

S. SAHNI, A. BHATT (1980). The complexity of design automation problems. *17th Design Automation Conference: Proceedings*, IEEE, New York, 402–411.

S. SAHNI, T. GONZALEZ (1976). P-complete approximation problems. *Journal of the Association for Computing Machinery* **23**, 555–565.

A. SAID, R. PALAZZO, JR. (1993). Using combinatorial optimization to design good unit-memory convolutional codes. *IEEE Transactions on Information Theory* **39**, 1100–1108.

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

S.L. SAVAGE (1976). Some theoretical implications of local optimality. *Mathematical Programming* **10**, 354–366.

M.W.P. SAVELSBERGH (1986). Local search for routing problems with time windows. C.L. MONMA (ED.). *Algorithms and Software for Optimization, Part I*, Annals of Operations Research 4, Baltzer, Basel, 285–305.

M.W.P. SAVELSBERGH (1990). An efficient implementation of local search algorithms for constrained routing problems. *European Journal of Operational Research* **47**, 75–85.

M.W.P. SAVELSBERGH (1992). The vehicle routing problem with time windows: minimizing route duration. *ORSA Journal on Computing* **4**, 146–154.

M.W.P. SAVELSBERGH, M. SOL (1995). The general pickup and delivery problem. *Transportation Science* **29**, 17–29.

A.A. SCHÄFFER, M. YANNAKAKIS (1991). Simple local search problems that are hard to solve. *SIAM Journal on Computing* **20**, 56–87.

J.D. SCHAFFER (ED.) (1989). *Proceedings of the Third International Conference on Genetic Algorithms*, Morgan Kaufmann, San Mateo, CA.

J.D. SCHAFFER, L.J. ESHELMAN (1991). On crossover as an evolutionarily viable strategy. R.K. BELEW, L.B. BOOKER (EDS.). *Proceedings of the Fourth International Conference on Genetic Algorithms*, Morgan Kaufmann, San Mateo, CA, 61–68.

M. SCHALG, Y.Z. LIAO, C.K. WONG (1983). An algorithm for optimal two-dimensional compaction of VLSI layouts. *Integration, The VLSI Journal* **1**, 179–209.

D. SCHLIERKAMP-VOOSEN, H. MÜHLENBEIN (1994). Strategy adaptation by competing subpopulations. Y. DAVIDOR, H.-P. SCHWEFEL, R. MÄNNER (EDS.). *Parallel Problem Solving from Nature III*, Lecture Notes in Computer Science 866, Springer, Berlin, 199–208.

B. SCHNETZLER (1992). Des opérateurs d'échange et une méthode de relaxation pour le problème du voyageur de commerce. *Recherche Opérationnelle/Operations Research* **26**, 57–81 (in French).

L. SCHRAGE, K.R. BAKER (1978). Dynamic programming solution of sequencing problems with precedence constraints. *Operations Research* **26**, 444–449.

A. SCHRIJVER (1986). *Theory of Linear and Integer Programming*, Wiley, Chichester.

P.C. SCHUUR (1997). Classification of acceptance criteria for the simulated annealing algorithm. *Mathematics of Operations Research* **22**, 266–275.

H.-P. SCHWEFEL (1981). *Numerical Optimization of Computer Models*, Wiley, Chichester.

C. SECHEN (1988a). Chip-planning, placement, and global routing of macro/custom cell integrated circuits using simulated annealing. *25th ACM/IEEE Design Automation Conference: Proceedings 1988*, IEEE Computer Society Press, Los Alamitos, CA, 73–80.

C. SECHEN (1988b). *VLSI Placement and Routing Using Simulated Annealing*, Kluwer, Boston, MA.

C. SECHEN, K.W. LEE (1987). An improved simulated annealing algorithm for row-based placement. *IEEE International Conference on Computer-Aided Design: ICCAD-87*, IEEE Computer Society Press, Washington, DC, 478–481.

C. SECHEN, A.L. SANGIOVANNI-VINCENTELLI (1985). The TimberWolf placement and routing package. *IEEE Journal on Solid State Circuits* **30**, 510–522.

