M. Jünger G. Reinelt G. Rinaldi

THE TRAVELING SALESMAN PROBLEM

R. 375 Gennaio 1994

Michael Jünger - Institut für Informatik der Universität zu Köln, Pohligstraße 1, D-50696 Köln, Germany.

Gerhard Reinelt - Institut für Angewandte Mathematik, Universität Heidelberg, Im Neuenheimer Feld 294, D-69120 Heidelberg, Germany.

Giovanni Rinaldi - Istituto di Analisi dei Sistemi ed Informatica del CNR, Viale Manzoni 30, 00185 Roma, Italy.

This paper presents a self-contained introduction into algorithmic and computational aspects of the traveling salesman problem and of related problems, along with their theoretical prerequisites as seen from the point of view of an operations researcher who wants to solve practical problem instances.

Extensive computational results are reported on most of the algorithms described. Optimal solutions are reported for instances with sizes up to several thousand nodes as well as heuristic solutions with provably very high quality for larger instances.

This is a preliminary version of one of the chapters of the volume "Networks" edited by M.O. Ball, T.L. Magnanti, C.L. Monma, and G.L. Nemhauser, of the series *Handbooks in Operations Research and Management Science*, to be published by North-Holland, Amsterdam.

CONTENTS

The Traveling Salesman Problem

Michael Jünger

Institut für Informatik der Universität zu Köln
Pohligstraße 1, D-50696 Köln, Germany

Gerhard Reinelt

Institut für Angewandte Mathematik, Universität Heidelberg
Im Neuenheimer Feld 294, D-69120 Heidelberg, Germany

Giovanni Rinaldi

Istituto di Analisi dei Sistemi ed Informatica
Viale Manzoni 30, I-00185 Roma, Italy

1. Introduction

A traveling salesman wants to visit each of a set of towns exactly once, starting from and returning to his home town. One of his problems is to find the shortest such trip.

The traveling salesman problem, TSP for short, has model character in many branches of Mathematics, Computer Science, and Operations Research. Heuristics, linear programming, and branch and bound, which are still the main components of todays most successful approaches to hard combinatorial optimization problems, were first formulated for the TSP and used to solve practical problem instances in 1954 by Dantzig, Fulkerson and Johnson.

When the theory of $\mathcal{NP}$-completeness developed, the TSP was one of the first problems to be proven $\mathcal{NP}$-hard by Karp in 1972. New algorithmic techniques have first been developed for or at least have been applied to the TSP to show their effectiveness. Examples are branch and bound, Lagrangean relaxation, Lin-Kernighan type methods, simulated annealing, and the field of polyhedral combinatorics for hard combinatorial optimization problems (polyhedral cutting plane methods and branch and cut).

This work was partially supported by EEC Contract SC1-CT91-0620

This chapter presents a self-contained introduction into algorithmic and computational aspects of the traveling salesman problem along with their theoretical prerequisites as seen from the point of view of an operations researcher who wants to solve practical instances. Lawler et al. (1985) motivated considerable research in this area, most of which became apparent at the specialized conference on the TSP which took place at Rice University in 1990. This chapter is intended to be a guideline for the reader confronted with the question of how to attack a TSP instance depending on its size, its structural properties (e.g., metric), the available computation time, and the desired quality of the solution (which may range from, say, a 50% guarantee to optimality). In contrast to previous surveys, here we are concerned with practical problem solving, i.e., theoretical results are presented in a form which make clear their importance in the design of algorithms for approximate but provably good, and optimal solutions of the TSP. For space reasons, we concentrate on the symmetric TSP and discuss related problems only in terms of their practical importance and the structural and algorithmic insights they provide for the symmetric TSP.

For the long history of the TSP we refer to Hoffman and Wolfe (1985). The relevant algorithmic approaches, however, have all taken place in the last 40 years. The developments until 1985 are contained in Lawler et al. (1985). This chapter gives the most recent significant developments. Historical remarks are confined to achievements which appear relevant from our point of view.

Let $K_n = (V_n, E_n)$ be the complete undirected graph with $n = |V_n|$ nodes and $m = |E_n| = \binom{n}{2}$ edges. An edge e with endpoints i and j is also denoted by ij, or by (i, j). We denote by $\mathbb{R}^{E_n}$ the space of real vectors whose components are indexed by the elements of E_n. The component of any vector $z \in \mathbb{R}^{E_n}$ indexed by the edge $e = ij$ is denoted by z_e, z_{ij}, or $z(i, j)$.

Given an objective function $c \in \mathbb{R}^{E_n}$, that associates a "length" c_e with every edge e of K_n, the *symmetric traveling salesman problem* consists of finding a *Hamiltonian cycle* (a cycle visiting every node exactly once) such that its c-length (the sum of the lengths of its edges) is as small (large) as possible. Without loss of generality, we only consider the minimization version of the problem. From now on we use the abbreviation *TSP* only for the symmetric traveling salesman problem.

Of special interest are the Euclidean instances of the traveling salesman problem. In these instances the nodes defining the problem correspond to points in the 2-dimensional plane and the distance between two nodes is the Euclidean distance between their corresponding points. More generally, instances that satisfy the triangle inequality, i.e., $c_{ij} + c_{jk} \geqslant c_{ik}$ for all three distinct i, j, and k, are of particular interest.

The reason for using a complete graph in the definition of the TSP is that for such a graph the existence of a feasible solution is always guaranteed, while for general graphs deciding the existence of a Hamiltonian cycle is an $\mathcal{NP}$-complete problem. Actually, the number of Hamiltonian cycles in K_n, i.e., the size of the set of feasible solutions of the TSP, is $(n-1)!/2$. The TSP defined on general graphs is shortly described in Section 2 along with other combinatorial

optimization problems whose relation to the TSP is close enough to make the algorithmic techniques covered in this chapter promising for the solution with various degrees of suitability. In Section 3 we discuss a selection of practical applications of the TSP or one of its close relatives. The algorithmic treatment of the TSP starts in Section 4 in which we cover approximation algorithms that cannot guarantee to find the optimum, but which are the only available techniques for finding good solutions to large problem instances. To assess the quality of a solution, one has to be able to compute a lower bound on the value of the shortest Hamiltonian cycle. Section 5 presents several relaxations on which lower bound computations can be based. Special emphasis is given to linear programming relaxations, which serve as a basis for finding optimal and provably good solutions within an enumerative environment to be discussed in Section 6. We do not address the algorithmic treatment of special cases of the TSP, where the special structure of the objective function can be exploited to find the optimal solution in polynomial time. Surveys on this subject are, e.g., Burkard (1989), Gilmore, Lawler and Shmoys (1985), van Dal (1992), van der Veen (1992), and Warren (1993). Finally, in Section 7 we report on computational experiments for several TSP instances.

2. Related problems

We begin with some transformations showing that the TSP can be applied in a more general way than suggested by its definition (for some further examples see, e.g., Garfinkel (1985)).

We give transformations to some related problems or variants of the TSP. It is often convenient to assume that all edge lengths are positive. By adding a suitable constant to all edge lengths we can bring any TSP instance into this form. However we do have to keep in mind that there are algorithms whose performance may be sensitive to such a transformation.

Since we are concerned with practical computation, we can assume rational, and thus, integer data.

Traveling salesman problems in general graphs

There may be situations where we want to find shortest Hamiltonian cycles in arbitrary graphs $G = (V, E)$, in particular in graphs which are not complete. Depending on the requirements we can treat such cases in two ways. We discuss the first possibility here, the second one is given below in the discussion of the graphical TSP.

If it is required that each node is visited exactly once and that only edges of the given graph must be used then we do the following. Add all missing edges giving them a sufficiently large weight M (e.g., $M > \sum_{e \in E} c_e$) and apply an algorithm for the TSP in complete graphs. If this algorithm terminates with an optimal solution containing none of the edges with weight M then this solution is also optimal for the original problem. If an edge with weight M is contained in the optimal solution then the original graph does not contain a Hamiltonian

cycle. Heuristics cannot guarantee to find a Hamiltonian cycle in G even if one exists, such a guarantee can only be provided by exact algorithms.

The second way to treat such problems is to allow that nodes may be visited more than once and edges be traversed more than once. If the given graph is connected we can always find a feasible round trip under this relaxation. This leads us to the so-called graphical traveling salesman problem.

The graphical traveling salesman problem

As in the case of the TSP we are given n cities, a set of connections between the cities represented in a graph $G = (V, E)$, and a "length" c_e for each connection $e \in E$. We assume that G is connected, otherwise no feasible solution exists. The *graphical traveling salesman problem* consists of finding a trip for the salesman to visit every city requiring the least possible total distance.

To define a feasible trip the salesman has to leave the home town (any node in the graph), visit any other town at least once, and go back to the home town. It is possible that a town is actually visited more than once and that an edge of G is "traveled" more than once. Such a feasible trip is called a *tour*. To avoid unbounded situations every edge has nonnegative weight. Otherwise we could use an edge as often as we like in both directions to achieve an arbitrarily negative length of the solution.

This is sometimes a more practical definition of the TSP because we may have cases where the underlying graph of connections is not Hamiltonian.

We transform a graphical TSP to a TSP as follows. Consider the TSP on the complete graph $K_n = (V_n, E_n)$, where for each edge $ij \in E_n$ the objective function coefficient d_{ij} is given by the c-length of a shortest path from i to j in the graph G. Solving the TSP in K_n gives a Hamiltonian cycle $H \subseteq E_n$. The solution of the graphical TSP can be obtained by replacing each edge in H that is not in G with the edges of a shortest path that connects its endpoints in G.

Hamiltonian and semi-Hamiltonian graphs

A graph is called *Hamiltonian* if it contains a Hamiltonian cycle and it is called *semi-Hamiltonian* if it contains a Hamiltonian path, i.e., a path joining two nodes of the graph and visiting every node exactly once. Checking if a graph $G = (V, E)$ is Hamiltonian or semi-Hamiltonian can be done by solving a TSP in a complete graph where all edges of the original graph obtain weight 1 and all other edges obtain weight 2. If the length of an optimal Hamiltonian cycle in the complete graph is n, then G is Hamiltonian and therefore semi-Hamiltonian. If the length is $n + 1$, then G is semi-Hamiltonian, but not Hamiltonian. And, finally, if the length is $n + 2$ or more, G is not semi-Hamiltonian.

The asymmetric traveling salesman problem

In this case the cost of traveling from city i to city j is not necessarily the same as for traveling from city j to city i. This is reflected by formulating the *asymmetric traveling salesman problem* (*ATSP*) as finding a shortest directed Hamiltonian cycle in a weighted digraph. Let $D = (W, A)$, $W = \{1, 2, \ldots, n\}$, $A \subseteq W \times W$, be the digraph for which the ATSP has to be solved. Let d_{ij} be

the distance from node i to node j, if there is an arc in A with tail i and head j. We define an undirected graph $G = (V, E)$ by

$$V = W \cup \{n+1, n+2, \ldots, 2n\},$$
$$E = \{(i, n+i) \mid i = 1, 2, \ldots, n\} \cup \{(n+i, j) \mid (i, j) \in A\}.$$

Edge weights are computed as follows

$$c_{i,n+i} = -M \qquad \text{for } i = 1, 2, \ldots, n,$$
$$c_{n+i,j} = d_{ij} \qquad \text{for } (i, j) \in A,$$

where M is a sufficiently large number, e.g., $M = \sum_{(i,j)\in A} d_{ij}$. It is easy to see that for each directed Hamiltonian cycle in D with length d_D there is a Hamiltonian cycle in G with length $c_G = d_D - nM$. In addition, all edges with weight $-M$ are contained in an optimal Hamiltonian cycle in G. Therefore, this cycle induces a directed Hamiltonian cycle in D.

In our discussion on computational results in Section 7 we report on the solution of a hard asymmetric TSP instance that we attacked with symmetric TSP methods.

The multisalesmen problem

Instead of just one salesman we have m salesmen available who are all located in city $n+1$ and have to visit cities $1, 2, \ldots, n$. The cost of the solution is the total distance traveled by all salesmen together (all of them must travel). This is the basic situation when in vehicle routing m vehicles, located at a common depot, have to serve n customers.

We can transform this problem to the TSP by splitting city $n+1$ into m cities $n+1, n+2, \ldots, n+m$. The edges $(i, n+k)$, with $1 \leqslant i \leqslant n$ and $2 \leqslant k \leqslant m$, receive the weight $c(i, n+k) = c(i, n+1)$, and all edges connecting the nodes $n+1, n+2, \ldots, n+m$ receive a large weight M.

The rural postman problem

We are given a graph $G = (V, E)$ with edge weights $c(i, j)$ and a subset $F \subseteq E$. The *rural postman problem* consists of finding a shortest tour, containing all edges in F, in the subgraph of G induced by some subset of V. We call such a tour a *rural postman tour* of G. As for the graphical TSP we have to assume nonnegative edge weights to avoid unbounded situations.

If F induces a connected subgraph of G, then we have the special case of a *Chinese postman problem* which can be solved in polynomial time using matching techniques (Edmonds and Johnson (1973)). In general the problem is $\mathcal{NP}$-hard, since the TSP can easily be transformed to it. First add a sufficiently large number to all edge weights to guarantee that triangle inequality holds. Then split each node i into two nodes i and i'. For any edge (i, j) generate edges (i', j) and (i, j') with weights $c(i', j) = c(i, j') = c(i, j)$, and the edges connecting i to i' and j to j' receive zero weights. F consists of all the edges (i, i').

Conversely, we can transform the rural postman problem to the TSP as follows. Let $G_F = (V_F, F)$ be the subgraph of G induced by F. With every node $i \in V_F$ we associate a set $S_i = \{s_i^j \mid j \in N(i)\}$ where $N(i)$ is the set of neighbors of node i in G_F. Construct the weighted complete graph $G' = (V', E', c')$ on the set $V' = \bigcup_{i \in V_F} S_i$. The edge weights c' are defined as follows.

$$c'(s_i^h, s_i^k) = 0 \quad \text{for } i \in V_F \text{ and } h, k \in N(i),\ h \neq k$$

$$c'(s_i^h, s_j^k) = \begin{cases} -M & \text{if } i = k \text{ and } j = h \\ d(i,j) & \text{otherwise} \end{cases} \qquad \begin{array}{l} \text{for all } i, j \in V_F, i \neq j, \\ \qquad h \in N(i), \\ \qquad k \in N(j), \end{array}$$

where we denote by $d(i,j)$ the c-length of a shortest path between i and j in G. It is trivial to transform an optimal Hamiltonian cycle in G' to an optimal rural postman tour in G. We can easily generalize this transformation for the case in which not only edges, but also some nodes are required to be in the tour. Such nodes are simply added to the resulting TSP instance, and all new edges receive as weights the corresponding shortest path lengths. In Section 7 we report on the solution of some instances that we obtained using this transformation.

The shortest Hamiltonian path problem

We are given a graph $G = (V, E)$ with edge weights c_{ij}. Two special nodes, say v_s and v_t, of V are also given. The task is to find a path from v_s to v_t visiting each node of V exactly once with minimum length, i.e., to find the shortest Hamiltonian path in G from v_s to v_t.

This problem can be solved as a standard TSP in two ways.

a) Choose M sufficiently large and assign weight $-M$ to the edge from v_s to v_t (which is created if it does not belong to E). Then compute the shortest Hamiltonian cycle in this graph. This cycle must contain edge $v_s v_t$ and thus solves the Hamiltonian path problem.

b) Add a new node 0 to V and edges from 0 to v_s and to v_t with weight 0. Each Hamiltonian cycle in this new graph corresponds to a Hamiltonian path from v_s to v_t in the original graph with the same length.

If only the starting point v_s of the Hamiltonian path is fixed we can solve the problem by introducing a new node 0 and adding edges from all nodes $v \in V \backslash \{v_s\}$ to 0 with zero length. Now we can solve the Hamiltonian path problem with starting point v_s and terminating point $v_t = 0$ which solves the original problem.

If also no starting point is specified, we just add node 0 and connect all other nodes to 0 with edges of length zero. In this new graph we solve the standard TSP.

The bottleneck traveling salesman problem

Instead of Hamiltonian cycles with minimum total length one searches in this problem for those whose longest edge is as short as possible. This *bottleneck traveling salesman problem* can be solved by a sequence of TSP instances. To see this, observe that the exact values of the distances are not of interest under

this objective function, only their relative order matters. Hence we may assume that we have at most $\frac{1}{2}n(n-1)$ different integral distances and that the largest of them is not greater than $\frac{1}{2}n(n-1)$. We now solve problems of the following kind for some parameter b:

> Is the graph consisting of all edges with weights at most b Hamiltonian?

This is exactly the problem discussed above. By performing a binary search on the parameter b (starting, e.g., with $b = \frac{1}{4}n(n-1)$) we can identify the smallest such b leading to a "yes" answer by solving at most $O(\log n)$ TSP instances.

We have seen that a variety of related problems can be transformed to the TSP. However, each such transformation has to be considered with some care, before actually trying to use it for practical problem solving. For example, the shortest path computations necessary to treat a graphical TSP as a TSP take time $O(n^3)$ which might not be acceptable in practice. Many transformations require the introduction of a large number M. This can lead to numerical problems or may even prevent the finding of feasible solutions at all using heuristics. In particular, for LP-based approaches, the usage of the "big M" cannot be recommended. Here it is preferable use "variable fixing techniques" (see Section 6) to force edges with cost $-M$ into the solution and prevent those with cost M in the solution. Moreover, in general, the transformations described above may produce TSP instances that are difficult to solve both for heuristic and exact algorithms.

3. Practical applications

Since we are aiming at the development of algorithms and heuristics for practical traveling salesman problem solving, we give a survey on some of the possible applications. The list is not complete but covers some important cases. We start with applications that can be modeled directly as one of the variants given in the previous section.

Drilling of printed circuit boards

A direct application of the TSP is the drilling problem whose solution plays an important rôle in economical manufacturing of printed circuit boards (PCBs). A computational study in an industry application of a large electronics company can be found in Grötschel, Jünger and Reinelt (1991).

To connect a conductor on one layer with a conductor on another layer, or to position (in a later stage of the PCB production) the pins of integrated circuits, holes have to be drilled through the board. The holes may be of different diameters. To drill two holes of different diameters consecutively, the head of the machine has to move to a tool box and change the drilling equipment. This is quite time consuming. Thus it is clear at the outset that one has to choose some diameter, drill all holes of the same diameter, change the drill, drill the holes of the next diameter, etc.

Thus, this drilling problem can be viewed as a sequence of TSP instances, one for each hole diameter, where the "cities" are the initial position and the set of all holes that can be drilled with one and the same drill. The "distance" between two cities is given by the time it takes to move the drilling head from one position to the other. The aim here is to minimize the travel time for the head of the machine.

X-Ray crystallography

An important application of the TSP occurs in the analysis of the structure of crystals (Bland and Shallcross (1987), Dreissig and Uebach (1990)). Here an X-ray diffractometer is used to obtain information about the structure of crystalline material. To this end a detector measures the intensity of X-ray reflections of the crystal from various positions. Whereas the measurement itself can be accomplished quite fast, there is a considerable overhead in positioning time since up to hundreds of thousands positions have to be realized for some experiments. In the two examples that we refer to, the positioning involves moving four motors. The time needed to move from one position to the other can be computed very accurately. The result of the experiment does not depend on the sequence in which the measurements at the various positions are taken. However, the total time needed for the experiment depends on the sequence. Therefore, the problem consists of finding a sequence that minimizes the total positioning time. This leads to a traveling salesman problem.

Overhauling gas turbine engines

This application was reported by Plante, Lowe and Chandrasekaran (1987) and occurs when gas turbine engines of aircraft have to be overhauled. To guarantee a uniform gas flow through the turbines there are so-called nozzle-guide vane assemblies located at each turbine stage. Such an assembly basically consists of a number of nozzle guide vanes affixed about its circumference. All these vanes have individual characteristics and the correct placement of the vanes can result in substantial benefits (reducing vibration, increasing uniformity of flow, reducing fuel consumption). The problem of placing the vanes in the best possible way can be modeled as a TSP with a special objective function.

The order-picking problem in warehouses

This problem is associated with material handling in a warehouse (Ratliff and Rosenthal (1981)). Assume that at a warehouse an order arrives for a certain subset of the items stored in the warehouse. Some vehicle has to collect all items of this order to ship them to the customer. The relation to the TSP is immediately seen. The storage locations of the items correspond to the nodes of the graph. The distance between two nodes is given by the time needed to move the vehicle from one location to the other. The problem of finding a shortest route for the vehicle with minimum pickup time can now be solved as a TSP. In special cases this problem can be solved easily, see van Dal (1992) for an extensive discussion and for references.

Computer wiring

A special case of connecting components on a computer board is reported in Lenstra and Rinnooy Kan (1974). Modules are located on a computer board and a given subset of pins has to be connected. In contrast to the usual case where a Steiner tree connection is desired, here the requirement is that no more than two wires are attached to each pin. Hence we have the problem of finding a shortest Hamiltonian path with unspecified starting and terminating points.

A similar situation occurs for the so-called testbus wiring. To test the manufactured board one has to realize a connection which enters the board at some specified point, runs through all the modules, and terminates at some specified point. For each module we also have a specified entering and leaving point for this test wiring. This problem also amounts to solving a Hamiltonian path problem with the difference that the distances are not symmetric and that starting and terminating point are specified.

Scheduling with sequence dependent process times

We are given n jobs that have to be performed on some machine. The time to process job j is t_{ij} if i is the job performed immediately before j (if j is the first job then its processing time is t_{0j}). The task is to find an execution sequence for the jobs such that the total processing time is as short as possible. Clearly, this problem can be modeled as a shortest (directed) Hamiltonian path problem.

Suppose the machine in question is an assembly line and that the jobs correspond to operations which have to be performed on some product at the workstations of the line. In such a case the primary interest would lie in balancing the line. Therefore, instead of the shortest possible time to perform all operations on a product, the longest individual processing time needed on a workstation is important. To model this requirement a bottleneck TSP is more appropriate.

Sometimes the TSP comes up as a subproblem in more complex combinatorial optimization processes that are devised to deal with production problems in industry. In such cases there is often no hope for algorithms with guaranteed performance, but hybrid approaches proved to be practical. We give three examples that cannot be transformed to the TSP, but share some characteristics of the TSP, or in which the TSP comes up as a subproblem.

Vehicle routing

Suppose that in a city n mail boxes have to be emptied every day within a certain period of time, say 1 hour. The problem is to find the minimum number of trucks to do this and the shortest time to do the collections using this number of trucks. As another example, suppose that n customers require certain amounts of some commodities and a supplier has to satisfy all demands with a fleet of trucks. The problem is to find an assignment of customers to the trucks and a delivery schedule for each truck so that the capacity of each truck is not exceeded and the total travel distance is minimized. Several variations of these

two problems, where time and capacity constraints are combined, are common in many real-world applications.

This problem is solvable as a TSP if there are no time and capacity constraints and if the number of trucks is fixed (say m). In this case we obtain an m-salesmen problem. Nevertheless, one may apply methods for the TSP to find good feasible solutions for this problem (see Lenstra and Rinnooy Kan (1974)).

Mask plotting in PCB production

For the production of each layer of a printed circuit board, as well as for layers of integrated semiconductor devices, a photographic mask has to be produced. In our case for printed circuit boards this is done by a mechanical plotting device. The plotter moves a lens over a photosensitive coated glass plate. The shutter may be opened or closed to expose specific parts of the plate. There are different apertures available to be able to generate different structures on the board.

Two types of structures have to be considered. A line is exposed on the plate by moving the closed shutter to one endpoint of the line, then opening the shutter and moving it to the other endpoint of the line. Then the shutter is closed. A point type structure is generated by moving (with the appropriate aperture) to the position of that point then opening the shutter just to make a short flash, and then closing it again. Exact modeling of the plotter control problem leads to a problem more complicated than the TSP and also more complicated than the rural postman problem. A real-world application in the actual production environment is reported in Grötschel, Jünger and Reinelt (1991).

Control of robot motions

In order to manufacture some workpiece a robot has to perform a sequence of operations on it (drilling of holes of different diameters, cutting of slots, planishing, etc.). The task is to determine a sequence of the necessary operations that leads to the shortest overall processing time. A difficulty in this application arises because there are precedence constraints that have to be observed. So here we have the problem of finding the shortest Hamiltonian path (where distances correspond to times needed for positioning and possible tool changes) that satisfies certain precedence relations between the operations.

4. Approximation algorithms for the TSP

When trying to solve practical TSP instances to optimality, one quickly encounters several difficulties. It may be possible that there is no algorithm at hand to solve an instance optimally and that time or knowledge do not permit the development and implementation of such an algorithm. The instances may be simply too large and therefore beyond the capabilities of even the best algorithms for attempting to find optimal solutions. On the other hand, it may also be possible that the time allowed for computation is not enough for an algorithm to reach the optimal solution. In all these cases there is a definite need for ap-

proximation algorithms (heuristics) which determine solutions of good quality and yield the best results achievable subject to the given side constraints.

It is the aim of this section to survey heuristics for the TSP and to give guidelines for their potential incorporation for the treatment of practical problems. We will first consider construction heuristics which build an initial Hamiltonian cycle. Procedures for improving a given cycle are discussed next. The third part is concerned with particular advantages one can exploit if the given problem instances are of geometric nature. A survey of other recently developed techniques concludes this section.

There is a huge number of papers dealing with finding near optimal solutions for the TSP. We therefore confine ourselves to the approaches that we think provide the most interesting ideas and that are important for attacking practical problems. This section is intended to give the practitioner enough detail to be able to design successful heuristics for large-scale TSP instances without studying additional literature. For further reading we recommend Golden and Stewart (1985), Bentley (1992), Johnson (1990) and Reinelt (1992, 1994).

An important point is the discussion of implementation issues. Although sometimes easily formulated, heuristics will often require extensive effort to obtain computer implementations that are applicable in practice. We will address these questions along with the presentation of the heuristics. We do not discuss techniques in detail. The reader should consult a good reference on algorithms and data structures (e.g., Cormen, Leiserson and Rivest (1989)) when doing own implementations.

Due to limited space we will not present many detailed computational results, rather we will give conclusions that we have drawn from computational testing. For our experiments we used problem instances from the public problem library TSPLIB (Reinelt (1991a, 1991b)). In this chapter we refer to a set of 30 Euclidean sample problems with sizes ranging from 105 to 2392 with known optimal solutions. The size of each problem instance appears in its name, e.g., *pcb442* is a TSP on 442 nodes. Since these problems come from real applications, our findings may be different from experiments on randomly generated problems. CPU times are given in seconds on a SUN SPARCstation 10/20. Some effort was put into the implementation of computer codes. However, it was not our intention to achieve ultimate performance, but to demonstrate the speedup that can be gained by careful implementation. Except for specially selected edges, distances were not stored but always computed by evaluating the Euclidean distance function. All CPU times include the time for distance computations.

Before starting to derive approximation algorithms, it is an interesting theoretical question, whether efficient heuristics can be designed that produces solutions with requested or at least guaranteed quality in polynomial time (polynomial in the problem size and in the desired accuracy). Whereas for other $\mathcal{NP}$-hard problems such heuristics do exist, there are only negative results for the general TSP. For a problem instance let c_H denote the length of the Hamiltonian cycle produced by heuristic H and let c_{opt} denote the length of an optimal cycle. Sahni and Gonzales (1976) show that, unless $\mathcal{P}=\mathcal{NP}$, for any constant $r \geqslant 1$ there does

not exist a polynomial time heuristic H such that $c_H \leqslant r \cdot c_{\text{opt}}$ for all problem instances.

A *fully polynomial approximation scheme* for a minimization problem is a heuristic H that for a given problem instance and any $\varepsilon > 0$ computes a feasible solution satisfying $c_H/c_{\text{opt}} \leqslant 1 + \varepsilon$ in time polynomial in the size of the instance and in $1/\varepsilon$. Such schemes are very unlikely to exist for the traveling salesman problem. Johnson and Papadimitriou (1985) show that, unless $\mathcal{P}{=}\mathcal{NP}$, there does not exist a fully polynomial approximation scheme for the Euclidean traveling salesman problem. This also holds in general for TSP instances satisfying the triangle inequality. The results tell us that for every heuristic there are problem instances where it fails badly. There are approximation results for problems satisfying the triangle inequality some of which will be addressed below.

It should be pointed out that running time and quality of an algorithm derived by theoretical (worst case or average case) analysis is usually insufficient to predict its behavior when applied to real-world problem instances.

In addition, the reader should be aware that polynomial time algorithms can still require a substantial amount of CPU time, if the polynomial is not of low degree. In certain applications algorithms having running time as low as $O(n^2)$ may not be acceptable. So, polynomiality by itself is not a sufficient criterion for efficiency in practice. It is our aim to show that, in the case of the traveling salesman problem, algorithms can be designed that are capable of finding good approximate solutions to even large sized real-world instances within rather moderate time limits. Thus, the $\mathcal{NP}$-hardness of the TSP does not imply the nonexistence of reasonable algorithms for practical problem instances. Furthermore, we want to make clear that designing efficient heuristics is not a straightforward task. Although ideas often seem elementary, it requires substantial effort to design practically useful computer codes.

The performance of a heuristic is best assessed by comparing the value of the approximate solution it produces with the value of an optimal solution. We say that a heuristic solution value c_H has *quality* $p\%$ if $100 \cdot (c_H - c_{\text{opt}})/c_{\text{opt}} = p$. If no provably optimal solutions are known, then the quality can only be estimated from above by comparing the heuristic solution value with a lower bound for the optimal value. A frequently used such lower bound is the *subtour elimination lower bound* (see Section 5). This bound can be computed exactly using LP techniques or it can at least be approximated using iterative approaches to be discussed in Section 6.

4.1. Construction heuristics

For the beginning we shall consider pure *construction procedures*, i.e., heuristics that determine a Hamiltonian cycle according to some construction rule, but do not try to improve upon this cycle. In other words, a Hamiltonian cycle is successively built, and parts already built remain in a certain sense unchanged throughout the algorithm. We will confine ourselves to some of the most commonly used construction principles.

Nearest neighbor heuristics

One of the simplest heuristics for the TSP is the so-called *nearest neighbor heuristic* which attempts to construct Hamiltonian cycles based on connections to near neighbors. The standard version is stated as follows.

procedure NEAREST_NEIGHBOR

(1) Select an arbitrary node j, set $l = j$ and $W = \{1, 2, \ldots, n\} \setminus \{j\}$.
(2) As long as $W \neq \emptyset$ do the following.
  (2.1) Let $j \in W$ such that $c_{lj} = \min\{c_{li} \mid i \in W\}$.
  (2.2) Connect l to j and set $W = W \setminus \{j\}$ and $l = j$.
(3) Connect l to the node selected in Step (1) to form a Hamiltonian cycle.

A possible variation of the standard nearest neighbor heuristic is the *double-sided nearest neighbor heuristic* where the current path can be extended from both of its endnodes.

The standard procedure runs in time $O(n^2)$. No constant worst case performance guarantee can be given. In fact, Rosenkrantz, Stearns and Lewis (1977) show that for arbitrarily large n there exist TSP instances on n nodes such that the nearest neighbor solution is $\Theta(\log n)$ times as long as an optimal Hamiltonian cycle. This also holds if the triangle inequality is satisfied.

If one displays nearest neighbor solutions one realizes the reason for this poor performance. The procedure proceeds very well and produces connections with short edges in the beginning. But as can be seen from a graphics display, several nodes are "forgotten" during the algorithm and have to be inserted at high cost in the end.

Although usually rather bad, nearest neighbor solutions have the advantage that they only contain a few severe mistakes. Therefore, they can serve as good starting solutions for subsequently performed improvement methods, and it is reasonable to put some effort in designing heuristics that are based on the nearest neighbor principle. For nearest neighbor solutions we obtained an average quality of 24.2% for our set of sample problems (i.e., on the average Hamiltonian cycles were 1.242 times longer than an optimal Hamiltonian cycle). In Johnson (1990) an average excess of 24% over an approximation of the subtour elimination lower bound is reported for randomly generated problems.

The standard procedure is easily implemented with a few lines of code. But, since running time is quadratic, this implementation can be too slow for large problems with 10,000 or 100,000 nodes, say. Therefore, even for this simple heuristic, it is worthwhile to think about speedup possibilities.

A basic idea, that we will apply for other heuristics as well, is the use of a *candidate subgraph*. A candidate subgraph is a subgraph of the complete graph on n nodes containing reasonable edges in the sense that they are "likely" to be contained in a short Hamiltonian cycle. These edges are taken with priority in the various heuristics, thus avoiding the consideration of the majority of edges that are assumed to be of no importance. For the time being we do not address

the question of how to choose such subgraphs and of how to compute them efficiently. This will be discussed in the subsection on geometric instances.

Because a major problem with nearest neighbor heuristics is that, in the end, nodes have to be connected at high cost, we modify it to avoid isolated nodes. To do this we first compute the 10 nearest neighbor subgraph, i.e., the subgraph containing for every node all edges to its 10 nearest neighbors. Whenever a node is connected to the current path we remove its incident edges in the subgraph. As soon as a node not contained in the path so far is connected to fewer than four nodes in the subgraph, we insert that node immediately into the path (eliminating all of its incident edges from the subgraph). To reduce the search for an insertion point, we only consider insertion after or before those nodes of the path that are among the 10 nearest neighbors of the node to be inserted. If all isolated nodes are added, the selection of the next node to be appended to the path is accomplished as follows. We first look for nearest neighbors of the node within its adjacent nodes in the candidate subgraph. If all nodes adjacent in the subgraph are already contained in the partial Hamiltonian cycle then we compute the nearest neighbor among all free nodes. The worst case time complexity is not affected by this modification.

Substantially less CPU time was needed to perform the modified heuristic compared to the standard implementation (even if the preprocessing time to compute the candidate subgraph is included). For example, whereas it took 15.3 seconds to perform the standard nearest neighbor heuristic for problem *pr2392*, the improved version required a CPU time of only 0.3 seconds. As expected, however, the variant still seems to have a quadratic component in its running time.

With respect to quality, insertion of forgotten nodes indeed improves the length of the nearest neighbor solutions. In contrast to the quality of 24.2% on average for the standard implementation, the modified version gave the average quality 18.6%. In our experiments we have chosen the starting node at random. The performance of nearest neighbor heuristics is very sensitive to the choice of the starting node. Choosing a different starting node can result in a solution whose quality differs by more than 10 percentage points.

Insertion heuristics

Another intuitively appealing approach is to start with cycles visiting only small subsets of the nodes and then extend these cycles by inserting the remaining nodes. Using this principle, a cycle is built containing more and more nodes of the problem until all nodes are inserted and a Hamiltonian cycle is found.

procedure INSERTION

(1) Select a starting cycle on k nodes $v_1, v_2, \ldots, v_k$ ($k \geqslant 1$) and set $W = V \setminus \{v_1, v_2, \ldots, v_k\}$.
(2) As long as $W \neq \emptyset$ do the following.
  (2.1) Select a node $j \in W$ according to some criterion.
  (2.2) Insert j at some position in the cycle and set $W = W \setminus \{j\}$.

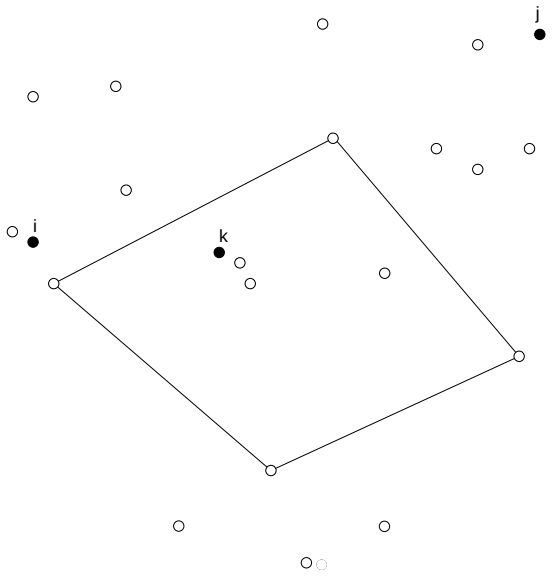

Figure 1. Insertion heuristics.

Of course, there are several possibilities for implementing such an insertion scheme. The main difference is the choice of the selection rule in Step (2.1). The starting cycle can be just some cycle on three nodes or, in degenerate cases, a loop ($k = 1$) or an edge ($k = 2$). The selected node to be inserted is usually inserted into the cycle at the point causing shortest increase of the length of the cycle.

The following are some choices for extending the current cycle (further variants are possible). We say that a node is a *cycle node* if it is already contained in the partial Hamiltonian cycle. For $j \in W$ we define $d_{\min}(j) = \min\{c_{ij} \mid i \in V \setminus W\}$.

NEAREST INSERTION: Insert the node that has the shortest distance to a cycle node, i.e., select $j \in W$ with $d_{\min}(j) = \min\{d_{\min}(l) \mid l \in W\}$.

FARTHEST INSERTION: Insert the node whose minimum distance to a cycle node is maximum, i.e., select $j \in W$ with $d_{\min}(j) = \max\{d_{\min}(l) \mid l \in W\}$.

CHEAPEST INSERTION: Insert the node that can be inserted at the lowest increase in cost.

RANDOM INSERTION: Select the node to be inserted at random and insert it at the best possible position.

Figure 1 visualizes the difference between the insertion schemes. Nearest insertion adds node i to the partial Hamiltonian cycle in the following step, farthest insertion chooses node j, and cheapest insertion chooses node k.

All heuristics except for cheapest insertion are easily implementable to run in time $O(n^2)$. Cheapest insertion can be executed in time $O(n^2 \log n)$ by storing for each external node a heap based on the insertion cost at the possible insertion points. Due to an $O(n^2)$ space requirement it cannot be used for large instances.

For some insertion type heuristics we have worst-case performance guarantees. For instances of the TSP obeying the triangle inequality, Hamiltonian cycles computed by the nearest insertion and cheapest insertion heuristic are less than twice as long as an optimal Hamiltonian cycle (Rosenkrantz, Stearns and Lewis (1977)). The result is sharp in the sense that there exist instances for which these heuristics yield solutions that are $2 - 2/n$ times larger than an optimal solution. Hurkens (1991) gave examples where random and farthest insertion yield Hamiltonian cycles that are $13/2$ times longer than an optimal Hamiltonian cycle (although the triangle inequality is satisfied).

On our set of sample problems we obtained average qualities 20.0%, 9.9%, and 11.1% for nearest, farthest, and random insertion, respectively. An average excess over the subtour bound of 27% for the nearest insertion and of 13.5% for the farthest insertion procedure is reported in Johnson (1990) for random problem instances. Though appealing at first sight, the cheapest insertion heuristic only yields an average quality of 16.8% (with substantially longer running times). Performance of insertion heuristics does not depend as much on the starting configuration as in the nearest neighbor heuristic. One can expect deviations of about 6% for the random insertion variant and about 7-8% for the others.

There are also variants of the above ideas where the node selected is not inserted at cheapest insertion cost but as a neighbor of the cycle node that is nearest to it. These variants are usually named "addition" instead of insertion. Bentley (1992) reports that the results are slightly inferior.

Heuristics based on spanning trees

The heuristics considered so far construct Hamiltonian cycles "from scratch" in the sense that they do not exploit any additional knowledge. The two heuristics to be described next use a minimum spanning tree as a basis for generating Hamiltonian cycles. They are particularly suited for TSP instances obeying the triangle inequality, but can, in principle, also be applied to general problems.

Before describing these heuristics we observe that, if the triangle inequality is satisfied, we can derive from any given tour a Hamiltonian cycle that is not longer than this tour. Let $v_{i_0}, v_{i_1}, \ldots, v_{i_k}$ be the sequence in which the nodes (including repetitions) are visited in the tour starting at v_{i_0} and returning to $v_{i_k} = v_{i_0}$. The following procedure obtains a Hamiltonian cycle.

procedure OBTAIN_CYCLE

(1) Set $T = \{v_{i_0}\}$, $v = v_{i_0}$, and $l = 1$.
(2) As long as $|T| < n$ perform the following steps.
  (2.1) If $v_{i_l} \notin T$ then set $T = T \cup \{v_{i_l}\}$, connect v to v_{i_l} and set $v = v_{i_l}$.
  (2.2) Set $l = l + 1$.
(3) Connect v to v_{i_0} to form a Hamiltonian cycle.

If the triangle inequality is satisfied, then every connection made in this procedure is either an edge of the tour or is a shortcut replacing a subpath of the tour by an edge connecting its two endnodes. Hence the resulting Hamiltonian cycle cannot be longer than the tour.

Both heuristics to be discussed next start with a minimum spanning tree and differ only in how a tour is generated from the tree.

procedure DOUBLETREE

(1) Compute a minimum spanning tree.
(2) Take all tree edges twice to obtain a tour.
(3) Call OBTAIN_CYCLE to get a Hamiltonian cycle.

The running time of the algorithm is dominated by the time needed to obtain a minimum spanning tree. Therefore we have time complexity $\Theta(n^2)$ for the general TSP and $\Theta(n \log n)$ for Euclidean problems (see Subsection 4.3).

If we compute the minimum spanning tree with Prim's algorithm (Prim (1957)), we could as well construct a Hamiltonian cycle along with the tree computation. We always keep a cycle on the nodes already in the tree (starting with the loop consisting of only one node) and insert the node into the current cycle which is added to the spanning tree. If this node is inserted at the best possible position this algorithm is identical to the nearest insertion heuristic. If it is inserted before or after its nearest neighbor among the cycle nodes, then we obtain the nearest addition heuristic.