C. SECHEN, A.L. SANGIOVANNI-VINCENTELLI (1986). TimberWolf 3.2: a new standard cell placement and global routing package. *23rd ACM/IEEE Design Automation Conference: Proceedings 1986*, IEEE Computer Society Press, Los Alamitos, CA, 432–439.

F. SEMET, E. TAILLARD (1993). Solving real-life vehicle routing problems efficiently using tabu search. F. GLOVER, E. TAILLARD, M. LAGUNA, D. DE WERRA (EDS.). *Tabu Search*, Annals of Operations Resesearch 41, Baltzer, Basel, 469–488.

E. SENETA (1981). *Non-Negative Matrices and Markov Chains*, 2nd ed., Springer, New York.

K. SHAHOOKAR, P. MAZUMDER (1990). A genetic approach to the standard cell placement using meta-genetic parameter optimization. *IEEE Transactions on Computer-Aided Design* **9**, 500–511.

K. SHAHOOKAR, P. MAZUMDER (1991). VLSI cell placement techniques. *Computing Surveys* **23**, 143–220.

D.B. SHMOYS, D.P. WILLIAMSON (1990). Analyzing the Held–Karp TSP bound: a monotonicity property with applications. *Information Processing Letters* **35**, 281–285.

P. SIARRY, L. BERGONZI, G. DREYFUS (1987). Thermodynamic optimization of block placement. *IEEE Transactions on Computer-Aided Design* **6**, 211–221.

W. SIEDLECKI, J. SKLANSKY (1989). Constrained genetic optimization via dynamic reward–penalty balancing and its use in pattern recognition. J.D. SCHAFFER (ED.). *Proceedings of the Third International Conference on Genetic Algorithms*, Morgan Kaufmann, San Mateo, CA, 141–150.

M.W. SIMMEN (1991). Parameter sensitivity on the elastic net approach to the traveling salesman problem. *Neural Computation* **3**, 363–374.

D.D. SLEATOR, R.E. TARJAN (1985). Self-adjusting binary search trees. *Journal of the Association for Computing Machinery* **32**, 652–686.

S.P. SMITH (1992). An experiment on using genetic algorithms to learn scheduling heuristics. *Proceedings of the SPIE-Conference on Applications of Artificial Intelligence*

X: Knowledge-Based Systems, SPIE–The International Society for Optical Engineering, Bellingham, WA, Volume 1707, 378–386.

W.E. SMITH (1956). Various optimizers for single-stage production. *Naval Research Logistics Quarterly* **3**, 59–66.

M.M. SOLOMON (1987). Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations Research* **35**, 254–265.

M.M. SOLOMON, E.K. BAKER, J.R. SCHAFFER (1988). Vehicle routing and scheduling problems with time window constraints: efficient implementation of solution improvement procedures. B.L. GOLDEN, A.A. ASSAD (EDS.). *Vehicle Routing: Methods and Studies*, North-Holland, Amsterdam, 85–105.

M.M. SOLOMON, J. DESROSIERS (1988). Time window constrained routing and scheduling problems. *Transportation Science* **22**, 1–13.

L. SONG, A. VANNELLI (1992). A VLSI placement method using tabu search. *Microelectronics Journal* **23**, 167–172.

G.B. SORKIN (1991). Efficient simulated annealing on fractal energy landscapes. *Algorithmica* **6**, 367–418.

J. SOUKUP (1981). Circuit layout. *Proceedings of the IEEE* **69**, 1281–1304.

J. STANDER, B.W. SILVERMAN (1994). Temperature schedules for simulated annealing. *Statistics and Computing* **4**, 21–32.

R.G. STANTON (1969). Covering theorems in groups (or:how to win at football pools). W.T. TUTTE (ED.). *Recent Progress in Combinatorics*, Academic Press, New York, 21–36.

T. STARKWEATHER, S. MCDANIEL, K. MATHIAS, D. WHITLEY, C. WHITLEY (1991). A comparison of genetic sequencing operators. R.K. BELEW, L.B. BOOKER (EDS.). *Proceedings of the Fourth International Conference on Genetic Algorithms*, Morgan Kaufmann, San Mateo, CA, 69–76.