In Christofides (1976) a more sophisticated method is suggested to make tours out of spanning trees. Namely, observe that it is sufficient to add a perfect matching on the odd-degree nodes of the tree. (A *perfect matching* of a node set W, $|W| = 2k$, is a set of k edges such that each node of W is incident to exactly one of these edges.) After addition of all matching edges all node degrees are even and hence the graph is a tour. The cheapest way (with respect to edge weights) to obtain a tour is to add a minimum weight perfect matching.

procedure CHRISTOFIDES

(1) Compute a minimum spanning tree.
(2) Compute a minimum weight perfect matching on the odd-degree nodes of the tree and add it to the tree to obtain a tour.
(3) Call OBTAIN_CYCLE to get a Hamiltonian cycle.

This procedure takes considerably more time than the previous one. Computation of a minimum weight perfect matching on k nodes can be performed in time $O(k^3)$ (Edmonds (1965)). Since a spanning tree may have $O(n)$ leaves, Christofides' heuristic has cubic worst case time.

Figure 2 displays the principle of this heuristic. Solid lines correspond to the edges of a minimum spanning tree, broken lines correspond to the edges of a perfect matching on the odd-degree nodes of this tree. The union of the two edge sets gives a tour. The sequence of the edges in the tour is not unique. So

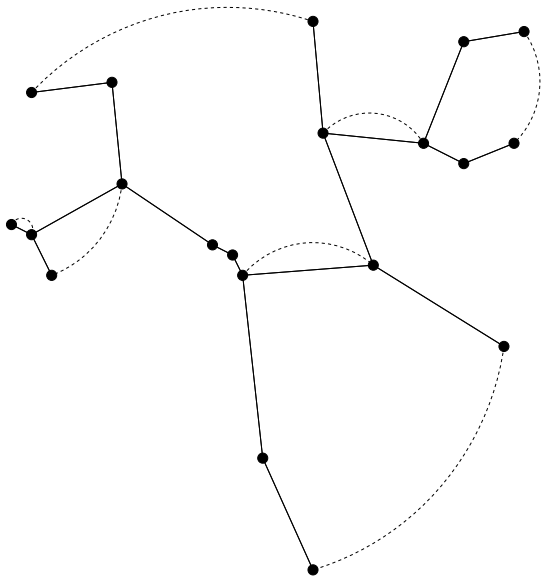

Figure 2. Christofides' heuristic.

one can try to find better solutions by determining different sequences. We do not elaborate on this since the gain to be expected is small.

A minimum spanning tree is not longer than a shortest Hamiltonian cycle and the matching computed in Step (2) of CHRISTOFIDES has weight at most half of the length of an optimal Hamiltonian cycle. Therefore, for every instance of the TSP obeying the triangle inequality, the double tree heuristic produces a solution which is at most twice as large as an optimal solution, and Christofides' heuristic produces a solution which is at most 1.5 times as large as an optimal solution. Cornuéjols and Nemhauser (1978) show that there are instances where Christofides' heuristic yields a Hamiltonian cycle that is $1.5 - 1/(2n)$ times longer than an optimal cycle.

It was observed that computing exact minimum weight perfect matchings in Step (2) does not pay off. Because of this and for reducing the running time, the necessary matching is usually computed by a heuristic. We have used the following one. First we double all edges incident with the leaves of the spanning tree, and then we compute a farthest insertion cycle on the remaining (and newly introduced) odd-degree nodes. This cycle induces two perfect matchings and we add the shorter one to our subgraph. Time complexity is reduced to $O(n^2)$ this way. It was observed in many experiments that the Christofides heuristic does not perform as well as it might have been expected. Although it has the best known worst case bound of any TSP heuristic, the experiments produced solutions which rarely yield qualities better than 10%.

For our set of sample problems the average quality was 38.08% for the double tree and 19.5% for the modified Christofides heuristic which coincides with the

findings in Johnson (1990). Running times for *pr2392* were 0.2 seconds for the double tree and 0.7 seconds for the modified Christofides heuristic (not including the time for the minimum spanning tree computation).

Savings methods

The final type of heuristic to be discussed in this subsection was originally developed for vehicle routing problems (Clarke and Wright (1964)). But it can also be usefully applied in our context, since the traveling salesman problem can be considered as a special vehicle routing problem involving only one vehicle with unlimited capacity. This heuristic successively merges subtours to eventually obtain a Hamiltonian cycle.

procedure SAVINGS

(1) Select a base node $z \in V$ and set up the $n-1$ subtours (z, v), $v \in V \setminus \{z\}$ consisting of two nodes each.
(2) As long as more than one subtour is left perform the following steps.
  (2.1) For every pair of subtours T_1 and T_2 compute the savings that is achieved if they are merged by deleting in each of them an edge to the base node and connecting the two open ends.
  (2.2) Merge the two subtours which provide the largest savings. (Note that this operation always produces a subtour which is a cycle.)

An iteration step of the savings heuristic is depicted in Figure 3. Two subtours are merged by deleting the edges from nodes i and j to the base node z and adding the edge ij.

In the implementation we have to maintain a list of possible mergings. The crucial point is the update of this list. We can consider the system of subtours as a system of paths (possibly having only one node) whose endnodes are thought of as being connected to the base node. A merge operation essentially consists of connecting two ends of different paths. For finding the best merge possibility we have to know for each endnode its best possible connection to an endnode of another path ("best" with respect to the cost of merging the corresponding subtours). Suppose that in Step (2.2) the two paths $[i_1, i_2]$ and $[j_1, j_2]$ are merged by connecting i_2 to j_1. The best merging now changes only for those endnodes whose former best merging was the connection to i_2 or to j_1, or for the endnode i_1 (j_2) if its former best merging was to j_1 (i_2). Because we do not know how many nodes are affected we can only bound the necessary update time by $O(n^2)$ giving an overall heuristic with running time $O(n^3)$. For small problems we can achieve running time $O(n^2 \log n)$, but we have to store the matrix of all possible savings which requires $O(n^2)$ storage space. Further remarks on the Clarke/Wright algorithm can be found in Potvin and Rousseau (1990). The average quality that we achieved for our set of problems was 9.8%.

We apply ideas similar to those above to speed up this heuristic. We again assume that we have a candidate subgraph of reasonable connections at hand. Now, merge operations are preferred that use a candidate edge for connecting

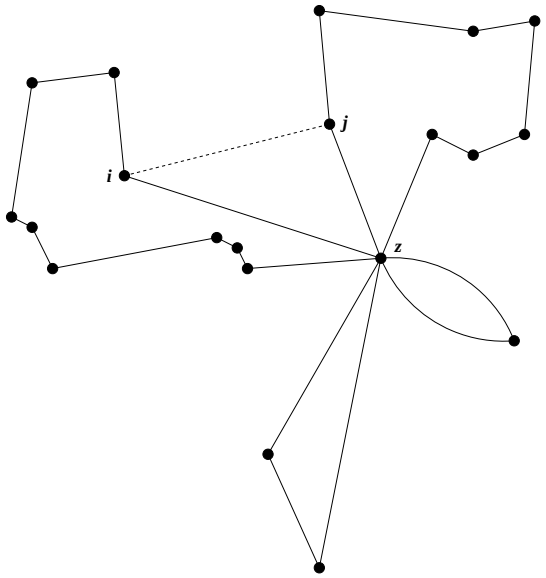

Figure 3. A savings heuristic.

two paths. The update is simplified in that for a node whose best merge possibility changes, only candidate edges incident to that node are considered for connections. If during the algorithm an endnode of a path becomes isolated, since none of its incident subgraph edges is feasible anymore, we compute its best merging by enumeration.

Surprisingly, the simplified heuristic yields solutions of similar average quality (9.6%) in much shorter time. For problem *pr2392* CPU time was 5.1 seconds with quality 12.4% compared to 194.6 seconds and quality 12.2% for the original version. We have also conducted experiments concerning the stability of this heuristic depending on the choice of the base node. It turns out that the savings heuristic gives much better results and is more stable than nearest neighbor or insertion heuristics.

Often, we will not apply the savings heuristic for constructing Hamiltonian cycles from scratch. Our purpose for employing this heuristic is to connect systems of paths in the following way. If we have a collection of paths we join all endnodes to a base node and proceed as in the usual heuristic. If we have long paths then the heuristic is started with few subtours and the necessary CPU time will be acceptable even without using more sophisticated speed up methods. If the paths are close to an optimal Hamiltonian cycle, we can obtain very good results.

This concludes our survey of construction heuristics suitable for general traveling salesman problem instances. In the special case of geometric instances there are further ideas that can be employed. Some of these are addressed in Subsection 4.3.

Table 1

Results of construction heuristics (Quality)

Problem	NN1	NN2	FI	CH	SA
lin105	33.31	10.10	11.22	19.76	5.83
pr107	6.30	9.20	2.13	8.95	9.22
pr124	21.02	8.16	11.87	16.49	4.20
pr136	34.33	17.90	8.59	27.83	6.73
pr144	4.96	13.68	3.12	15.55	9.97
pr152	19.53	20.44	4.24	19.75	9.44
u159	15.62	30.43	10.34	20.95	12.05
rat195	17.86	16.23	12.87	24.41	5.42
d198	25.79	17.57	3.85	15.40	6.96
pr226	22.76	20.87	1.42	20.95	8.93
gil262	25.95	19.47	5.93	19.05	8.86
pr264	20.32	17.38	9.12	17.60	10.56
pr299	27.77	19.71	9.13	19.93	11.95
lin318	26.85	18.68	10.87	18.42	8.24
rd400	23.13	23.37	9.61	21.48	9.00
pr439	27.04	15.74	12.24	17.39	13.31
pcb442	21.36	16.09	13.83	18.59	10.20
d493	28.52	17.82	11.61	17.44	8.84
u574	29.60	19.20	11.39	20.02	12.36
rat575	24.82	18.81	10.20	21.87	9.07
p654	31.02	27.99	6.89	21.73	10.66
d657	31.26	16.66	11.87	17.50	10.20
u724	23.16	20.34	11.65	21.00	10.44
rat783	27.13	18.66	12.09	21.34	9.88
pr1002	24.35	24.28	10.85	20.67	10.24
pcb1173	28.18	19.00	14.22	18.77	10.53
rl1304	28.58	21.59	17.81	15.92	9.86
nrw1379	24.43	18.89	9.71	24.14	10.54
u1432	25.50	19.07	12.59	24.05	10.41
pr2392	24.96	20.27	14.32	18.70	12.40

Table 1 contains for our sample problem set the qualities of the solutions (i.e., the deviations in percent from an optimal solution) found by the standard nearest neighbor heuristic started at node $\lfloor n/2 \rfloor$ (NN1), the variant of the nearest neighbor heuristic using candidate sets started at node $\lfloor n/2 \rfloor$ (NN2), the farthest insertion heuristic started with the loop $\lfloor n/2 \rfloor$ (FI), the modified Christofides heuristic (CH), and the savings heuristic with base node $\lfloor n/2 \rfloor$ (SA). All heuristics (except for the standard nearest neighbor and the farthest insertion heuristic) were executed in their fast versions using the 10 nearest neighbor candidate subgraph. Table 2 lists the corresponding CPU times (without times for computing the candidate sets).

Table 2

CPU times for construction heuristics

Problem	NN1	NN2	FI	CH	SA
lin105	0.03	0.01	0.06	0.01	0.02
pr107	0.03	0.01	0.07	0.01	0.03
pr124	0.05	0.01	0.10	0.01	0.02
pr136	0.05	0.02	0.10	0.01	0.03
pr144	0.06	0.02	0.13	0.01	0.03
pr152	0.06	0.02	0.13	0.01	0.06
u159	0.07	0.01	0.16	0.01	0.03
rat195	0.11	0.02	0.23	0.01	0.04
d198	0.10	0.02	0.23	0.02	0.05
pr226	0.13	0.02	0.30	0.02	0.08
gil262	0.18	0.03	0.40	0.02	0.07
pr264	0.19	0.03	0.43	0.02	0.11
pr299	0.24	0.04	0.52	0.02	0.07
lin318	0.27	0.04	0.59	0.03	0.09
rd400	0.42	0.05	0.94	0.05	0.13
pr439	0.51	0.05	1.12	0.04	0.20
pcb442	0.51	0.05	1.14	0.04	0.15
d493	0.64	0.07	1.43	0.05	0.20
u574	0.95	0.07	1.93	0.07	0.28
rat575	0.93	0.08	1.94	0.06	0.24
p654	1.19	0.09	2.51	0.07	0.53
d657	1.14	0.09	2.56	0.06	0.34
u724	1.49	0.10	3.10	0.10	0.39
rat783	1.63	0.11	3.63	0.11	0.48
pr1002	2.63	0.14	6.02	0.17	0.86
pcb1173	3.65	0.16	8.39	0.17	1.12
rl1304	4.60	0.18	10.43	0.13	1.85
nrw1379	5.16	0.22	11.64	0.30	1.70
u1432	5.54	0.17	12.52	0.34	1.64
pr2392	15.27	0.33	35.42	0.72	5.07

From our computational experiments with these construction heuristics we have drawn the following conclusions. The clear winners are the savings heuristics, and because of the considerably lower running time we declare the fast implementation of the savings heuristic as the best construction heuristic. This is in conformity with other computational testings, for example Arthur and Frendeway (1985). If one has to employ a substantially faster heuristic then one should use the variant of the nearest neighbor heuristic where forgotten nodes are inserted. For geometric problems minimum spanning trees are readily available. In such a case the fast variant of Christofides' heuristic can be used instead of the nearest neighbor variant.

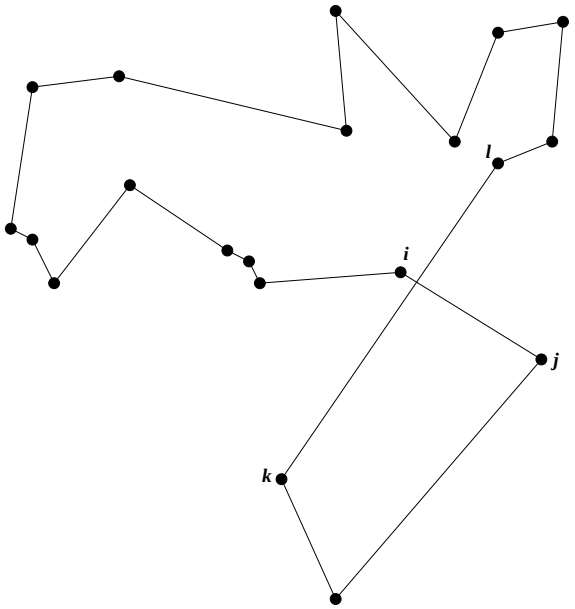

Figure 4. A 2-opt move.

4.2. Improvement heuristics

The Hamiltonian cycles computed by the construction heuristics in the previous subsection were only of moderate quality. Although they can be useful for some applications, they are not satisfactory in general.

In this subsection we address the question of how to improve these cycles. In general, the heuristics we will discuss here are defined using a certain type of basic move to alter the current cycle. We will proceed from fairly simple moves to more complicated ones. Further types of moves can be found in Gendreau, Hertz and Laporte (1992).

Two-opt exchange

This improvement approach is motivated by the following observation for Euclidean problems. If a Hamiltonian cycle crosses itself it can be easily shortened. Namely, erase two edges that cross and reconnect the resulting two paths by edges that do not cross (this is always possible). The new cycle is shorter than the old one.

A *2-opt move* consists of eliminating two edges and reconnecting the two resulting paths in a different way to obtain a new cycle. The operation is depicted in Figure 4, where we obtain a better solution if edges ij and kl are replaced by edges ik and jl. Note that there is only one way to reconnect the paths, since adding edges il and jk would result in two subtours. The 2-opt improvement heuristic is then outlined as follows.

procedure 2-OPT

(1) Let T be the current Hamiltonian cycle.
(2) Perform the following until *failure* for every node i is obtained.
  (2.1) Select a node i.
  (2.2) Examine all 2-opt moves involving the edge between i and its successor in the cycle. If it is possible to decrease the cycle length this way, then choose the best such move, otherwise declare *failure* for node i.
(3) Return T.

Assuming integral data, the procedure runs in finite time. But, there are classes of instances where the running time cannot be bounded by a polynomial in the input size. Checking whether an improving 2-opt move exists takes time $O(n^2)$ because we have to consider all pairs of cycle edges.

The implementation of 2-OPT can be done in a straightforward way. But, observe that it is necessary to have an imposed direction on the cycle to be able to decide which two edges have to be added in order not to generate subtours. Having performed a move, the direction has to be reversed for one part of the cycle. CPU time can be saved if the update of the imposed direction is performed such that the direction on the longer path is maintained and only the shorter path is reversed. One can incorporate this shorter path update by using an additional array giving the rank of the nodes in the current cycle (an arbitrary node receives rank 1, its successor gets rank 2, etc.). Having initialized these ranks we can determine in constant time which of the two paths is shorter, and the ranks have to be updated only for the nodes in the shorter path. With such an implementation it still took 88.0 seconds to perform the 2-opt heuristic on a nearest neighbor solution for problem *pr2392*. The quality of the final solution was 8.3%.

Speedup possibilities are manifold. First of all, we can make use of a candidate subgraph. The number of possible 2-opt moves that are examined can then be reduced by requiring that in every 2-opt move at least one candidate edge is used to reconnect the paths.

Another modification addresses the order in which cycle edges are considered for participating in a 2-opt move. A straightforward strategy could use a fixed enumeration order, e.g., always scanning the nodes in Step (2.1) of the heuristic in the sequence $1, 2, \ldots, n$ and checking if a move containing the edge from node i to its successor in the current cycle can participate in an allowed move (taking restrictions based on the candidate set into account). But usually, one observes that, in the neighborhood of a successful 2-opt move, more improving moves can be found. The fixed enumeration order does not consider this. We have therefore implemented the following dynamic order. The nodes of the problem are stored in a list (initialized according to the sequence of the nodes in the cycle). In every iteration step the first node is taken from the list, scanned as described below, and reinserted at the end of the list. If i is the current node to be scanned, we examine if we can perform an improving 2-opt move which introduces a candidate edge having i as one endnode. If an improving move is found then

all four nodes involved in that move are stored at the beginning of the node list (and therefore reconsidered with priority). The reduction in running time is considerable, because many fewer moves are examined. For example, when starting with a random Hamiltonian cycle for problem *rl5934*, with the dynamic enumeration only 85,762,731 moves were considered instead of 215,811,748 moves with the fixed enumeration. The reduction is less significant if one starts the 2-opt improvement with reasonable starting solutions. Since the 2-opt heuristic is very sensitive with respect to the sequence in which moves are performed, one can obtain quite different results for the two versions even for the same start. However, with respect to average quality both variants perform equally well.

Another point for speeding up computations further is to reduce the number of distance function evaluations, which accounts for a large portion of the running time. A thorough discussion of this issue can be found in Bentley (1992). For example, one can inhibit evaluation of a 2-opt move that cannot be improving in the following way. When considering a candidate edge ij for taking part in a 2-opt move, we check if i and j have the same neighbors in the cycle as when ij was considered previously. If ij could not be used before in an improving move it can also not be used now. Furthermore, one can restrict attention to those moves where one edge ij is replaced by a shorter edge ik, since this must be true for one of the pairs.

Using an implementation of 2-opt based on the above ideas we can now perform the heuristic on a nearest neighbor solution for *pr2392* in 0.4 seconds achieving a Hamiltonian cycle of length 9.5% above the optimum. The average quality for our set of sample problems was 8.3%.

Performance of 2-opt can be improved by incorporating a simple additional move, namely *node insertion*. Such a move consists of removing one node from the current cycle and reinserting it at a different position. Since node insertion is not difficult to implement, we suggest to combine 2-opt and node insertion. On our set of sample problems we achieved an average quality of 6.5% using this combination. For problem *pr2392* we obtained a solution with quality 7.3% in 2.2 seconds. With his 2-opt implementation starting with a nearest neighbor solution Johnson (1990) achieved an excess of 6.5% over an approximation of the subtour bound. Bentley (1992) reports an excess of 8.7% for 2-opt and of 6.7% for a combination of 2-opt and node insertion. In both cases classes of random problems were used.

A further general observation for speeding up heuristics is the following. Usually, decrease in the objective function value is considerable in the first steps of the heuristic and then tails off. In particular, it takes a final complete round through all allowed moves to verify that no further improving move is possible. Therefore, if one stops the heuristics early (e.g., if only a very slow decrease is observed over some period) not too much quality is lost.

The 3-opt heuristic and variants

To have more flexibility for modifying the current Hamiltonian cycle we could break it into three parts instead of only two and combine the resulting paths

in the best possible way. Such a modification is called *3-opt move*. The number of combinations to remove three edges of the cycle is $\binom{n}{3}$, and there are eight ways to connect three paths to form a cycle (if each of them contains at least one edge).

Note that node insertion and 2-opt exchange are special 3-opt moves. Node insertion is obtained if one path of the 3-opt move consists of just one node, a 2-opt move is a 3-opt move where one eliminated edge is used again for reconnecting the paths.