J.M. STEELE (1981). Complete convergence of short paths and Karp's algorithm for the TSP. *Mathematics of Operations Research* **6**, 374–378.

K. STEIGLITZ, P. WEINER (1968). Some improved algorithms for computer solution of the traveling salesman problem. *Proceedings of the 6th Annual Allerton Conference on Communication, Control, and Computing*, Department of Electrical Engineering and the Coordinated Science Laboratory, University of Illinois, Urbana, IL, 814–821.

D. STEIN (1977). *Scheduling dial-a-ride transportation systems: an asymptotic approach*, PhD thesis, Harvard University, Cambridge, MA.

G. STIMPFL-ABELE, L. GARRIDO (1991). Fast track finding with neural nets. *Computer Physics Communications* **64**, 46–56.

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

P.N. STRENSKI, S. KIRKPATRICK (1991). Analysis of finite length annealing schedules. *Algorithmica* **6**, 346–366.

Y. SUGAI, H. HIRATA (1991). Hierarchical algorithm for a partition problem using simulated annealing: application to placement in VLSI layout. *International Journal of Systems Science* **22**, 2471–2487.

J.Y. SUH, D. VAN GUCHT (1987a). *Distributed genetic algorithms*, Report 225, Computer Science Department, Indiana University, Bloomington, IN.

J.Y. SUH, D. VAN GUCHT (1987b). Incorporating heuristic information into genetic search. J.J. GREFENSTETTE (ED.). *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, Lawrence Erlbaum, Hillsdale, NJ, 100–107.

W.-J. SUN, C. SECHEN (1993). Efficient and effective placement for very large circuits. *1993 IEEE/ACM International Conference on Computer-Aided Design*, IEEE Computer Society Press, Los Alamitos, CA, 170–177.

G. SYSWERDA (1989). Uniform crossover in genetic algorithms. J.D. SCHAFFER (ED.). *Proceedings of the Third International Conference on Genetic Algorithms*, Morgan Kaufmann, San Mateo, CA, 2–9.

G. SYSWERDA (1993). Simulated crossover in genetic algorithms. D. WHITLEY (ED.). *Foundations of Genetic Algorithms 2*, Morgan Kaufmann, San Mateo, CA, 239–255.

H. SZU, R. HARTLEY (1987). Fast simulated annealing. *Physics Letters A* **122**, 157–162.

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

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

E. TAILLARD (1993a). Parallel iterative search methods for vehicle routing problems. *Networks* **23**, 661–673.

E. TAILLARD (1993b). Benchmarks for basic scheduling problems. *European Journal of Operational Research* **64**, 278–285.

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

R. TAMMASSIA, I.G. TOLLIS (1988). On improving channel routability by lateral shifting of the shores. *SIGDA Newsletter* **1**, 18–30.

L. TAO, Y.C. ZHAO, K. THULASIRAMAN, M.N.S. SWAMY (1992). Simulated annealing and tabu search algorithms for multiway graph partition. *Journal of Circuits, Systems, and Computers* **2**, 159–185.

O. TAUSSKY, J. TODD (1948). Covering theorems for groups. *Annales de la Société Polonaise de Mathématique* **21**, 303–305.

S.R. THANGIAH (1993). *Vehicle routing with time windows using genetic algorithms*, Technical report SRU-SpSc-TR-93-23, Slippery Rock University, Slippery Rock, PA.

S.R. THANGIAH, A.V. GUBBI (1993). Effect of genetic sectoring on vehicle routing problems with time windows. *Proceedings IEEE International Conference on Developing and Managing Intelligent System Projects*, IEEE, New York, 146–153.

S.R. THANGIAH, K.E. NYGARD, P.L. JUELL (1991). GIDEON: a genetic algorithm system for vehicle routing with time windows. *Seventh IEEE Conference on Artificial Intelligence Applications*, IEEE Computer Society Press, Los Alamitos, CA, 322–328.

P.M. THOMPSON, J.B. ORLIN (1989). *Theory of cyclic transfers*, MIT Operations Research Center Report OR 200–89, MIT, Cambridge, MA.