To examine all 3-opt moves takes time $O(n^3)$. Update after a 3-opt move is also more complicated than in the 2-opt case. The direction of the cycle may change on all but the longest of the three involved paths.

Therefore we decided to not consider full 3-opt (which takes 4661.2 seconds for problem *pcb442* when started with a nearest neighbor solution), but to limit in advance the number of 3-opt moves that are considered. The implemented procedure is the following.

procedure 3-OPT

(1) Let T be the current Hamiltonian cycle.
(2) For every node $i \in V$ define some set of nodes $N(i)$.
(3) Perform the following until *failure* is obtained for every node i.
  (3.1) Select a node i.
  (3.2) Examine all possibilities to perform a 3-opt move which eliminates three edges each having at least one endnode in $N(i)$. If it is possible to decrease the cycle length this way, then choose the best such move, otherwise declare *failure* for node i.
(4) Return T.

If we limit the cardinality of $N(i)$ by some fixed constant independent of n then checking in Step (3.2) if an improving 3-opt move exists at all takes time $O(n)$ (but with a rather large constant hidden by the O-notation).

We implemented the 3-opt routine using a dynamic enumeration order for node selection and maintaining the direction of the cycle on the longest path. Search in Step (3.2) is terminated as soon as an improving move is found. For a given candidate subgraph G_C we defined $N(i)$ as the set of all neighbors of i in G_C. In order to limit the CPU time (which is cubic in the cardinality of $N(i)$ for Step (3.2)) the number of nodes in each set $N(i)$ is bounded by 50 in our implementation.

With this restricted 3-opt version we achieved an average quality of 3.8% when started with a nearest neighbor solution and of 3.9% when started with a random solution (G_C was the 10 nearest neighbor subgraph augmented by the Delaunay graph to be defined in 4.3). CPU time is significantly reduced compared to the full version. Time for *pcb442* is now 18.2 seconds with the nearest neighbor start. For their respective versions of 3-opt, an excess of 3.6% over an approximation of the subtour bound is reported in Johnson (1990) and an excess of 4.5% was achieved in Bentley (1992).

Also the restricted versions of 3-opt are rather time consuming. Thus it is worthwhile to consider variants that reduce running time further. One particular variant is the so-called *Or-opt* procedure (Or (1976)). In this variant of 3-opt it is required that one of the paths involved in the move has exactly l edges. Results obtained with this procedure lie between 2-opt and 3-opt (as can be expected) and it does not contribute significantly to the quality of the final solution if values larger than 3 are used for l.

Better performance than with the 3-opt heuristic can be obtained with general *k-opt* exchange moves, where k edges are removed from the cycle and the resulting paths are reconnected in the best possible way. A complete check of the existence of an improving k-opt move takes time $O(n^k)$ and is therefore only applicable for small problems. One can, of course, design restricted searches for higher values of k in the same way as we did for $k = 3$. For a discussion of update aspects see Margot (1992).

One might suspect that with increasing k the k-opt procedure should yield provably better approximations to the optimal solution. However, Rosenkrantz, Stearns and Lewis (1977) show that for every $n \geqslant 8$ and every $k \leqslant n/4$ there exists a TSP instance on n nodes and a k-optimal solution such that the optimal and k-optimal values differ by a factor of $2 - 2/n$. Nevertheless, this is only a worst case result. One observes that for practical applications it does pay to consider larger values of k and design efficient implementations of restricted k-opt procedures.

Lin-Kernighan type exchange

The final heuristic to be discussed in this subsection was originally described in Lin and Kernighan (1973). The motivation for this heuristic is based on experience gained from practical computations. Namely, one observes that the more flexible and powerful the possible cycle modifications, the better are the obtained results. In fact, simple moves quickly run into local optima of only moderate quality. On the other hand, the natural consequence of applying k-opt for larger k requires a substantially increasing running time. Therefore, it seems more reasonable to follow an approach suggested by Lin and Kernighan.

Their idea is based on the observation that sometimes a modification slightly increasing the cycle length can open up new possibilities for achieving considerable improvement afterwards. The basic principle is to build complicated modifications that are composed of simpler moves where not all of these moves necessarily have to decrease the cycle length. To obtain reasonable running times, the effort to find the parts of the composed move has to be limited. Many variants of this principle are possible. We do not describe the original version of this algorithm which contains a 3-opt component, but discuss a somewhat simpler version where the basic components are 2-opt and node insertion moves. General complexity issues concerning the Lin-Kernighan heuristic are addressed in Papadimitriou (1990).

When building a move, in each substep we have some node from which a new edge is added to the cycle according to some criterion. We illustrate our proce-

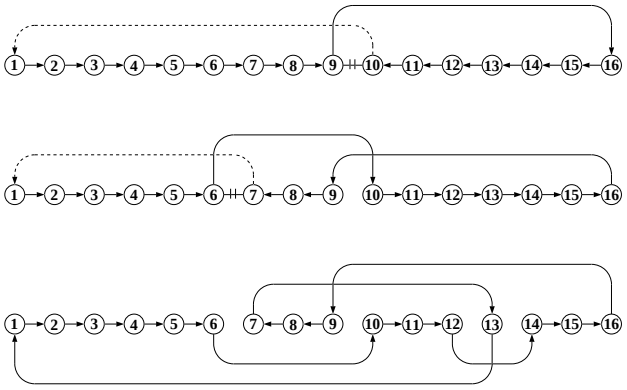

Figure 5. The Lin-Kernighan heuristic.

dure by the example of Figure 5. Suppose we start with the canonical Hamiltonian cycle $1, 2, \ldots, 16$ for a problem on 16 nodes and we decide to construct a modification starting from node 16.

In the first step it is decided to eliminate edge $(1, 16)$ and introduce the edge from node 16 to node 9. Adding this edge creates a subtour, and therefore edge $(9, 10)$ has to be deleted. To complete the cycle, node 10 is connected to node 1.

If we stop at this point we have simply performed a 2-opt move. The fundamental new idea is not to connect node 10 to node 1, but to search for another move starting from node 10. Suppose we now decide to add edge $(10, 6)$. Again, one edge, namely $(6, 7)$, has to be eliminated to break the subtour. The sequence of moves could be stopped here, if node 7 is joined to node 1.

As a final extension we perform a node insertion for node 13 instead, and place this node between 1 and 7. Thus we remove edges $(12, 13)$ and $(13, 14)$ and add edges $(12, 14)$, $(7, 13)$ and $(1, 13)$.

Note that the direction changes on some parts of the cycle while performing these moves and that these new directions have to be considered in order to be able to perform the next moves correctly. When building the final move we obtained three different solutions on the way. The best of these solutions (which is not necessarily the final one) can now be chosen as the new current Hamiltonian cycle.

Realization of this procedure is possible in various ways. We have chosen the following options.

– To speed up search for submoves a candidate subgraph is used. Edges to be added from the current node to the cycle are only taken from this set and are selected according to a local gain criterion. Let i be the current node. We define the local gain g_{ij} that is achieved by adding edge ij to the cycle as follows. If jk is the edge to be deleted if a 2-opt move is to be performed, then we set $g_{ij} = c_{jk} - c_{ij}$. If jk and jl are the edges to be deleted if a node insertion move is to be performed, then $g_{ij} = c_{jk} + c_{jl} - c_{lk} - c_{ij}$. The edge with the maximum local gain is chosen to enter the solution and the corresponding move is performed.

– The number of submoves in a move is limited in advance, and a dynamic enumeration order is used to determine the starting node for the next move.

– Examination of more than one candidate edge to enter the cycle is possible. The maximum number of candidates examined from the current node and the maximum number of submoves up to which alternative edges are taken into account are specified in advance. This option introduces an enumeration component for selecting the first few submoves.

The basic outline of the heuristic is then given as follows.

procedure LIN-KERNIGHAN

(1) Let T be the current Hamiltonian cycle.
(2) Perform the following computation until *failure* is obtained for every node i.
  (2.1) Select a node i to serve as a start for building a composed move.
  (2.2) Try to find an improving move starting from i according to the guidelines and the parameters discussed above. If no such move can be found, then declare *failure* for node i.
(3) Return T.

A central implementation issue concerns the management of the tentative moves. Since most of them do not lead to an improvement of the solution, it is reasonable to avoid an explicit cycle update for every such move, but only update as little information as possible.

We use an idea that was reported by Applegate, Chvátal and Cook (1990). Consider for example a 2-opt move. Its effect on the current solution is completely characterized by storing how the two resulting paths are reconnected and if their direction has changed. To this end it suffices to know the endnodes of every path and the edges connecting them. For every other node its neighbors are unchanged, and, since we have ranks associated with the nodes, we can easily identify the path in which a node is contained. In general, the current Hamiltonian cycle is represented by a cycle of intervals of ranks where each interval represents a subpath of the starting Hamiltonian cycle. For an efficient identification of the interval to which a specific node belongs the intervals are kept in a balanced binary search tree. Therefore, the interval containing a given node can be identified in time $O(\log m)$ if we have m intervals. Note, that also in the interval representation we have to reorient paths of the sequence. But, as long as we have few intervals (i.e., few tentative submoves), this can be done fast. Of course, the number of intervals should not become too large because the savings in execution time decreases with the number of intervals that have to be managed. Therefore, if we have too many intervals, we clear the interval structure and generate correct successor and predecessor pointers to represent the current cycle. The longest path represented as an interval can remain unchanged, i.e., for its interior nodes successors, predecessors, and ranks do not have to be altered.

Possible choices for the parameters of this heuristic are so numerous that we cannot document all experiments here. We only discuss some basic insights.

The observations we gained from our experiments can be summarized as follows.

– At least 15 submoves should be allowed for every move in order to be able to generate reasonably complicated moves.

– It is better not to start out with a random solution, but with a locally good Hamiltonian cycle. But, this is of less importance when more elaborate versions of the Lin-Kernighan procedure are used.

– It is advisable to consider several alternative choices for the edge to be added from the first node.

– Exclusion of node insertion moves usually leads to inferior results.

We report about two variants of the Lin-Kernighan approach for our set of sample problems.

In the first variant, the candidate subgraph is the 6 nearest neighbor subgraph augmented by the Delaunay graph. Up to 15 submoves are allowed to constitute a move. Three alternative choices for the edge to be added to the cycle in the first submove are considered. Submoves are 2-opt and node insertion moves.

In the second variant, the candidate subgraph is the 8 nearest neighbor subgraph augmented by the Delaunay graph. Up to 15 submoves are allowed to constitute a move. Two alternative entering edges are considered for each of the first three submoves of a move (This gives a total of eight possibilities examined for the first three submoves of a move). Submoves are 2-opt and node insertion moves.

In contrast to simpler heuristics, the dependence on the starting solution is not very strong. Results and CPU times differ only slightly for various types of starting solutions. Even if one starts with a random Hamiltonian cycle not much quality is lost. Starting with a nearest neighbor solution we obtained an average quality of 1.9% for variant 1 and 1.5% for variant 2. Running time of the Lin-Kernighan exchange for problem *pr2392* was 61.7 and 122.3 seconds for the respective variants. Variant 2 is more expensive since more possibilities for moves are enumerated (larger candidate set and deeper enumeration level). In general, higher effort usually leads to better results. Similar results are given in Johnson (1990). Another variant of the Lin-Kernighan heuristic is discussed in Mak and Morton (1993).

As a final experiment we ran an extension of the Lin-Kernighan heuristic first proposed by Johnson (1990). The Lin-Kernighan heuristic, as every other heuristic, terminates in a local optimum which depends on the start and on the moves that are performed. To have a chance of finding good local optima one can start the procedure several times with different starting solutions. A more reasonable idea is not to restart with a completely new starting solution but only to perturb the current solution. This way one escapes the local optimum by making a move that increases the length but still has a solution that is close to an optimal one at least in some segments. Computational experiments show that this approach is superior. Johnson (1990) suggests that after termination of the Lin-Kernighan heuristic a random 4-opt move is performed and the heuristic is reapplied. Using this method several optimal solutions of some larger problems

Table 3

Results of improvement heuristics

Problem	2-O	3-O	LK1	LK2	ILK
lin105	8.42	0.00	0.77	0.00	0.00
pr107	3.79	2.05	1.53	0.81	0.00
pr124	2.58	1.15	2.54	0.39	0.00
pr136	10.71	6.14	0.55	0.72	0.38
pr144	3.79	0.39	0.56	0.06	0.00
pr152	2.93	1.85	0.00	0.19	0.00
u159	14.00	11.49	2.20	1.59	0.00
rat195	6.46	3.01	1.55	1.55	0.47
d198	3.85	6.12	0.63	1.51	0.16
pr226	13.17	1.72	0.72	0.49	0.00
gil262	10.26	3.07	1.18	2.44	0.55
pr264	4.39	6.04	0.12	0.01	0.49
pr299	10.46	4.37	1.55	1.36	0.15
lin318	9.54	2.67	1.87	1.17	0.53
rd400	5.01	3.42	2.34	1.41	0.75
pr439	6.52	3.61	2.73	2.68	0.38
pcb442	8.74	3.01	1.41	1.94	0.90
d493	9.37	3.32	2.23	1.47	0.84
u574	7.85	4.61	2.05	0.98	0.60
rat575	7.93	4.46	2.48	1.68	1.03
p654	14.89	0.62	4.14	2.95	0.03
d657	7.57	3.52	3.10	1.65	0.74
u724	8.09	4.20	2.60	1.38	0.67
rat783	9.07	4.22	1.94	1.77	0.91
pr1002	8.46	3.80	2.92	2.72	1.51
pcb1173	10.72	5.26	2.18	3.22	1.46
rl1304	13.21	7.08	5.07	1.73	1.62
nrw1379	8.25	3.65	2.48	1.76	1.13
u1432	10.48	5.39	1.51	2.45	0.99
pr2392	9.48	5.26	2.95	2.90	1.75

(e.g., *pr2392*) were found. In our experiment we used the second variant of the Lin-Kernighan heuristic described above, but this time allowing 40 submoves per move. In addition, we performed a restricted 3-opt after termination of each Lin-Kernighan heuristic. This approach was iterated 20 times. We now obtained an average quality of 0.6%.

Table 3 gives the quality of the solutions found by 2-opt (2-O), 3-opt (3-O), and the two versions of the Lin-Kernighan heuristic described above (LK1 and LK2). Column (ILK) displays the results obtained with the iterated Lin-Kernighan heuristic. The improvement heuristics were started with a nearest neighbor solution. Table 4 lists the corresponding CPU times.

Table 4

CPU times for improvement heuristics

Problem	2-O	3-O	LK1	LK2	ILK
lin105	0.02	4.10	1.15	4.09	168.40
pr107	0.01	2.73	0.71	2.25	121.34
pr124	0.02	3.82	1.03	3.08	219.52
pr136	0.03	3.76	1.05	2.98	221.96
pr144	0.02	6.37	1.20	3.85	304.52
pr152	0.02	3.44	1.03	2.85	260.19
u159	0.02	6.03	1.46	4.26	314.53
rat195	0.02	4.41	1.93	4.86	409.74
d198	0.03	7.22	5.27	6.04	520.23
pr226	0.02	12.85	2.64	7.16	488.87
gil262	0.04	7.84	2.84	8.37	575.52
pr264	0.03	9.83	3.53	8.29	455.71
pr299	0.04	10.27	3.47	10.97	750.62
lin318	0.04	12.56	5.30	11.98	825.53
rd400	0.05	13.19	4.57	13.33	1153.41
pr439	0.06	14.60	7.34	16.04	1086.08
pcb442	0.08	18.23	5.03	17.60	1079.45
d493	0.08	17.69	8.88	15.89	1465.07
u574	0.10	34.12	7.92	27.09	1677.75
rat575	0.07	17.38	7.13	29.37	1547.93
p654	0.08	41.65	9.09	17.17	1303.30
d657	0.10	22.19	12.51	26.00	1958.84
u724	0.10	30.27	9.37	26.55	1921.41
rat783	0.14	27.50	12.78	39.24	2407.84
pr1002	0.19	41.69	17.78	42.01	2976.47
pcb1173	0.17	55.41	17.07	55.10	3724.98
rl1304	0.21	112.02	22.88	54.73	4401.12
nrw1379	0.26	52.68	21.22	73.58	4503.37
u1432	0.19	61.85	16.91	66.21	3524.59
pr2392	0.40	148.63	61.72	122.33	8505.42

From our computational tests we draw the following conclusions. If we want to achieve very good results, simple basic moves are not sufficient. If simple moves are employed, then it is advisable to apply them to reasonable starting solutions since they are not powerful enough for random starts. Nearest neighbor like solutions are best suited for simple improvement schemes since they consist of rather good pieces of a Hamiltonian cycle and contain few bad ones that can be easily repaired. For example, the 2-opt improvement heuristic applied to a farthest insertion solution would lead to much inferior results, although the farthest insertion heuristic delivers much better Hamiltonian cycles than those found by the nearest neighbor heuristic.

If one attempts to find solutions in the range of 1% above optimality one has to use the Lin-Kernighan heuristic since it can avoid bad local minima. However, applying it to large problems requires that considerable effort is spent in deriving an efficient implementation. A naïve implementation would consume an enormous amount of CPU time. If time permits, the iterated version of the Lin-Kernighan heuristic is the method of choice for finding good approximate solutions. For a more general discussion of local search procedures see Johnson, Papadimitriou and Yannakakis (1988).

It is generally observed that quality of heuristics degrades with increasing problem size, therefore more tries are necessary for larger problems. In Johnson (1990) and Bentley (1992) some interesting insights are reported for problems with up to a million nodes.

4.3. Special purpose algorithms for geometric instances

TSP instances that arise in practice often are of geometric nature in that the points defining the problem instance correspond to locations in a space. The length of the edge connecting nodes i and j is the distance of the points corresponding to the nodes according to some metric, i.e., a function that satisfies the triangle inequality. Usually the points are in 2-dimensional space and the metric is the Euclidean (L_2), the Maximum (L_∞), or the Manhattan (L_1) metric.

In this subsection we discuss advantages that can be gained from geometric instances. Throughout this subsection we assume that the points defining a problem instance correspond to locations in the plane and that the distance of two points is their Euclidean distance.

Geometric heuristics

Bartholdi and Platzman (1982) introduced the so-called *space filling curve heuristic* for problem instances in the Euclidean plane. It is particularly easy to implement and has some interesting theoretical properties. The heuristic is based on a bijective mapping $\psi : [0,1] \to [0,1] \times [0,1]$, a so-called *space filling curve*. The name comes from the fact that when varying the arguments of ψ from 0 to 1 the function values fill the unit square completely. Surprisingly, such functions exist and, what is interesting here, they can be computed efficiently and also for a given $y \in [0,1] \times [0,1]$ a point $x \in [0,1]$ such that $\psi(x) = y$ can be found in constant time.

The function used by Bartholdi and Platzman models the recursive subdivision of squares into four equally sized subsquares. The space filling curve is obtained by patching the four respective subcurves together. The heuristic is given as follows.

procedure SPACEFILL

(1) Scale the points to the unit square.
(2) For every point i with coordinates x_i and y_i compute z_i such that $\psi(z_i) = (x_i, y_i)$.

(3) Sort the numbers z_i in increasing order.
(4) Connect the points by a cycle respecting the sorted sequence of the z_i's (to complete the cycle connect the two points with smallest and largest z-value).

Since the values z_i can be computed in constant time the overall computation time is dominated by the time to sort these numbers in Step (3) and hence this heuristic runs in time $\Theta(n \log n)$.

It can be shown, that if the points are contained in a rectangle of area F then the Hamiltonian cycle is not longer than $2\sqrt{nF}$. Bartholdi and Platzman have also shown that the quotient of the length of the heuristic and the length of an optimal solution is bounded by $O(\log n)$.

At this point, we comment briefly on average case analysis for the Euclidean TSP. Suppose that the n points are drawn independently from a uniform distribution on the unit square and that c_{opt} denotes the length of an optimal solution. Beardwood, Halton and Hammersley (1959) show that there exists a constant C such that $\lim_{n\to\infty} c_{\text{opt}}/\sqrt{n} = C$ and they give the estimate $C \approx 0.765$. Such behavior can also be proved for the space filling curves heuristic with a different constant $\overline{C}$. Bartholdi and Platzman (1982) give the estimate $\overline{C} \approx 0.956$. Therefore, for this class of random problems the space filling curves heuristic can be expected to yield solutions that are approximately 25% larger than an optimal solution as n tends to infinity.

Since in space filling curves heuristic adding or deleting points does not change the relative order of the other points in the Hamiltonian cycle, this heuristic cannot be expected to perform too well. In fact, for our set of sample problems we achieved an average quality of 35.7%. As expected, running times are very low, e.g., 0.2 seconds for problem *pr2392*. Experiments show that space filling curves solutions are not suited as starts for improvement heuristics. They are useful if only extremely short computation times are allowed.

In the well-known *strip heuristic* the problem area is cut into $\sqrt{n}$ parallel vertical strips of equal width. Then Hamiltonian paths are constructed that collect the points of every strip sorted by the vertical coordinate, and finally these paths are combined to form a solution. The procedure runs in time $O(n \log n)$. Such a straightforward partition into strips is very useful for randomly generated problems, but can and will give poor results on real-world instances. The reason is that partition into parallel strips may not be adequate for the given point configuration.

To overcome this drawback, other approaches do not divide the problem area into strips but into segments, for example into squares or rectangles. In *Karp's partitioning heuristic* (Karp (1977)) the problem area is divided by horizontal and vertical cuts such that each segment contains no more than a certain number k of points. Then, a dynamic programming algorithm is used to compute an optimal Hamiltonian cycle on the points contained in each segment. In a final step all subtours are glued together according to some scheme to form a Hamiltonian cycle through all points. For fixed k the optimal solutions of the respective subproblems can be determined in linear time (however, depending on k,

a large constant associated with the running time of the dynamic programming algorithm is hidden).

We give another idea to reduce the complexity of a large scale problem instance. Here the number of nodes of the problem is reduced in such a way that the remaining nodes still give a satisfactory representation of the geometry of the original points. Then a Hamiltonian cycle on this set of representative nodes is computed in order to serve as an approximation for the cycle through all nodes. In the final step the original nodes are inserted into this cycle (where the number of insertion points that will be checked can be specified) and the representative nodes (if not original nodes) are removed. More precisely, we use the following bucketing procedure.

procedure NODE_REDUCTION

(1) Compute an enclosing rectangle for the given points.
(2) Recursively subdivide each rectangle into four equally sized parts by a horizontal and a vertical line until each rectangle
   - contains no more than 1 point, or
   - is the result of at least m recursive subdivisions and contains no more than k points.
(3) Represent each (nonempty) rectangle by the center of gravity of the points contained in it.
(4) Compute a Hamiltonian cycle through the representative nodes.
(5) Insert the original points into this cycle. To this end at most $l/2$ insertion points are checked before and after the corresponding representative nodes in the current cycle. The best insertion point is then chosen.
(6) Remove all representative nodes that are not original nodes.

The parameters m, k, and l, and the heuristic needed in Step (4) can be chosen with respect to the available CPU time.

This heuristic is only suited for very large problems and we did not apply it to our sample set. One can expect qualities of 15% to 25% depending on the point configuration.

The heuristic is similar to a clustering algorithm given in Litke (1984), where clusters of points are represented by a single point. Having computed an optimal Hamiltonian cycle through the representatives, clusters are expanded one after another. A further partitioning heuristic based on geometry is discussed in Reinelt (1994). Decomposition is also a topic of Hu (1967). Since many geometric heuristics are fairly simple, they are amenable to probabilistic analysis. Some interesting results on average behavior can be found in Karp and Steele (1985).

Success of such simple approaches is limited, because global view is lost and parts of the final solution are computed independently from each other. In Johnson (1990) a comparison of various geometric heuristics is given, concluding that the average excess over the subtour bound for randomly generated problems is 64.0%, 30.2%, and 23.2% for Karp's partitioning heuristic, strip heuristic, and Litke's clustering method, respectively. These results show that it is necessary

to incorporate more sophisticated heuristics into simple partitioning schemes as we did in our procedure. Then, one can expect qualities of about or below 20%. In any case, these approaches are very fast and can virtually handle arbitrary problem sizes. If the given point configuration decomposes in a natural way, then much better results can be expected.

Convex hull starts

Let $v_1, v_2, \ldots, v_k$ be those points of the problem defining the boundary of the convex hull of all given points (in this order). Then in any optimal Hamiltonian cycle this sequence is respected, otherwise it would contain crossing edges and hence could not be optimal. Therefore it is reasonable to use the cycle $(v_1, v_2, \ldots, v_k)$ as start for the insertion heuristics.

Convex hulls can be computed very quickly (in time $\Theta(n \log n)$, see e.g., Graham (1972)). Therefore, only negligible additional CPU time is necessary to compute a good starting cycle for the insertion heuristics in the Euclidean case.

It turns out that the quality of solutions delivered by insertion heuristics is indeed improved if convex hull start is used. But, gain in quality is only moderate. In particular, also with this type of start, our negative assessment of insertion heuristics still applies.

Delaunay graphs

A very powerful tool for getting insight into the geometric structure of a Euclidean problem is the Voronoi diagram, or its dual: the Delaunay triangulation. Although known for a long time (Voronoï (1908), Delaunay (1934)), these structures have only recently received significant attention in the literature on computation.

Let $S = \{P_1, P_2, \ldots, P_n\}$ be a finite subset of $\mathbb{R}^2$ and let $d : \mathbb{R}^2 \times \mathbb{R}^2 \longrightarrow \mathbb{R}$ denote the Euclidean metric. We define the *Voronoi region* $\mathrm{VR}(P_i)$ of a point P_i by $\mathrm{VR}(P_i) = \{P \in \mathbb{R}^2 \mid d(P, P_i) \leqslant d(P, P_j) \text{ for all } j = \{1, 2, \ldots, n\}\}$, i.e., $\mathrm{VR}(P_i)$ is the set of all points that are at least as close to P_i as to any other point of S. The set of all n Voronoi regions is called the *Voronoi diagram* $V(S)$ of S. Figure 6 shows the Voronoi diagram for a set of 15 points in the plane.

Given the Voronoi diagram of S, the *Delaunay triangulation* $G(S)$ is the undirected graph $G(S) = (S, D)$ where $D = \{\{P_i, P_j\} \mid \mathrm{VR}(P_i) \cap \mathrm{VR}(P_j) \neq \emptyset\}$. It is easy to see that $G(S)$ is indeed a triangulated graph.

In the following we use an alternative definition which excludes those edges (P_i, P_j) for which $|\mathrm{VR}(P_i) \cap \mathrm{VR}(P_j)| = 1$. In this case the name is misleading, because we do not necessarily have a triangulation anymore, and to avoid misinterpretation from now on we speak about the *Delaunay graph*. In contrast to the Delaunay triangulation defined above, the Delaunay graph is guaranteed to be a planar graph (implying $|D| = \mathrm{O}(n)$). Moreover, as the Delaunay triangulation, it contains a minimum spanning tree of the complete graph on S with edge weights $d(P_i, P_j)$ and contains for each node an edge to a nearest neighbor. Figure 7 shows the Delaunay triangulation corresponding to the Voronoi diagram displayed in Figure 6.

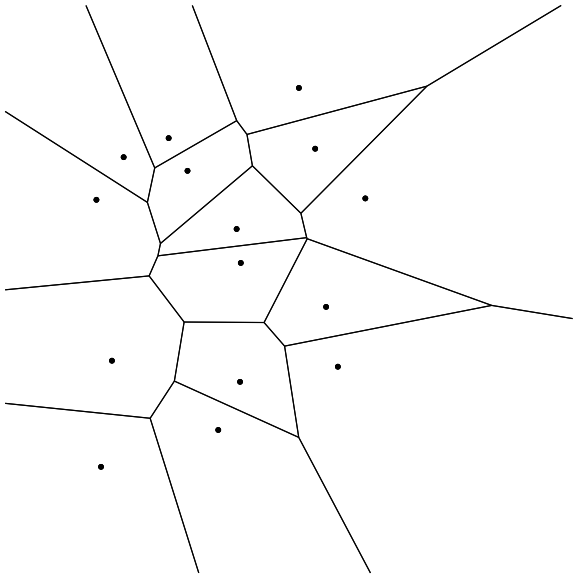

Figure 6. A Voronoi diagram.

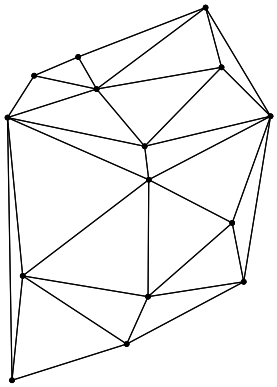

Figure 7. A Delaunay graph.

The Delaunay graph can be computed very efficiently. There are algorithms computing the Voronoi diagram (and hence the Delaunay triangulation) in time $O(n \log n)$ (see e.g., Shamos and Hoey (1975)). For practical purposes an algorithm given in Ohya, Iri and Murota (1984) seems to perform best. It has worst case running time $O(n^2)$, but linear running time can be observed for real problems. In the same paper some evidence is given that the linear expected running time for randomly generated problems can be mathematically proven. A rigorous proof, however, is still missing. We used an implementation of this algorithm in our experiments. CPU times are low, 4.5 seconds for computing

the Delaunay graph for a set of 20,000 points. We note that computing Voronoi diagrams and Delaunay graphs in a numerically stable way is a nontrivial task. In Jünger, Reinelt and Zepf (1991) and Kaibel (1993) it is shown how round-off errors during computation can be avoided to obtain reliable computer codes.

There are heuristics which try to directly exploit information from the Voronoi diagram (Ruján, Evertsz and Lyklema (1988), Cronin (1990), Segal, Zhang and Tsai (1991)).

The Delaunay graph can be exploited to speed up computations, as can be seen from what follows.

Minimum spanning trees

For Euclidean problem instances, one can compute minimum spanning trees very fast because computation can be restricted to the edge set of the Delaunay graph. Now we can use Kruskal's algorithm (Kruskal (1956)) which runs (if properly implemented using fast union-find techniques) in time $O(n \log m)$ where m is the number of edges of the graph. In the Euclidean case we thus obtain a running time of $O(n \log n)$. For example, it takes time 1.3 seconds to compute a minimum spanning tree for problem *pr2392* from the Delaunay graph. Using more sophisticated data structures the theoretical worst-case running time can be improved further (see Tarjan (1983) and Cormen, Leiserson and Rivest (1989)), but this does not seem to be of practical importance.

Implementing the nearest neighbor heuristic efficiently

Using the Delaunay graph, we can improve the running time of the standard nearest neighbor heuristic for Euclidean problem instances. Namely, if we want to determine the k nearest neighbors of some node, then it is sufficient to consider only nodes which are at most k edges away in the Delaunay graph. Using breadth-first search starting at a node, say i, we compute for $k = 1, 2, \ldots$ the k-th nearest neighbor of i until a neighbor is found that is not yet contained in the current partial Hamiltonian cycle. Due to the properties of the Delaunay graph we should find the nearest neighbor of the current node by examining only a few edges. Since in the last steps of the algorithm we have to collect the forgotten nodes (which are far away from the current node) it makes no sense to use the Delaunay graph any further. We found that it is faster, if the final nodes are just added using the standard nearest neighbor approach.

The worst case time complexity of this modified nearest neighbor search is still $O(n^2)$ but, in general, reduction of running time is considerable. For *rl5934* we reduced the running time to 0.7 seconds (adding the final 200 nodes by the standard nearest neighbor routine) compared to 40.4 seconds for the standard implementation. Plotting CPU times versus problem sizes exhibits that we can expect an almost linear growth of running time.

Computing candidate sets efficiently

We have observed the importance of limiting search for improving moves (for example by using candidate sets). In this subsection we address the question of

which candidate sets to use and of how to compute them efficiently for geometric problems. Three types of such sets were considered.

An obvious one is the *nearest neighbor candidate set*. Here, for every node the edges to its k nearest neighbors (where k is usually between 5 and 20) are determined. The candidate set consists of the collection of the corresponding edges. For example, optimal solutions for the problems *pcb442*, *rat783*, or *pr2392* are contained in their 8 nearest neighbor subgraphs. On the other hand, in problem *d198* the points form several clusters, so even the 20 nearest neighbor subgraph is still disconnected. Nevertheless, the edges to neighbors provide promising candidates to be examined. The idea of favoring near neighbors has already been used by Lin and Kernighan to speed up their algorithm. They chose $k = 5$ for their computational experiments.

A straightforward enumeration procedure computes the k nearest neighbors in time $O(n^2)$. As described above neighbor computations can be performed much faster if the Delaunay graph is available. For example, computation of the 10 nearest neighbor subgraph for a set of 20,000 points takes 8.3 seconds. In our practical experiments we observed a linear increase of the running time with the problem size.

Another candidate set is derived from the Delaunay graph itself, since it seems to give useful information about the geometric structure of a problem. It is known, however, that this graph does not have to contain a Hamiltonian cycle (Dillencourt (1987a, 1987b)). First experiments showed that it provides a candidate set too small for finding good Hamiltonian cycles. We therefore decided to use the *Delaunay candidate set*. This set is composed of the Delaunay graph as defined above and transitive edges of order 2, i.e., if node i is connected to node j, and node j is connected to node k in the Delaunay graph, then the edge from i to k is also taken into the candidate set. (This set may contain some very long edges that can be deleted in a heuristic way.)

Also this candidate subgraph can be computed very efficiently (e.g., in 14.5 seconds for 20,000 nodes). The average cardinality of the three subgraphs for our test of sample problems nodes is $2.75n$ for the Delaunay graph, $5.73n$ for the 10 nearest neighbor graph, and $9.82n$ for the Delaunay set.

Another efficient way of computing nearest neighbors is based on k-d-trees (see Bentley (1992) and Johnson (1990)). Using the Delaunay graph we can achieve running times competitive with this approach.

Experiments have shown that the nearest neighbor candidate set fails on clustered point configurations, whereas the Delaunay candidate set seems to have advantages for such configurations but contains too many edges. The *combined candidate set* attempts to combine the advantages of the two previous ones. For every node the edges to its k nearest neighbors (where k between about 5 and 20) are determined. The candidate set consists of the collection of these edges and those of the Delaunay graph.

We found that, in general and if applicable, the combined candidate set is preferable. It was therefore used in most of our practical computations. Of course, further candidate sets can be constructed. One possibility is to partition the plane

into regions according to some scheme and then give priority to edges connecting points in adjacent regions.

Note that the very fast computation of these candidate sets strongly relies on the geometric nature of the problem instances. In general, one has to find other ways for deriving suitable candidate sets.

We have outlined some ideas which are useful for the handling of very large geometric traveling salesman problems. Though applied here only to Euclidean problems, the methods or variants of them are also suitable for other types of geometric problems. Delaunay triangulations for the Manhattan or maximum metric have the same properties as for the Euclidean metric and can be computed as efficiently. For treating geometric problems on several hundred thousand nodes it is necessary to use methods of the type discussed above. Exploitation of geometric information is an active research field, and we anticipate further interesting contributions in this area.

4.4. A survey of other recent approaches

The heuristics discussed so far have a chance to find optimal solutions. But, even if we apply the best heuristic of the previous subsections, namely the Lin-Kernighan heuristic, we will usually encounter solutions of quality only about 1%. This is explained by the fact that, due to limited modification capabilities, every improvement heuristic will only find a local minimum. The weaker the moves that can be performed, the larger is the difference between a locally optimal solution and a true optimal solution. One way to overcome this drawback is to start improvement heuristics many times with different (randomly generated) starts because this increases the chance of finding better local minima. Success is limited, though.

Most of the heuristics we consider in this subsection try to escape from local minima or avoid local minima in a more systematic way. A basic ingredient is the use of randomness or *stochastic search* in contrast to the purely deterministic heuristics we have discussed so far. The first random approach is the so-called *Monte-Carlo algorithm* (Metropolis, Rosenbluth, Rosenbluth, Teller and Teller (1953)). In some cases the design of a particular method is influenced by the desire of imitating nature (which undoubtedly is able to find solutions to highly complex problems) in the framework of combinatorial optimization. We have not implemented the heuristics of this subsection, but give references to the literature.

Simulated Annealing

The approach of *simulated annealing* is based on a correspondence between the process of searching for an optimal solution in a combinatorial optimization problem and phenomena occurring in physics (Kirkpatrick, Gelatt and Vecchi (1983), Cerny (1985)).

To visualize this analogy consider the physical process of cooling a liquid to its freezing point with the goal of obtaining an ordered crystalline structure. Rapid cooling would not achieve this, one rather has to slowly cool (anneal) the liquid in order to allow improper structures to readjust and to have a perfect order (ground state) at the crystallization temperature. At each temperature step the system relaxes to its state of minimum energy.

Simulated annealing is based on the following analogy between such a physical process and an optimization method for a combinatorial minimization problem. Feasible solutions correspond to states of the system (an optimal solution corresponding to a ground state, i.e., a state of minimum energy). The objective function value resembles the energy in the physical system. Relaxation at a certain temperature is modeled by allowing random changes of the current feasible solution which are controlled by the level of the temperature. Depending on the temperature, alterations that increase the energy (objective function) are more or less likely to occur. At low temperatures it is very improbable that the energy of the system increases. System dynamics is imitated by local modifications of the current feasible solution.

Modifications that increase the length of a Hamiltonian cycle are possible, but only accepted with a certain probability. Pure improvement heuristics as we have discussed so far can be interpreted in this context as rapid quenching procedures that do not allow the system to relax.

The general outline of a simulated annealing procedure for the TSP is the following.

procedure SIMULATED_ANNEALING

(1) Compute an initial Hamiltonian cycle T and choose an initial temperature ϑ and a repetition factor r.
(2) As long as the stopping criterion is not satisfied perform the following steps.
  (2.1) Do the following r times.
    (2.1.1) Perform a random modification of the current cycle to obtain the cycle T' and let $\Delta = c(T') - c(T)$ (difference of lengths).
    (2.1.2) If $\Delta < 0$ then set $T = T'$. Otherwise compute a random number x, $0 \leqslant x \leqslant 1$ and set $T = T'$ if $x < e^{-\frac{\Delta}{\vartheta}}$
  (2.2) Update ϑ and r.
(3) Output the best solution found.

Simulated annealing follows the general principle, that improving moves are always accepted, whereas moves increasing the length of the current cycle are only accepted with a certain probability depending on the increase and current value of ϑ.

The formulation has several degrees of freedom and various realizations are possible. Usually 2-opt or 3-opt moves are employed as basic modification in Step (2.1.1). The temperature ϑ is decremented in Step (2.2) by setting $\vartheta = \gamma\vartheta$ where γ is a real number close to 1, and the repetition factor r is usually initialized with the number of cities and updated by $r = \alpha r$ where α is some factor

between 1 and 2. Realization of Step (2.2) determines the so-called *annealing schedule* or *cooling scheme* (much more complicated schedules are possible). The scheme given above is named geometric cooling. The procedure is stopped if the length of the current Hamiltonian cycle was not altered during several temperature steps. Expositions of general issues for the development of simulated annealing procedures can be found in Aarts and Korst (1989) and Johnson, Aragon, McGeoch and Scheron (1991), a bibliography is given in Collins, Eglese and Golden (1988).

Computational experiments for the TSP are for example reported in Kirkpatrick (1984), van Laarhoven (1988), Johnson (1990). It is generally observed that simulated annealing can find very good or even optimal solutions and beats Lin-Kernighan concerning quality. To be certain of this, however, one has to spend considerable CPU time because temperature has to be decreased very slowly and many repetitions at each temperature step are necessary.

We think that the most appealing property of simulated annealing is its fairly simple implementation. The principle can be used to approach very complicated problems if only a basic subroutine is available that turns a feasible solution into another feasible solution by some modification. Hajek (1985) proved convergence of the algorithm to an optimal solution with probability 1, if the basic move satisfies a certain property. Unfortunately, the theoretically required annealing scheme is not suited for practical use. The proper choice of an annealing scheme should not be underestimated. It is highly problem dependent and only numerous experiments can find the most suitable parameters.

A variant of simulated annealing enhanced by deterministic local improvement (3-opt) leads to so-called *large-step Markov chain methods* (see Martin, Otto and Felten (1992)). When such methods are properly implemented, near optimal solutions can be found faster than with pure simulated annealing. A related heuristic motivated by phenomena from physics is *simulated tunneling* described in Ruján (1988).

A simplification of simulated annealing, called *threshold accept*, is proposed in Dueck and Scheuer (1988). This heuristic removes the probability involved for the acceptance of a bad move in the original method. Rather, in each major iteration (Step (2.1)) an upper bound is given by which the length of the current cycle is allowed to be increased by the basic move. This threshold value is decreased according to some rule. The procedure is stopped if changes of the solution are not registered for several steps. Computational results are shown to display the same behavior as simulated annealing. A theoretical convergence result can also be obtained (Althöfer and Koschnick (1989)).

An even simpler variant is discussed in Dueck (1990) under the name of *great-deluge* heuristic. Here for each major iteration there is an upper limit on the length of Hamiltonian cycles that are accepted. Every random move yielding a cycle better than this length is accepted (note the difference from the threshold accept approach). The name of this approach comes from the interpretation that (for a maximization problem) the limit corresponds to a rising level of water and moves leading "into the water" are not accepted. This method is reported to

yield good results with fairly moderate computation times for practical traveling salesman problems arising from drilling printed-circuit boards.

Evolutionary strategies and genetic algorithms

The development of these two related approaches was motivated by the fact that many very good (or presumably optimal) solutions to highly complex problems can be found in nature itself.

The first approach is termed *evolutionary strategy* since it is based on analogues of "mutation" and "selection" to derive an optimization heuristic (Rechenberg (1973)). Its basic principle is the following.

procedure EVOLUTION

(1) Compute an initial Hamiltonian cycle T.
(2) As long as the stopping criterion is not satisfied perform the following steps.
  (2.1) Generate a modification of T to obtain the cycle T'.
  (2.2) If $c(T') - c(T) < 0$ then set $T = T'$.
(3) Output the best solution found.