P.M. THOMPSON, H.N. PSARAFTIS (1993). Cyclic transfer algorithms for multivehicle routing and scheduling problems. *Operations Research* **41**, 935–946.

M.J. TODD (1980). The monotonic bounded Hirsch conjecture is false for dimension at least 4. *Mathematics of Operations Research* **5**, 599–601.

C.A. TOVEY (1983). On the number of iterations of local improvement algorithms. *Operations Research Letters* **2**, 231–238.

C.A. TOVEY (1984). A simplified NP-complete satisfiability problem. *Discrete Applied Mathematics* **8**, 85–90.

C.A. TOVEY (1985). Hill climbing with multiple local optima. *SIAM Journal on Algebraic and Discrete Methods* **6**, 384–393.

C.A. TOVEY (1986). Low order polynomial bounds on the expected performance of local improvement algorithms. *Mathematical Programming* **35**, 193–224.

M. TROYON (1988). *Quelques heuristiques et résultats asymptotiques pour trois problèmes d'optimisation combinatoire*, PhD thesis 754, Ecole Polytechnique Fédérale de Lausanne, Lausanne (in French).

N.L.J. ULDER, E.H.L. AARTS, H.-J. BANDELT, P.J.M. VAN LAARHOVEN, E. PESCH (1991). Genetic local search algorithms for the traveling salesman problem. H.-P. SCHWEFEL, R. MÄNNER (EDS.). *Parallel Problem Solving from Nature*, Lecture Notes in Computer Science 496, Springer, Berlin, 109–116.

R.J.M. VAESSENS, E.H.L. AARTS, J.K. LENSTRA (1992). A local search template. R. MÄNNER, B. MANDERICK (EDS.). *Parallel Problem Solving from Nature 2*, North-Holland, Amsterdam, 65–74.

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

R.J.M. VAESSENS, E.H.L. AARTS, J.H. VAN LINT (1993). Genetic algorithms in coding theory: a table for $A_3(n, d)$. *Discrete Applied Mathematics* **45**, 71–87.

J. VAISEY, A. GERSHO (1988). Simulated annealing and codebook design. *ICASSP 88: 1988 International Conference on Acoustics, Speech, and Signal Processing*, IEEE, New York, 1176–1179.

A.J. VAKHARIA, Y.-L. CHANG (1990). A simulated annealing approach to scheduling a manufacturing cell. *Naval Research Logistics* **37**, 559–577.

A.I. VAKHUTINSKY, B.L. GOLDEN (1996). A hierarchical strategy for solving traveling salesman problems using elastic nets. *Journal of Heuristics* **1**, 67–76.

C.L. VALENZUELA, A.J. JONES (1994). Evolutionary divide and conquer (I): a novel genetic approach to the TSP. *Evolutionary Computation* **1**, 313–333.

M.P. VECCHI, S. KIRKPATRICK (1983). Global wiring by simulated annealing. *IEEE Transactions on Computer-Aided Design* **2**, 215–222.

S.L. VAN DE VELDE (1993). Duality-based algorithms for scheduling unrelated parallel machines. *ORSA Journal on Computing* **5**, 192–205.

M.G.A. VERHOEVEN, E.H.L. AARTS (1996). Parallel local search techniques. *Journal of Heuristics* **1**, 43–65.

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

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

M.G.A. VERHOEVEN, P.C.J. SWINKELS, E.H.L. AARTS (1995). *Parallel local search and the traveling salesman problem*, Working paper, Philips Research Laboratories, Eindhoven.

R.V.V. VIDAL (ED.) (1993). *Applied Simulated Annealing*, Lecture Notes in Economics and Mathematical Systems 396, Springer, Berlin.

V.G. VIZING (1978). Complexity of the traveling salesman problem in the class of monotonic improvement algorithms. *Engineering Cybernetics* **4**, 623–626.

H.-M. VOIGT, H. MÜHLENBEIN, D. CVETKOVIC (1995). Fuzzy recombination for the breeder genetic algorithm. L.J. ESHELMAN (ED.). *Proceedings of the Sixth International Conference on Genetic Algorithms*, Morgan Kaufmann, San Fransisco, CA, 104–111.