In contrast to previous methods of this subsection, moves increasing the length of the Hamiltonian cycle are not accepted. The term "evolution" is used because the moves generated in Step (2.1) are biased by knowledge acquired so far, i.e., somehow moves that lead to a decrease of cycle length should influence the generation of the next move. This principle, however, is hardly followed in practice, moves taken into account are usually k-opt moves generated at random. Formulated this way the procedure cannot leave local minima and experiments show that it indeed gets stuck in poor local minima. Moreover, convergence is slow, justifying the name "creeping random search" which is also used for this method. To leave local minima one has to incorporate the possibility of perturbations that increase the cycle length (Ablay (1987)). Then this method resembles a mixture of pure evolutionary strategy, simulated annealing, threshold accept, and tabu search (see below).

More powerful in nature than mutation-selection is genetic recombination. Interpreted in terms of the TSP this means that new solutions should not be constructed from just one parent solution but rather be a suitable combination of two or more. Heuristics following this principle are termed *genetic algorithms*.

procedure GENETIC_ALGORITHM

(1) Compute an initial set $\mathcal{T}$ of Hamiltonian cycles.
(2) As long as the stopping criterion is not satisfied perform the following steps.
  (2.1) Recombine two or more cycles of $\mathcal{T}$ to obtain a new cycle T which is added to $\mathcal{T}$.
  (2.2) Reduce the set $\mathcal{T}$ according to some rule.
(3) Output the best solution found during the heuristic.

We see that Step (2.1) mimics reproduction in the population $\mathcal{T}$ and that Step (2.2) corresponds to a "survival of the fittest" rule.

There are numerous possible realizations. Usually, subpaths of given cycles are connected to form new cycles and reduction is just keeping the set of k best solutions of $\mathcal{T}$. One can also apply deterministic improvement methods to the newly generated cycle T before performing Step (2.2). Findings of optimal solutions are reported for some problem instances (the largest one being problem *att532*) with an enormous amount of CPU time. For further reading we refer to Mühlenbein, Gorges-Schleuter and Krämer (1988), Goldberg (1989), and Ulder, Pesch, van Laarhoven, Bandelt and Aarts (1990).

Tabu Search

Some of the above heuristics allow length-increasing moves, so local minima can be left during computation. No precaution, however, is taken to prevent the heuristic to revisit a local minimum several times. This absence was the starting point for the development of *tabu search* where a built-in mechanism is used to forbid (tabu) returning to the same feasible solution. In principle the heuristic works as follows.

procedure TABU_SEARCH

(1) Compute an initial Hamiltonian cycle T and start with an empty tabu list $\mathcal{L}$.
(2) As long as the stopping criterion is not satisfied perform the following steps.
  (2.1) Perform the best move that is not forbidden by $\mathcal{L}$.
  (2.2) Update the tabu list $\mathcal{L}$.
(3) Output the best solution found.

Again, there are various possibilities to realize a heuristic based on the tabu search principle. Basic difficulties are the design of a reasonable tabu list, the efficient management of this list, and the selection of the most appropriate move in Step (2.1). A thorough discussion of these issues can be found in Glover (1990). Computational results for the TSP are reported in Knox and Glover (1989), Malek, Guruswamy, Owens and Pandya (1989), and Malek, Heap, Kapur and Mourad (1989).

Neural Networks

This approach tries to mimic the mode of operation of the human brain. Basically one models a set of neurons connected by a certain type of interconnection network. Based on the inputs that a neuron receives, a certain output is computed which is propagated to other neurons. A variety of models addresses activation status of neurons, determination of outputs and propagation of signals in the net with the basic goal of realizing some kind of learning mechanism. The result computed by a neural network either appears explicitly as output or is given by the state of the neurons.

In the case of the TSP there is, for example, the "elastic band" approach (Durban and Willshaw (1987)) for Euclidean problem instances. Here a position in the plane is associated with each neuron. In the beginning, the neurons are ordered along a circle. During the computation, neurons are "stimulated" and

approach a cycle through the given set of points. Applications for the TSP can also be found in Fritzke and Wilke (1991). For further reading on *neural networks* or *connectionism* see Hopfield and Tank (1985), Kemke (1988) and Rumelhart, Hinton and McClelland (1986). Computational results are not yet convincing.

Summarizing, we would classify all heuristics presented in this subsection as *randomized improvement heuristics*. Also the iterated Lin-Kernighan heuristic falls into this class, since it performs a random move after each iteration. The analogues drawn from physics or biology are entertaining, but we think that they are a bit overstressed. The central feature is the systematic use of randomness which may avoid local minima and therefore yield a chance of finding optimal solutions (if CPU time is available). It is interesting from a theoretical point of view that convergence to an optimal solution with probability 1 can be shown for some variants, but the practical impact of these results is limited. The approaches have the great advantage, however, that they are generally applicable to combinatorial optimization problems and other types of problems. They can be implemented routinely with little knowledge about problem structure. If enough CPU and real time is available they can be applied (after spending some time for parameter tuning) to large problems with a good chance of finding solutions close to the optimum.

For many practical applications the heuristics presented in this subsection may be sufficient for treating the problems satisfactorily. But, if one is (or has to be) more ambitious and searches for proven optimal solutions or solutions meeting a quality guarantee, one has to go beyond these methods. The remainder of this chapter is concerned with solving TSP instances to optimality or computing near optimal solutions with quality guarantees.

5. Relaxations

A *relaxation* of an optimization problem P is another optimization problem R, whose set of feasible solutions $\mathcal{R}$ properly contains all feasible solutions $\mathcal{P}$ of P. The objective function of R is an arbitrary extension on $\mathcal{R}$ of the objective function of P. Consequently, the objective function value of an optimal solution to R is less than or equal to the objective function value of an optimal solution to P. If P is a hard combinatorial problem and R can be solved efficiently, the optimal value of R can be used as a *lower bound* in an enumeration scheme to solve P. The closer the optimal value of R to the optimal value of P, the more efficient is the enumeration algorithm.

Since the TSP is an $\mathcal{NP}$-hard combinatorial optimization problem, the standard technique to solve it to optimality is based on an enumeration scheme, and so the study of effective relaxations is fundamental in the process of devising good exact algorithms. We consider here discrete and continuous relaxations, i.e., relaxations with discrete and continuous feasible sets.

Before we describe these relaxations we give some notation and recall some basic concepts.

For any edge set $F \subseteq E_n$ and any $x \in \mathbb{R}^{E_n}$, $x(F)$ denotes the sum $\sum_{e \in F} x_e$. For a node set $W \subset V_n$, $E_n(W) \subset E_n$ denotes $\{uv \in E_n \mid u, v \in W\}$ and $\delta_n(W) \subset E_n$ denotes $\{uv \in E_n \mid u \in W, v \in V_n \setminus W\}$. We call $\delta_n(W)$ a *cut* with *shores* W and $V_n \setminus W$.

The solution set of the TSP is the set $\mathcal{H}_n$ of all Hamiltonian cycles of K_n. A Hamiltonian cycle, as defined in Section 1, is a subgraph $H = (V_n, E)$ of K_n satisfying the following requirements:

$$\text{(a) all nodes of } H \text{ have degree 2;} \tag{5.1}$$

$$\text{(b) } H \text{ is connected.} \tag{5.2}$$

The edge set of a subgraph of K_n whose nodes have all degree 2 is a *perfect 2-matching*, i.e., a collection of simple disjoint cycles of at least three nodes and with no chords such that each node of K_n belongs to some of these cycles. Consequently, a Hamiltonian cycle can be defined as a connected perfect 2-matching.

It is easy to see that if a perfect 2-matching is connected, then it is also biconnected, i.e., it is necessary to remove at least two edges to disconnect it. Therefore, the requirements (5.1) and (5.2) can be replaced by

$$\text{(a) all nodes of } H \text{ have degree 2;} \tag{5.3}$$

$$\text{(b) } H \text{ is biconnected.} \tag{5.4}$$

With every $H \in \mathcal{H}_n$ we associate a unique *incidence vector* $\chi^H \in \mathbb{R}^{E_n}$ by setting

$$\chi_e^H = \begin{cases} 1 & \text{if } e \in H \\ 0 & \text{otherwise.} \end{cases}$$

The incidence vector of every Hamiltonian cycle satisfies the system of equations

$$A_n x = \mathbf{2}, \tag{5.5}$$

where A_n is the node-edge incidence matrix of K_n and $\mathbf{2}$ is an n-vector having all components equal to 2. The equations $A_n x = \mathbf{2}$ are called the *degree equations* and translate the requirement (5.3) into algebraic terms. In addition, for any nonempty $S \subset V_n$ and for any Hamiltonian cycle H of K_n, the number of edges of H with an endpoint in S and the other in $V_n - S$ is at least 2 (and even). Therefore, the intersection of the edge set of H with the cut $\delta_n(S)$ has cardinality at least 2 (and even), and so χ^H must satisfy the following set of inequalities:

$$x(\delta_n(S)) \geqslant 2 \qquad \text{for all } \emptyset \neq S \subset V_n. \tag{5.6}$$

These inequalities are called *subtour elimination inequalities* because they are not satisfied by the incidence vector of nonconnected 2-matchings (i.e., the union of two or more subtours), and so they translate the requirement (5.4) into algebraic terms.

Given an objective function $c \in \mathbb{R}^{E_n}$ that associates a "length" c_e with every edge e of K_n, the TSP can be solved by finding a solution to the following integer linear program:

Problem 5.1

$$\begin{aligned} \min \quad & cx \\ \text{s.t.} \quad & A_n x = \mathbf{2}, && (5.7) \\ & x(\delta_n(S)) \geqslant 2 \qquad \text{for all } \emptyset \neq S \subset V_n, && (5.8) \\ & \mathbf{0} \leqslant x \leqslant \mathbf{1}, && (5.9) \\ & x \text{ integer}. && (5.10) \end{aligned}$$

This is the most commonly used integer linear programming formulation of the TSP.

5.1. Subtour relaxation

An LP relaxation of the TSP is the linear program obtained by relaxing the integrality condition (5.10) of any integer programming formulation of the problem. Consequently, an LP relaxation has a polyhedron in $\mathbb{R}^{E_n}$ as a feasible set and thus it is a *continuous relaxation*.

A simple LP relaxation is the one defined by the constraints (5.7)–(5.9). The polyhedron defined by these constraints is called the *subtour elimination polytope* and the corresponding relaxation is called the *subtour relaxation*.

The number of constraints defined in (5.7)–(5.9) is $n + 2^n - 2 + 2m$. Some of them are redundant, though, and it is not difficult to show that the system (5.7)–(5.9) can be reduced to one not containing redundant constraints and having n equations and $2^{n-1} - n - 1 + m$ inequalities, still a number of constraints given by an exponential function of n. Such a huge linear program obviously cannot be solved by a direct application of the simplex method or of any other efficient algorithm for linear programming. However, as we will describe in Subsection 5.5, an optimal solution of a linear objective function over the subtour elimination polytope can be found in polynomial time. Moreover, the optimization over this polytope can be carried out very efficiently in practical computation by a simple iterative procedure. We start, for example, by solving the linear program defined by the constraints (5.7) and (5.9) and by a small subset (possibly empty) of the constraints (5.8). Then we check if the optimal solution satisfies all the constraints (5.8). If so, we terminate since we have found the optimal solution over the subtour elimination polytope. Otherwise, we add some of the violated constraints to the linear program and we solve it again. The number of iterations

that are necessary before obtaining the optimum over the subtour polytope is, in practice, a small fraction of n, and the constraints of the linear programs that are solved at each iteration form a small subset of (5.7)–(5.9).

We discuss this procedure in more detail in Subsection 5.5. However, the reader who is not familiar with polyhedral combinatorics should keep it in mind while reading Subsection 5.4, where other continuous relaxations, also involving constraints sets of exponential size, are considered.

It would be interesting to know how close the bound c_L obtained by solving the subtour relaxation is to the length of an optimal Hamiltonian cycle c_{opt}. Wolsey (1980) and Shmoys and Williamson (1990) show that for any nonnegative cost function c the ratio c_L/c_{opt} is at least 2/3. The 2/3 bound is not shown to be tight and actually it is conjectured in Goemans (1993) that $c_L/c_{\text{opt}} \geqslant 3/4$. To prove this conjecture a deeper knowledge of the structure of the subtour elimination polytope would probably be necessary. Some results in this direction are given by Boyd and Pulleyblank (1990) who characterize some of its vertices. But it is still an open problem how to completely characterize all of them.

Computational experiments show that for many instances the above ratio is very close to 1 (see the fourth column of Table 5). There are actually classes of instances for which the optimization over the subtour elimination polytope always yields the incidence vector of an optimal Hamiltonian cycle (thus the ratio is 1 for these instances). Padberg and Sung (1988) show that some instances of the TSP that are very hard for k-opt heuristic algorithms, fall into this category.

The excellent quality of the lower bound obtained from the subtour relaxation is most probably the main reason of the successful computation of the optimal solutions of several large TSP instances reported in the literature (see Section 7).

5.2. 1-tree relaxation

A discrete relaxation of the TSP can be obtained by defining a combinatorial problem whose feasible solutions are subgraphs of K_n satisfying a subset of the requirements (5.1) and (5.2) and whose optimal solutions can be found in polynomial time.

The requirements (5.1) and (5.2) imply that any subgraph of K_n that satisfies them has exactly n edges and spans K_n, i.e., has an edge incident with any node of K_n. Therefore, we only consider relaxations having as feasible solutions spanning subgraph of K_n with n edges.

As a first relaxation we can, for example, drop the requirement (5.1) and consider as feasible solutions the connected spanning subgraphs of K_n with n edges. These graphs consist of a spanning tree plus an extra edge. With respect to the edge weights given by c, a minimal weight graph of this kind can be found by taking a minimal spanning tree and adding an edge not belonging to the tree with minimal weight. The minimal spanning tree can be found in polynomial time (see Subsection 4.3).

A better relaxation can be obtained by dropping the requirement (5.1) for all nodes except one, say node 1. Denote by $K_n \setminus \{1\}$ the subgraph of K_n induced by

$V_n \setminus \{1\}$. The subgraph of a Hamiltonian cycle induced by the nodes in $V_n \setminus \{1\}$ is a Hamiltonian path of $K_n \setminus \{1\}$. This is a subgraph H_p of $K_n \setminus \{1\}$ satisfying the following requirements:

(a) H_p spans $K_n \setminus \{1\}$;
(b) H_p has $n-2$ edges;
(c) H_p is connected;
(d) all nodes of H_p but two have degree 2. (5.11)

The union of a Hamiltonian path on $K_n \setminus \{1\}$ and of two edges incident with node 1 is definitely a relaxation of a Hamiltonian cycle, but not a useful one, since finding a minimum cost Hamiltonian path in $K_n \setminus \{1\}$ is as difficult as finding a minimum cost Hamiltonian cycle in K_n. However, instead of a Hamiltonian path we can consider its relaxation obtained by dropping the requirement (5.11). It can be shown that the resulting graph is a tree that spans $K_n \setminus \{1\}$.

Now we can produce a useful relaxation of the TSP by taking the union of a pair of edges incident with node 1 and a tree spanning $K_n \setminus \{1\}$. Such a graph is called a *1-tree*. The 1-tree that minimizes the objective function c can be easily obtained by finding the minimum spanning tree in $K_n \setminus \{1\}$ and adding to it the shortest two edges incident with node 1. Thus the complexity of finding a minimum 1-tree amounts to that of finding a minimum spanning tree.

Unfortunately, this relaxation is not very strong. To strengthen the lower bound obtained by finding a minimum cost 1-tree, we modify the objective function in the following way. If for all $u \in V_n$ we add a constant λ_u to the objective function coefficients of all edges of $\delta(u)$, the length of all Hamiltonian cycles of K_n increases by the same amount $\sum_{u \in V_n} 2\lambda_u$ (since any Hamiltonian cycle has exactly two edges in each edge set $\delta(u)$). For any edge (u, v) of K_n, the corresponding coefficient of the new objective function is $c'(u, v) = c(u, v) + \lambda_u + \lambda_v$ (since (u, v) belongs to both $\delta(u)$ and $\delta(v)$). Consequently, the optimal value of the problem

$$\min_{H \in \mathcal{H}_n} \left\{ \sum_{i<j} (c_{ij} + \lambda_i + \lambda_j) \chi^H_{ij} \right\} - 2 \sum_{i \in V} \lambda_i$$

does not depend on the vector $\lambda \in \mathbb{R}^{V_n}$.

The same does not hold for 1-trees, though, because in general not all nodes of a 1-tree satisfy (5.1). Therefore, the length of an optimal 1-tree

$$L(\lambda) = \min_{\chi^{1T}} \left\{ \sum_{i<j} (c_{ij} + \lambda_i + \lambda_j) \chi^{1T}_{ij} \right\} - 2 \sum_{i \in V} \lambda_i,$$

where χ^{1T} is the incidence vector of a 1-tree in K_n, is a (nonconstant) function of $\lambda \in \mathbb{R}^{V_n}$. $L(\lambda)$ is a lower bound for the c-length of a Hamiltonian cycle and in

general different vectors λ^1 and λ^2 yield different lower bounds $L(\lambda^1)$ and $L(\lambda^2)$. Thus the tightest bound is obtained by solving the maximization problem

$$\max_{\lambda}\{L(\lambda)\}, \tag{5.12}$$

called the Lagrangean dual problem. The bound obtained by solving (5.12) is proposed in Held and Karp (1970, 1971) and it is known as the Held-Karp bound.

It is interesting to note that the lower bound produced by the subtour relaxation is equivalent to the Held-Karp bound (see, e.g., Nemhauser and Wolsey (1988), pp. 469–475). Although these two bounds are identical they are quite different from the methodological viewpoint. Problem (5.12) is a piecewise linear nondifferentiable concave optimization problem and can be solved by an iterative procedure (see Section 6) whose convergence can be very slow. In practical computation the procedure is stopped prematurely, thus providing a lower bound that is worse than the theoretical one given in (5.12). On the contrary, the bound obtained from optimizing over the subtour elimination polytope can be computed by solving a (usually short) sequence of linear programs, the last of which provides the exact theoretical bound (5.12). For these reasons the latter method is preferable. However, the computation based on the subtour relaxation requires a linear program optimizer and a quite complex algorithm, while the computation of an approximation of the Held-Karp bound can be carried out with a very simple procedure that requires moderate amounts of computer memory (see Section 6). Therefore, when the instance is too large to be attacked by the available linear program optimizers or when one is willing to afford only moderate implementation costs, the Held-Karp approach seems more suitable.

Any LP relaxation tighter than the subtour relaxation produces a lower bound superior to the Held-Karp bound. This explains why the algorithms based on the LP relaxations perform much better than those based on the 1-tree relaxation. For this reason we will spend most of this section describing the LP relaxations in more detail.

5.3. 2-matching relaxation

A 2-matching relaxation for the TSP is another discrete relaxation obtained by relaxing the requirement (5.2), i.e., by solving Problem 5.1 without the constraints (5.8). The resulting relaxation is a minimum cost 2-matching that can be found in polynomial time (see Edmonds and Johnson (1970), Cunningham and Marsh (1978), and Padberg and Rao (1982)). However, implementing an algorithm for the minimum cost 2-matching problem efficiently is not as simple as for the minimum cost spanning tree problem. The lower bound obtained by finding the minimum cost 2-matching is in general poor (see, e.g., Table 5) and, like for the 1-tree, it can be improved using techniques in the same spirit as for the Held-Karp bound. The improved bound is better than the one obtained by solving the problem (5.12) (see, e.g., Nemhauser and Wolsey (1988), pp. 469–475).

5.4. Strong LP relaxations

An LP relaxation of the TSP is not unique, since every linear program obtained from Problem 5.1 by adding to (5.7)–(5.9) any number of linear constraints which are *valid*, i.e., satisfied by the incidence vectors of all Hamiltonian cycles of K_n, is also an LP relaxation.

To produce a relaxation that is stronger than the subtour relaxation it is necessary to add inequalities to (5.7)–(5.9). They must satisfy two requirements. In order to produce a mathematically correct relaxation they must be valid. To produce a tight relaxation they must be "strong", in the sense that they must define a polytope substantially smaller than the subtour polytope. For two valid inequalities $a_1x \geqslant b_1$ and $a_2x \geqslant b_2$ we say that the first is *stronger* than the second (or that the first *dominates* the second) if the polytope defined by $a_1x \geqslant b_1$ and by (5.7)–(5.9) is properly contained in the polytope defined by $a_2x \geqslant b_2$ and (5.7)–(5.9). To produce good relaxations it is quite natural to look for valid inequalities that are not dominated by any other. The derivation of these inequalities is intimately related to the study of the structure of a polytope associated with the TSP, called the symmetric traveling salesman polytope.

The *symmetric traveling salesman polytope* (STSP(n)) is the convex hull of the set of the incidence vectors of all Hamiltonian cycles of K_n, i.e.,

$$\mathrm{STSP}(n) = \mathrm{conv}\left\{\chi^H \mid H \in \mathcal{H}_n\right\}.$$

It is known that there exists a finite minimal set $\mathcal{B}^=$ of linear equations and a finite minimal set $\mathcal{B}^\leqslant$ of linear inequalities whose set of solutions is precisely STSP(n). The sets $\mathcal{B}^=$ and $\mathcal{B}^\leqslant$ are minimal in the sense that the removal of any of their elements results in a polytope that properly contains STSP(n). The equations of $\mathcal{B}^=$ are precisely the degree equations (5.5). Each of the inequalities in $\mathcal{B}^\leqslant$ defines a *facet* of STSP(n) (for the definition of facet and for other basic concepts of polyhedral theory we refer to Grötschel and Padberg (1985), Nemhauser and Wolsey (1988), and Pulleyblank (1983)). The equations of $\mathcal{B}^=$ and the inequalities of $\mathcal{B}^\leqslant$ are the constraints of the best LP relaxation of the TSP. In fact there is always an optimal extreme solution to this relaxation that is the incidence vector of an optimal Hamiltonian cycle, for any objective function c. Unfortunately, the set $\mathcal{B}^\leqslant$ contains an enormous number of inequalities and its size grows exponentially with n. Presently, a complete description of a minimal system of inequalities defining STSP(n) is known only for $n \leqslant 8$. For $n = 6$ the set $\mathcal{B}^\leqslant$ contains 100 inequalities and is described by Norman (1955). For $n = 7$ the complete description of $\mathcal{B}^\leqslant$, which contains 3,437 inequalities, is given by Boyd and Cunningham (1991). The set $\mathcal{B}^\leqslant$ for $n = 8$ (containing 194187 inequalities) is described by Christof, Jünger and Reinelt (1991). It is very unlikely that a complete description of this system of inequalities can be found for all n. Nevertheless, good LP relaxations can be obtained using only subsets of $\mathcal{B}^\leqslant$. For this reason and because the description of STSP(n) with linear inequalities is a challenging mathematical task by itself, many researchers have been interested in studying this polytope. The work in this area has been

focused mostly on characterizing large families of valid inequalities for STSP(n) and, in many cases, in showing that some of these families are subsets of $\mathcal{B}^{\leqslant}$.

The first systematic study of the polyhedral structure of STSP(n) was done by Grötschel and Padberg. The results of their work are published in the doctoral dissertation of Grötschel (1977) and in the papers Grötschel and Padberg (1974, 1977, 1978, 1979a, 1979b). Their main discovery was a large family of facet-defining inequalities, the *comb inequalities*, that are the major contributors to the LP relaxations used by the most successful algorithms for the solution of the TSP to optimality (see sections 5.6 and 6). Another important piece of the current knowledge of the TSP polytope is provided by Cornuéjols, Fonlupt and Naddef (1985), who describe many new classes of inequalities that are facet-defining for GTSP, the polyhedron associated with the graphical traveling salesman problem. In fact, many results on GTSP can be extended to STSP, due to a strong relationship between the two polyhedra that is described and exploited in Naddef and Rinaldi (1993).

Many other important results on STSP and on related issues appeared in the literature. We cite only those that provide new members of the set $\mathcal{B}^{\leqslant}$ of facet-defining inequalities for this polytope. These results are reported in the next subsection. They may appear too technical to those who are mainly interested in the algorithmic issues. However, it may be worth to observe that the possibility to produce faster and more robust algorithms for finding optimal or provably good solutions of the TSP seems to depend right on the exploitation of these results.

5.5. Known facets of STSP(n)

The incidence vectors of all Hamiltonian cycles of K_n satisfy the degree equations (5.7), and so STSP(n) is not a full dimensional polytope (its dimension is $m - n$), i.e., it is contained in the intersection of the n hyperplanes defined by the degree equations. A consequence of this fact is that, unlike in the case of a full dimensional polyhedron, a facet-defining inequality for STSP(n) is not uniquely defined (up to a multiplication by a scalar). If $hx \geqslant h_0$ defines a facet of STSP(n), then the inequality $fx \geqslant f_0$, with $f = \lambda A_n + \pi h$, $f_0 = \lambda \mathbf{2} + \pi h_0$, $\pi > 0$, and $\lambda \in \mathbb{R}^{V_n}$, defines the same facet. The two inequalities are said to be *equivalent*.

As a consequence of this lack of uniqueness the facet-defining inequalities of STSP(n) are described in the literature in different forms. We describe the two most frequently used forms: the closed form and the tight triangular form. The direction of the inequalities is "$\leqslant$" for the first form and "$\geqslant$" for the second.

Let $\mathcal{S} = \{S_1, S_2, \ldots, S_t\}$ be a collection of subsets of V_n and let $r(\cdot)$ denote a suitable function of $\mathcal{S}$, which only depends on the number of its members but not on their size. An inequality is written in *closed form* if it is as follows:

$$\sum_{S \in \mathcal{S}} \alpha_S x(E_n(S)) \leqslant \sum_{S \in \mathcal{S}} \alpha_S |S| - r(\mathcal{S}), \tag{5.13}$$

where α_S is an integer associated with $S \in \mathcal{S}$. The closed form is nice for describing an inequality, but has two major drawbacks. Not all facet-defining inequalities of STSP(n) can be written in closed form. In addition, the closed form is not unique: for example, it can be shown that replacing any set S in (5.13) by its complement $V_n \setminus S$ produces a different inequality which is equivalent to (5.13).

A more interesting form for the facet-defining inequalities of STSP(n) is the tight triangular form. An inequality $fx \geqslant f_0$ defined on $\mathbb{R}^{E_n}$ is said to be in *tight triangular form* (or in *TT form*) if the following conditions are satisfied:

(a) the coefficients of f satisfy the triangle inequality, i.e., $f(u,v) \leqslant f(u,w) + f(w,v)$ for every triple u, v, w of distinct nodes in V_n;

(b) for all $u \in V_n$ there exists a pair of distinct nodes v, $w \in V_n \setminus \{u\}$ such that $f(v,w) = f(u,v) + f(u,w)$.

Let $hx \geqslant h_0$ be any inequality defined on $\mathbb{R}^{E_n}$. An inequality $fx \geqslant f_0$ in TT form that is equivalent to $hx \geqslant h_0$, with $f = \lambda A_n + \pi h$ and $f_0 = \lambda \mathbf{2} + \pi h_0$, can be obtained by setting π to any positive value and

$$\lambda_u = \frac{\pi}{2} \max \left\{ h(v,w) - h(u,v) - h(u,w) \;\middle|\; v,w \in V_n \setminus \{u\},\ v \neq w \right\}$$

for all $u \in V_n$.

The properties of the TT form of the inequalities can be used to explain the tight relationship between STSP(n) and GTSP(n). In particular, being in TT form is a necessary and sufficient condition for a facet-defining inequality of STSP(n) to be facet-defining for GTSP(n). For the details see Naddef and Rinaldi (1993).

Although two equivalent inequalities define the same facet, using a form rather than another may not be irrelevant in computation. The inequalities that we consider are used as a constraint of some linear program and all current LP optimizers are very sensitive to the density (percentage of nonzero coefficients) of the constraints. The lower the density the faster is the LP optimizer. The inequalities in TT form are in general denser than those in closed form. However, when only a subset of the variables is explicitly represented in a linear program, which is often the case when solving large TSP instances (see Section 6), the inequalities in closed form tend to be denser.

We now describe the basic inequalities that define facets of STSP(n).

Trivial inequalities

The inequalities $x_e \geqslant 0$ for $e \in E_n$ are called the *trivial inequalities* (this is the only form used for these inequalities). A proof that they are facet-defining for STSP(n) (with $n \geqslant 5$) is given in Grötschel and Padberg (1979b).

Subtour elimination inequalities

The subtour elimination inequalities (5.6) define facets of STSP(n). In (5.6) they are written in TT form. The corresponding closed form, obtained by setting $\mathcal{S} = \{S\}$, $\alpha_S = 1$ and $r(\mathcal{S}) = 1$, is

$$x(E_n(S)) \leqslant |S| - 1.$$

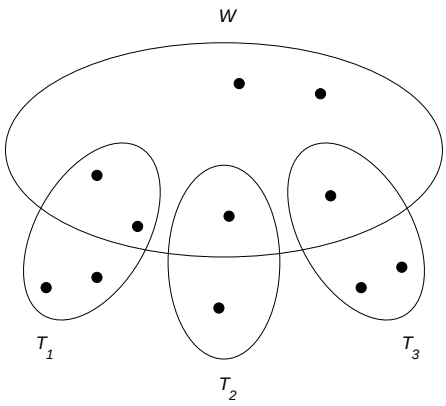

Figure 8. Handle and teeth of a comb.

These inequalities are introduced by Dantzig, Fulkerson and Johnson (1954) who do not address the issue of whether these inequalities are facet-defining for the TSP polytope. A proof that if $2 \leqslant |S| \leqslant n-2$, they are facet-defining for STSP(n) (with $n \geqslant 4$) is given in Grötschel and Padberg (1979b).

Comb inequalities

A *comb inequality* is defined by setting $\mathcal{S} = \{W, T_1, \ldots, T_k\}$, $\alpha_S = 1$ for all $S \in \mathcal{S}$, and $r(\mathcal{S}) = (3k+1)/2$. The set W is called the *handle* and the sets T_i are called the *teeth* of the comb. The inequality is facet-defining if the handle and the teeth satisfy the following conditions:

(i) $|T_i \cap W| \geqslant 1$ for $i = 1, \ldots, k$,
(ii) $|T_i \backslash W| \geqslant 1$ for $i = 1, \ldots, k$,
(iii) $T_i \cap T_j = \emptyset$ for $1 \leqslant i < j \leqslant k$,
(iv) $k \geqslant 3$ and odd.

Special cases of comb inequalities have different names in the literature. If (i) is satisfied *with equality*, the inequality is called a *Chvátal comb*. If all teeth have cardinality 2 the comb inequality is also called *2-matching* inequality. Figure 8 shows an example of a comb inequality.

The 2-matching inequalities were discovered by Edmonds (1965), who used them to provide a complete description of the polytope associated with the 2-matching problem. Chvátal (1973) defined a class of valid inequalities as a generalization of the 2-matching inequalities and called them *comb inequalities*. Now we refer to the inequalities in this class as *Chvátal combs*. Grötschel and Padberg (1979a, 1979b) generalized Chvátal's combs to a larger class that they called the *comb* inequalities and showed that the inequalities of this class are facet-defining for STSP(n) (with $n \geqslant 6$).

Clique-tree inequalities

A *clique-tree inequality* is defined by setting $\mathcal{S} = \{W_1, \ldots, W_r, T_1, \ldots, T_k\}$ (the sets W_i are called the *handles* and the sets T_i are called the *teeth* of the clique-

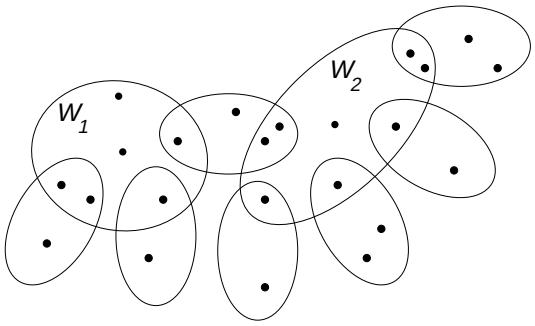

Figure 9. Handles and teeth of a clique-tree.

tree), $\alpha_S = 1$ for all $S \in \mathcal{S}$, and $r(\mathcal{S}) = (2\sum_{i=1}^{k} h(T_i) + k + 1)/2$, where $h(T)$ is the number of handles that have nonempty intersection with T. A *clique-tree* is a subgraph of K_n whose cliques are the handles and the teeth. The inequality is facet-defining if the clique-tree is connected and the following conditions are satisfied:

(i) no two teeth intersect;
(ii) no two handles intersect;
(iii) each tooth contains at least two and at most $n-2$ nodes;
(iv) each tooth contains at least one node not belonging to any handle;
(v) each handle intersects an odd number ($\geqslant 3$) of teeth;
(vi) if a tooth T and a handle W have a nonempty intersection, then $W \cap T$ is an articulation set of the clique-tree, i.e., the removal of the nodes in $W \cap T$ from K_n disconnects the clique-tree.

The clique-tree inequalities are a proper generalization of the comb inequalities. A clique-tree inequality is displayed in Figure 9. These inequalities are introduced and proved to define facets of STSP(n) (with $n \geqslant 11$) by Grötschel and Pulleyblank (1986).

PWB inequalities

By *simple PWB inequalities* we denote three classes of inequalities, namely the path, the wheelbarrow, and the bicycle inequalities.

The simple path inequalities are defined by graphs that are called path configurations. For any odd $k \geqslant 3$ and any k-tuple of positive integers $(n_1, \ldots, n_k)$, with $n_i \geqslant 2$ for $i \in \{1, \ldots, k\}$, let $P(n_1, \ldots, n_k) = (V_P, E_P)$ be the graph with node set and edge set given by

$$V_P = \{Y, Z\} \cup \left\{u_j^i \mid j \in \{1, \ldots, n_i\},\ i \in \{1, \ldots, k\}\right\}$$
$$E_P = \left\{u_j^i u_{j+1}^i \mid j \in \{0, \ldots, n_i\},\ i \in \{1, \ldots, k\}\right\},$$

respectively, where for convenience we label $u_0^i = Y$, and $u_{n_{i+1}}^i = Z$.

We call $P(n_1, \ldots, n_k)$ a *k-path configuration*. A k-path configuration is the union of k disjoint paths connecting Y to Z. The length of a path i is n_i, i.e.,

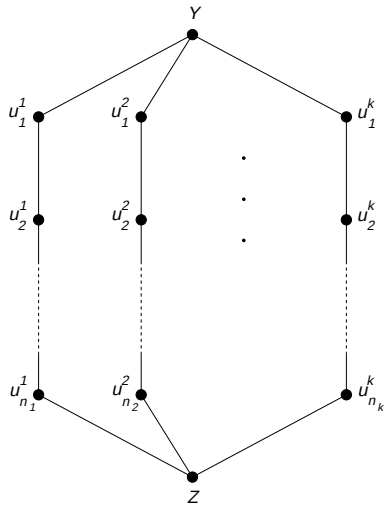

Figure 10. A k-path configuration.

the number of the internal nodes. The nodes Y and Z are called the *odd nodes* of the configuration, all the other nodes are called *even*. The edges of a path configuration are called *path-edges*, see Figure 10.

A *simple path inequality* associated with the k-path configuration $P(n_1, \ldots, n_k)$ is the inequality on $\mathbb{R}^{E_n}$, with $n = 2 + \sum_{i=1}^{k} n_i$, defined (in TT form) by

$$fx \geqslant f_0 = 1 + \sum_{i=1}^{k} \frac{n_i + 1}{n_i - 1}, \tag{5.14}$$

where

$$f_e = \begin{cases} \dfrac{|j-q|}{n_i - 1} & \text{for } e = u_j^i u_q^i, \\ & \quad i \in \{1, \ldots, k\}, \\ & \quad j, q \in \{0, \ldots, n_i + 1\}, \\ & \quad j \neq q, \\ \dfrac{1}{n_i - 1} + \dfrac{1}{n_r - 1} + \left| \dfrac{j-1}{n_i - 1} - \dfrac{q-1}{n_r - 1} \right| & \text{for } e = u_j^i u_q^r, \\ & \quad i \neq r \in \{1, \ldots, k\}, \\ & \quad j \in \{1, \ldots, n_i\}, \\ & \quad q \in \{1, \ldots, n_r\}. \\ 1 & \text{for } e = YZ, \end{cases}$$

A *simple wheelbarrow inequality* associated with the k-path configuration $P(n_1, \ldots, n_k)$ is the inequality on $\mathbb{R}^{E_n}$, with $n = 1 + \sum_{i=1}^{k} n_i$, defined (in

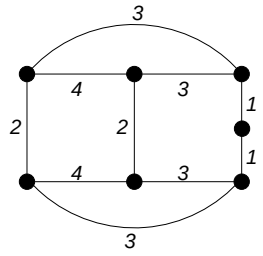

Figure 11. A bicycle inequality with 7 nodes.

TT form) by (5.14), where the coefficients and the right hand side are defined as above, but node Y and all edges incident with it are removed. If both Y and Z and all edges incident with them are removed, the inequality of $\mathbb{R}^{E_n}$, with $n = \sum_{i=1}^{k} n_i$, defined as above is called *simple bicycle inequality*. In Figure 11, a simple bicycle inequality with 7 nodes is illustrated. The coefficient of a missing edge is given by the f-length of a shortest path between its endnodes.

If $n_i = t$ for $t \geqslant 2$ and integer and for $i \in \{1, \ldots, k\}$, then the above inequalities are called *regular* (or *t-regular*).

The PWB inequalities are defined in Cornuéjols, Fonlupt and Naddef (1985) and proved to be facet-defining for GTSP, the polyhedron associated with the graphical traveling salesman problem. A proof that simple PWB inequalities define facets of STSP(n) (with $n \geqslant 6$) is given in Naddef and Rinaldi (1988).

Ladder inequalities

A *ladder inequality* is defined by a family $\mathcal{S}$ of sets

$$\mathcal{S} = \{W_1, W_2, P_1, P_2, T_1, \ldots, T_t, D_1, \ldots, D_m\},$$

with $t \geqslant 0$ and $m \geqslant 0$. The sets W_i, P_i, T_i, and D_i are called *handles*, *pendant teeth*, *regular teeth*, and *degenerate teeth*, respectively. A ladder inequality associated with $\mathcal{S}$ is defined as follows:

$$\sum_{S \in \mathcal{S}} \alpha_S x(E_n(S)) + x(E_n(P_1 \cap W_1 : P_2 \cap W_2)) \leqslant \sum_{S \in \mathcal{S}} \alpha_S |S| - 2t - 3m - 4,$$

where $\alpha_S = 2$ if $S \in \{D_1, \ldots, D_m\}$ and $\alpha_S = 1$ otherwise, and where we denote the set of edges of E_n with one endpoint in X and the other in Y by $E_n(X : Y)$. Observe that the inequality is not in closed form due to the last term of its left hand side. The ladder inequality is facet-defining for STSP(n), ($n \geqslant 8$) if the following conditions are satisfied:

(i) no two teeth intersect;
(ii) the two handles do not intersect;
(iii) P_1 intersects only W_1 and P_2 intersects only W_2;
(iv) each regular or degenerate tooth intersects both handles;

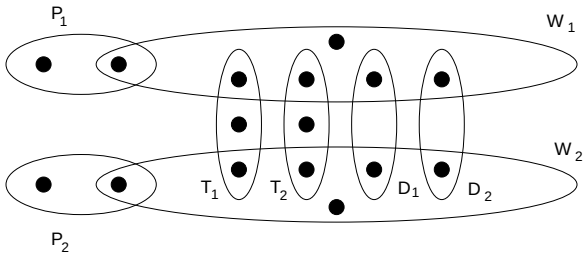

Figure 12. Handles and teeth of a ladder inequality.

(v) each regular or pendant tooth contains at least one node not belonging to any handle;

(vi) each degenerate tooth does not contain nodes that do not belong to one of the two handles;

(vii) $t + m$ is even and at least 2.

A ladder inequality with 16 nodes is shown in Figure 12.

The first ladder inequality with 8 nodes is described in Boyd and Cunningham (1991). The proof that the ladder inequalities are facet-defining for STSP(n), with $n \geqslant 8$ is given in Boyd, Cunningham, Queyranne and Wang (1993).

Crown inequalities

For any integer $k \geqslant 2$ let $C(k) = (V_C, E_C)$ be the graph with the following node and edge sets

$$V_C = \{u_i \mid i \in \{1, \ldots, 4k\}\},$$
$$E_C = \{u_i u_{[i+1]} \mid i \in \{1, \ldots, 4k\}\},$$

where $[i]$ stands for the expression $((i-1) \bmod 4k) + 1)$. We call $C(k)$ a *crown configuration*.

A *simple crown* inequality associated with $C(k)$ is the inequality $fx \geqslant f_0$ (in TT form), where $f_0 = 12k(k-1) - 2$, and, for $i \in \{1, \ldots 4k\}$,

$$f(u_i, u_{[i+j]}) = \begin{cases} 4k - 6 + |j| & \text{for } 1 \leqslant |j| \leqslant 2k-1, \\ 2(k-1) & \text{for } j = 2k. \end{cases}$$

The edges $u_i u_{[2k+i]}$ are called *diameters* of the simple crown inequality. A crown inequality with 8 nodes is shown in Figure 13.

Simple crown inequalities were discovered and proved to define facets of STSP(n) (with $n \geqslant 8$) by Naddef and Rinaldi (1992).