A. VOLGENANT, R. JONKER (1983). The symmetric traveling salesman problem and edge exchanges in minimal 1-trees. *European Journal of Operational Research* **12**, 394–403.

Q. WANG (1987). Optimization by simulating molecular evolution. *Biological Cybernetics* **57**, 95–101.

E.W. WEBER (1983). On the football pool problem for 6 matches: a new upper bound. *Journal of Combinatorial Theory, Series A* **35**, 106–108.

A. WEIGEND, B.A. HUBERMAN, D. RUMELHART (1990). Predicting the future: a connectionist approach. *International Journal of Neural Systems* **3**, 193–210.

P. WEINER, S.L. SAVAGE, A. BAGCHI (1976). Neighborhood search algorithms for guaranteeing optimal traveling salesman tours must be inefficient. *Journal of Computer and System Sciences* **12**, 25–35.

F. WERNER (1993). On the heuristic solution of the permutation flow shop problem by path algorithms. *Computers & Operations Research* **20**, 707–722.

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

D. WHITLEY (1989). The genitor algorithm and selection pressure: why rank-based allocation of reproductive trials is best. J.D. SCHAFFER (ED.). *Proceedings of the Third*

International Conference on Genetic Algorithms, Morgan Kaufmann, San Mateo, CA, 116–121.

D. WHITLEY, T. STARKWEATHER, D. FUQUAY (1989). Scheduling problems and traveling salesmen: the genetic edge recombination operator. J.D. SCHAFFER (ED.). *Proceedings of the Third International Conference on Genetic Algorithms*, Morgan Kaufmann, San Mateo, CA, 133–140.

D. WHITLEY, T. STARKWEATHER, D. SHANER (1990). Using simulations with genetic algorithms for optimizing schedules. *Proceedings of the 1990 Summer Computer Simulation Conference*, North-Holland, Amsterdam, 288–293.

M. WIDMER (1991). Job shop scheduling with tooling constraints: a tabu search approach. *Journal of the Operational Research Society* **42**, 75–82.

M. WIDMER, A. HERTZ (1989). A new heuristic method for the flow shop sequencing problem. *European Journal of Operational Research* **41**, 186–193.

L.J. WILKERSON, J.D. IRWIN (1971). An improved method for scheduling independent tasks. *AIIE Transactions* **3**, 239–245.

J.A.G. WILLARD (1989). *Vehicle routing using r-optimal tabu search*, MSc thesis, Management School, Imperial College, London.

L.T. WILLE (1987). The football pool problem for 6 matches: a new upper bound obtained by simulated annealing. *Journal of Combinatorial Theory, Series A* **45**, 171–177.

L.T. WILLE (1990). Improved binary code coverings by simulated annealing. *Congressus Numerantium* **73**, 53–58.

L.T. WILLE (1996). New binary covering codes obtained by simulated annealing. *IEEE Transactions on Information Theory* **42**, 300–302.

K. WILLIAMSON HOKE (1988). Completely unimodal numberings of a simple polytope. *Discrete Applied Mathematics* **20**, 69–81.

R.J. WILLIS, B.J. TERRILL (1994). Scheduling the Australian state cricket season using simulated annealing. *Journal of the Operational Research Society* **45**, 276–280.

G.V. WILSON, G.S. PAWLEY (1988). On the stability of the traveling salesman problem algorithm of Hopfield and Tank. *Biological Cybernetics* **58**, 63–70.

L.A. WOLSEY (1980). Heuristic analysis, linear programming, and branch and bound. *Mathematical Programming Studies* **13**, 121–134.

C.-P. WONG, R.-D. FIEBRICH (1987). Simulated annealing-based circuit placement algorithm on the Connection Machine system. *Proceedings 1987 IEEE International Conference on Computer Design: VLSI in Computers & Processors; ICCD'87*, IEEE Computer Society Press. Los Alamitos, 78–82.

D.F. WONG, H.W. LEONG, C.L. LIU (1986). Multiple PLA folding by the method of simulated annealing. *Proceedings of the IEEE 1986 Custom Integrated Circuits Conference*, IEEE, New York, 351–355.

D.F. WONG, H.W. LEONG, C.L. LIU (1988). *Simulated Annealing for VLSI Design*, Kluwer, Boston, MA.

D.F. WONG, C.L. LIU (1986). A new algorithm for floorplan design. *23rd ACM/IEEE Design Automation Conference: Proceedings 1986*, IEEE Computer Society Press, Los Alamitos, CA, 101–107.

D.F. WONG, C.L. LIU (1987). Floorplan design for rectangular and L-shaped modules. *IEEE International Conference on Computer-Aided Design: ICCAD-87*, IEEE Computer Society Press, Washington, DC, 520–523.

D.L. WOODRUFF, M.L. SPEARMAN (1992). Sequencing and batching for two classes of jobs with deadlines and setup times. *Production and Operations Management* **1**, 87–102.

F.Y. WU (1983). The Potts model. *Review of Modern Physics* **54**, 235–268.

X. XU, W.T. TSAI (1991). Effective neural algorithms for the traveling salesman problem. *Neural Networks* **4**, 193–205.

T. YAMADA, R. NAKANO (1992). A genetic algorithm applicable to large-scale job-shop problems. R. MÄNNER, B. MANDERICK (EDS.). *Parallel Problem Solving from Nature 2*, North-Holland, Amsterdam, 281–290.

T. YAMADA, B.E. ROSEN, R. NAKANO (1994). A simulated annealing approach to job shop scheduling using critical block transition operators. *The 1994 IEEE International Conference on Neural Networks*, IEEE, New York, 4687–4692.

M. YANNAKAKIS (1990). The analysis of local search problems and their heuristics. C. CHOFFRUT, T. LENGAUER (EDS.). *STACS 90: 7th Annual Symposium on Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science 415, Springer, Berlin, 298–311.

M. YANNAKAKIS (1994). On the approximation of maximum satisfiability. *Journal of Algorithms* **17**, 475–502.

C.A. YANO, Y.-D. KIM (1991). Algorithms for a class of single-machine weighted tardiness and earliness problems. *European Journal of Operational Research* **52**, 167–178.

T. YOSHIMURA, E.S. KUH (1982). Efficient algorithms for channel routing. *IEEE Transactions on Computer-Aided Design* **1**, 25–35.

A.L. YUILLE (1990) Generalized deformable models, statistical physics, and matching problems. *Neural Computation* **2**, 1–24.

A.L. YUILLE, K. HONDA, C. PETERSON (1991). Particle tracking by deformable templates. *IJCNN-91-Seattle: International Joint Conference on Neural Networks*, IEEE, New York, I 7–12.

M. ZACHARIASEN, M. DAM (1996). Tabu search on the geometric traveling salesman problem. I.H. OSMAN, J.P. KELLY (EDS.). *Meta-heuristics: Theory and Applications*, Kluwer, Boston, 571–587.

N. ZADEH (1980). *What is the worst case behavior of the simplex algorithm?* Technical report OR80-27, Department of Operations Research, Stanford University, Stanford, CA.

M.R. ZARGHAM (1992). A simulated annealing multi-layer router. *Integration, The VLSI Journal* **13**, 179–193.

R.J.A. ZEESTRATEN (1985). *Two-dimensional compaction*, MSc thesis, Philips Research Laboratories, Eindhoven.

K. ZEGER, A. GERSHO (1989). Stochastic relaxation algorithm for improved vector quantiser design. *Electronics Letters* **25**, 896–898.

K. ZEGER, J. VAISEY, A. GERSHO (1992). Globally optimal vector quantizer design by stochastic relaxation. *IEEE Transactions on Signal Processing* **40**, 310–322.

Z.Z. ZHANG (1992). The connections between neural networks and minimum weighted distance decoding. *Proceedings of the International Joint Conference on Neural Networks, Volume 1*, Publishing House of Electronics Industry, Beijing, 282–286.

D.N. ZHOU, V. CHERKASSKY, T.R. BALDWIN, D.E. OLSON (1991). A neural network approach to job-shop scheduling. *IEEE Transactions on Neural Networks* **2**, 175–179.