Extensions of facet-defining inequalities

Due to the very complex structure of STSP(n) it is very difficult to describe all inequalities known to define facets of this polytope. A technique used to simplify the description is to define some operations on the inequalities that allow

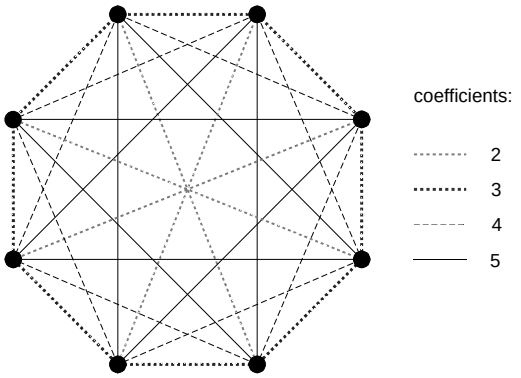

Figure 13. A crown inequality with 8 nodes.

the derivation of new inequalities from others that have already been characterized. Many inequalities can be described in such a constructive way, using the inequalities described above as building blocks. We describe here two kinds of operations: the *zero node-lifting* of a node and the *edge cloning*. Below we show an application of another operation, the *2-sum composition* of inequalities. These operations are better described on a graph associated with an inequality. The graph $G_h = (V_n, E_n, h)$ associated with an inequality $hx \geqslant h_0$ of $\mathbb{R}^{E_n}$ is a weighted complete graph with n nodes, with a weight for each edge that is given by the corresponding inequality coefficient.

Let $hx \geqslant h_0$ be a facet-defining inequality for STSP(n) and let $G_h = (V_n, E_n, h)$ be its associated graph. Let u be a node of G_h and $G_{h^*} = (V_{n+k}, E_{n+k}, h^*)$ be the weighted complete graph obtained by adding k copies of node u and of its star to G_h. More precisely, G_{h^*} contains G_h as a subgraph and $h^*_{ij} = h_{ij}$ for all $e \in E_n$, $h^*_{ij} = h_{uj}$ for all $i \in V_{n+k} \setminus V_n$ and all $j \in V_n$, and $h^*_{ij} = 0$ for all i and j in $V_{n+k} \setminus V_n$. The inequality $h^*x^* \geqslant h_0$ defined on $\mathbb{R}^{E_{n^*}}$ and having G_{h^*} as associated graph is said to be obtained by *zero node-lifting* of node u.

An inequality in TT form with all the coefficients strictly positive is called *simple*. A facet-defining inequality in TT form that has a zero coefficient is always derivable from a simple inequality in TT form by a repeated application of the zero node-lifting. In Naddef and Rinaldi (1993) a simple sufficient condition is given for an inequality in TT form, obtained by zero node-lifting of a facet-defining inequality, to inherit the property of being facet-defining. This condition is verified by all the inequalities known to date that are facet-defining for STSP(n) and in particular, by the inequalities described above. For an inequality obtained by zero node-lifting of a simple inequality, we use the convention of keeping the same name of the simple inequality but dropping the word "simple". Consequently, the *PWB inequalities* and the *crown inequalities* are obtained by zero node-lifting of their corresponding simple archetypes and are all facet-defining for STSP(n).

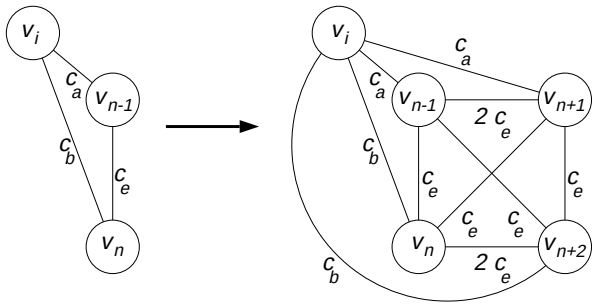

Figure 14. Cloning of an edge.

Let uv be an edge of G_h and let $G_{h'} = (V_{n+2}, E_{n+2}, h')$ be the weighted complete graph with $V_{n+2} = V_n \cup \{u', v'\}$ defined as follows. The graph G_h is a subgraph of $G_{h'}$ and h' is defined as follows:

$$\begin{aligned}
h'(u', j) &= h(u, j) && \text{for all } j \in V_n \setminus \{u\},\\
h'(v', j) &= h(v, j) && \text{for all } j \in V_n \setminus \{v\},\\
h'(u, u') &= h'(v, v') = 2h(u, v),\\
h'(u', v') &= h(u, v).
\end{aligned}$$

The inequality $h'x' \geqslant h_0 + 2h(u, v)$ defined on $\mathbb{R}^{E_{n+2}}$ and having $G_{h'}$ as associated graph is said to be obtained from $hx \geqslant h_0$ by *cloning* the edge uv.

In Figure 14, we give an example of the cloning of an edge. The inequality represented by the graph on the right hand side is obtained by cloning the edge e. The cloning of an edge can be repeated any number of times. In Naddef and Rinaldi (1993) sufficient conditions are given for an edge of the graph associated with a facet-defining inequality in TT form to be clonable, i.e., to be cloned as described before, while producing a facet-defining inequality. A path-edge of a PWB inequality belonging to a path of length 2 is clonable. The inequalities obtained by cloning any set of these edges any number of times are called *extended* PWB inequalities (see Naddef and Rinaldi (1988)). A diameter edge of a crown inequality is clonable. The inequalities obtained by cloning any set of diameters any number of times are called *extended crowns* (see Naddef and Rinaldi (1992)).

A generalization of the zero node-lifting is described in Naddef and Rinaldi (1993) and called 1-node lifting. Several sufficient conditions for an inequality, obtained by zero node-lifting of a facet-defining inequality, to be facet-defining for STSP(n) are given in Queyranne and Wang (1993). Some of them are very simply to check and apply to basically all known inequalities facet-defining for STSP(n).

2-sum composition of path inequalities

The *2-sum composition* of inequalities is an operation that produces new facet-defining inequalities by merging two inequalities known to be facet-defining. Instead of describing the operation in general, we give an example of its application

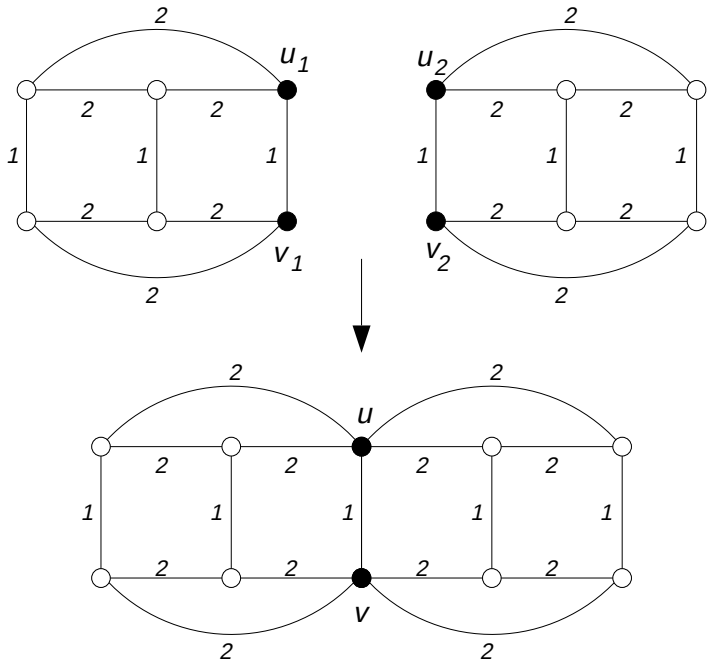

Figure 15. The composition of two bicycle inequalities.

that produces a large class of facet-defining inequalities for STSP(n), called the *regular parity path-tree inequalities*. We define these inequalities recursively.

A simple regular PWB inequality is a *regular parity path-tree inequality*. Let $f^1x^1 \geqslant f_0^1$ and $f^2x^2 \geqslant f_0^2$ be two regular parity path-tree inequalities and let $G^1 = (V_{n_1}, E_{n_1}, f^1)$ and $G^2 = (V_{n_2}, E_{n_2}, f^2)$ be their corresponding associated graphs. Let u_1v_1 be a path-edge of the first inequality and u_2v_2 a path-edge of the second, satisfying the following conditions:

(i) the nodes u_1 and u_2 have the same parity (they are either both odd or both even) and so do the nodes v_1 and v_2;

(ii) $f^1(u_1, v_1) = f^2(u_2, v_2) = \varepsilon$.

The condition (ii) can always be satisfied by multiplying any of the two inequalities by a suitable positive real number. Let $G' = (V, E', f')$ be the weighted graph with $n = n_1 + n_2 - 2$ nodes obtained from G^1 and G^2 by identifying the nodes u_1 and u_2 and the nodes v_1 and v_2. We call the nodes that result from the identification of the two pairs u and v, respectively. Each of these two nodes is *odd* if it arises by the identification of two odd nodes, otherwise it is *even*. The edge uv qualifies as a *path-edge*. The node and the edge set of G' are given by $V' = (V_{n_1} \cup V_{n_2} \setminus \{u_1, v_1, u_2, v_2\}) \cup \{u, v\}$ and $E' = E_{n_1} \cup E_{n_2} \setminus \{u_1v_1, u_2v_2\} \cup \{uv\}$.

Let $G = (V, E, f)$ be the weighted graph obtained from G' by adding the edge ij for all $i \in V_{n_1} \setminus \{u_1, v_1\}$ and all $j \in V_{n_2} \setminus \{u_2, v_2\}$, with weight $f(i, j)$ given by the f-length of the shortest path from i to j in G'.

The inequality $fx \geqslant f_0 = f_0^1 + f_0^2 - 2\varepsilon$, having G as associated graph, is a *regular parity path-tree* inequality.

The PWB inequalities that are used in the composition of a regular parity path-tree are called *components* of the inequality. Figure 15 illustrates the composition of two bicycle inequalities by a 2-sum operation.

The s-sum composition of inequalities (of which the 2-sum is a special case) is introduced in Naddef and Rinaldi (1991) in the context of GTSP, as a tool to produce new facet-defining inequalities from known ones. The 2-sum composition for STSP is described in Naddef and Rinaldi (1993). A proof that regular parity path-tree inequalities define facets of STSP(n), with $n \geqslant 10$, is given in Naddef and Rinaldi (1988). Other composition operations for facet-defining inequalities of STSP are described in Queyranne and Wang (1990).

Relations between TT and other inequalities

The inequalities in TT form described above include most of the inequalities presently known that define facets of STSP(n). We conclude this subsection on the facets of STSP(n) by briefly showing how these inequalities are related to the other known facet-defining inequalities, described in closed form.

– The *2-matching inequalities* are 2-regular PWB inequalities derived from simple PWB inequalities by allowing zero node-lifting only on the nodes Y and Z.

– The *Chvatál comb inequalities* are 2-regular PWB inequalities derived from simple PWB inequalities by allowing zero node-lifting on all nodes but u_2^i for $i \in \{1, \ldots, k\}$.

– The *comb inequalities* are 2-regular PWB inequalities.

– The *chain inequalities* (see below) are 2-regular PWB inequalities where *only one* path-edge is cloned any number of times (consequently the chain inequalities are a special case of the extended PWB inequalities, and so they are facet-defining for STSP(n)).

– The *clique-tree inequalities* are regular parity path-tree inequalities obtained from 2-regular PWB inequalities with the condition that the nodes Z of all the component PWB inequalities are identified together in a single node.

Other facet-defining inequalities

To complete the list of all known facet-defining inequalities for STSP we mention a few more. Chvátal (1973) shows that an inequality defined by the Petersen graph is facet-defining for STSP(10). A generalization of this inequality, which is facet-defining for $n > 10$, is given in Maurras (1975). Three inequalities facet-defining for STSP(8) are described in Christof, Jünger and Reinelt (1991). The inequalities have to be added to trivial, subtour elimination, PWB, chain, ladder and crown inequalities to provide a complete description of STSP(8).

Other valid inequalities for STSP(n)

The fact that collections of node sets, satisfying some conditions, can be used to describe facet-defining inequalities for STSP(n) in closed form has motivated

many researchers to proceed along these lines and consider collections of node sets satisfying more complex conditions.

A first generalization of comb inequalities obtained by replacing a tooth by a more complex structure leads to the *chain* inequalities, described in Padberg and Hong (1980), where only a proof of validity is given.

Another generalization of the comb inequalities is obtained by allowing not just a single handle but a nested family of handles. The inequalities obtained in this way are called *star* inequality and are described in Fleischmann (1988) where it is proved that they are valid for GTSP. The star inequalities properly contain the PWB inequalities but for those which are not PWB inequalities only a proof of validity for GTSP is currently known. Actually, some of them do not define facets of STSP(n) (see Naddef (1990)) and some others do (see Queyranne and Wang (1990)).

A generalization of the clique-tree inequalities is produced by relaxing the conditions (iii) and (vi) of the definition. The resulting inequalities are called *bipartition* inequalities (Boyd and Cunningham (1991)). Further generalizations lead to the *hyperstar* inequalities (Fleischmann (1987)) and to the *binested* inequalities (Naddef (1992)).

For all these inequalities only a proof of validity is given in the cited papers. Therefore, these inequalities provide good candidates for members of the set $\mathcal{B}^{\leqslant}$ of all facet-defining inequalities of STSP(n), and can be used to provide stronger LP relaxations to the TSP.

For a complete survey on these classes of inequalities see Naddef (1990).

5.6. The separation problem for STSP(n)

In order to have a relaxation that produces a good lower bound, it is necessary that the LP relaxation contains at least the subtour elimination inequalities. The number of these constraints is exponential in n (it is precisely $2^{n-1}-n-1$) and it becomes much larger if other inequalities, like 2-matching or comb inequalities, are added to the relaxation. Consequently, to find the optimal solution of an LP relaxation we cannot apply a linear programming algorithm directly to the matrix that explicitly represents all these constraints.

Let $\mathcal{L}$ be a system that contains a huge number of inequalities which are valid for STSP(n) and suppose that we want to solve the problem

Problem 5.2

$$\begin{aligned} \min \quad & cx \\ \text{s.t.} \quad & A_n x = \mathbf{2}, \\ & lx \leqslant l_0 \qquad \text{for all } (l, l_0) \in \mathcal{L}, \\ & \mathbf{0} \leqslant x \leqslant \mathbf{1}. \end{aligned}$$

In principle Problem 5.2 can be solved by the following cutting-plane procedure:

procedure CUTTING_PLANE

Input: n, c, a family of "known" inequalities $\mathcal{L}$.

(1) Set $\mathcal{L}' = \emptyset$
(2) Solve $\min\{cx \mid A_n x = \mathbf{2},\ lx \leqslant l_0$ with $(l, l_0) \in \mathcal{L}',\ \mathbf{0} \leqslant x \leqslant \mathbf{1}\}$ and let z be its solution.
(3) Find one or more inequalities in $\mathcal{L}$ violated by z.
(4) If none is found, then stop. Otherwise add the violated inequalities to $\mathcal{L}'$ and go to (2).

Procedure CUTTING_PLANE stops after a finite number of steps, because $\mathcal{L}$ is finite. The *core* of the procedure is the problem solved in Step (3), which is called the *separation problem* and is formally stated as follows:

Problem 5.3 *Given a point $z \in \mathbb{R}^{E_n}$ and a family $\mathcal{L}$ of inequalities in $\mathbb{R}^{E_n}$, identify one (or more) inequalities in $\mathcal{L}$ violated by z or prove that no such inequality exists.*

An *exact separation procedure* for a family of inequalities $\mathcal{L}$ is one that solves Problem 5.3, a *heuristic separation procedure* is one that may find violated inequalities, but that in case it cannot find any, is unable to guarantee that no violated inequalities exist in $\mathcal{L}$.

The faster, the more efficient and the more "prolific" the separation procedure used in Step (3) of the cutting plane procedure, the faster is the resolution of Problem 5.2. This statement is substantiated by a result of Grötschel, Lovász and Schrijver (1981,1988) and Padberg and Rao (1981) that can be stated as follows:

Proposition 5.4 *Problem 5.2 is solvable in polynomial time if and only if Problem 5.3 is solvable in polynomial time.*

Unfortunately an exact separation procedure that runs in polynomial time is presently known only for two classes of TSP inequalities. These classes are the subtour elimination and 2-matching inequalities. It follows that the lower bound produced by an LP relaxation that has all subtour and all 2-matching inequalities can be computed in polynomial time. Heuristic separation procedures have been developed for the comb and the clique-tree inequalities.

Let $z \in \mathbb{R}^{E_n}$ be a point that we want to separate from STSP(n) with a facet-defining inequality belonging to a given class. It is assumed, without loss of generality, that z satisfies (5.7) and (5.9). Usually, all separation algorithms operate on a graph associated with z, the support graph of z. The *support graph* $G_z = (V_n, E, z)$ of z is a weighted graph whose edge set E contains all edges of E_n corresponding to a positive component of z. The weight associated with $e \in E$ is z_e. The edges of G_z with weight 1 are called *1-edges*.

The separation for subtour elimination inequalities

The point z violates a subtour elimination inequality (5.8) if and only if the minimum weight cut in G_z has weight less than 2. Since the minimum weight cut in a graph with nonnegative edge weights can be found in polynomial time with the algorithm proposed by Gomory and Hu (1961), Problem 5.3 can be solved in polynomial time for the subtour elimination inequalities. Therefore, by Proposition 5.4, the subtour relaxation is solvable in polynomial time.

The Gomory-Hu algorithm is based on the computation of $n-1$ maximum flow problems on some weighted graphs derived from G_z. The complexity of a maximum flow algorithm is $O(|V||E|\log(|V|^2/|E|))$ (see Goldberg and Tarjan (1988)), and so the complexity of the algorithm is $O(|V|^2|E|\log(|V|^2/|E|))$. For large instances of the TSP such a complexity is expensive in terms of actual computation time, since Problem 5.3 has to be solved several times in a branch and cut algorithm (see Section 6). For this reason many heuristic procedures have been proposed to find violated subtour elimination inequalities quickly (see, e.g., Crowder and Padberg (1980) and Grötschel and Holland (1991)). Padberg and Rinaldi (1990a) describe an exact algorithm that finds the minimum weight cut in a graph with a drastic reduction in the number of maximum flow computations. Even though the algorithm has the same worst case time bound as the Gomory-Hu algorithm, it runs much faster in practice and it allows the execution of an exact separation algorithm at every iteration of a branch and cut algorithm. The idea of this algorithm is to exploit some simple sufficient conditions on G_z that guarantee that two nodes belong to the same shore of a minimum cut. If two nodes satisfy one of these conditions then they are *contracted*. The *contraction* of two nodes in G_z produces a new weighted graph where the two nodes are identified into a single node; loops are removed and any two parallel edges are replaced by a single edge with weight equal to the sum of their weights. The resulting graph has one node less and the shores of a minimum cut in it can be turned into the shores of a minimum cut in G_z, by replacing the node that results from the identification with the two original nodes. The contraction of a pair of nodes can be applied recursively until no more reductions apply. At this point the Gomory-Hu algorithm can be applied to the resulting reduced graph.

A different algorithm also based on the contraction of pairs of nodes is proposed by Nagamochi and Ibaraki (1992a, 1992b). After each major step of the algorithm a pair of nodes u and v is identified that have the following property. Either $\delta(\{u\})$ is a minimum cut or $\delta(\{v\})$ is a minimum cut or u and v belong to the same shore of a minimum cut. After recording the better of the two cuts $\delta(\{u\})$ and $\delta(\{v\})$, u and v are contracted. After $|V|-1$ contractions the graph reduces to a single node and the best of the recorded cuts is the minimum cut. The algorithm does not require the computation of a maximum flow and runs in $O(|E||V|+|V|^2\log|V|)$ time.

Another algorithm for the minimum cut is proposed in Hao and Orlin (1992). It is a modified version of a maximum flow algorithm and it is able to compute the minimum cut in the same running time required by the computation of a single maximum flow ($O(|V||E|\log(|V|^2/|E|))$).

Finally we mention a randomized algorithm for computing a minimum cut with high probability. The algorithm runs in $O(|E||V|^2 \log^3 |V|)$ time (Karger (1993)) and an improved version in $O(|V|^2 \log^3 |V|)$ time (Karger and Stein (1993)).

All these algorithms can be conveniently utilized to solve the separation problem of the subtour elimination inequalities efficiently.

The separation for the 2-matching inequalities

Let G_z be defined as before and let us apply the following operations on it. First every edge e of G_z is split in two edges e' and e'' that have the same weight as e and are incident with a new node v_e. The resulting graph G'_z has $n + |E|$ nodes and $2|E|$ edges. All its nodes are labeled *even*. Then for each pair of edges $\{e', e''\}$ that comes from splitting, either e' or e'' is complemented, i.e., its weight z_e is replaced by $1 - z_e$ and the label of each endpoint becomes *odd* if it was *even* (and *even* if it was *odd*). Call G^*_z the new weighted and labeled graph. The number of odd nodes of G^*_z is even and at least $|E|$ (each of the nodes that is produced by splitting is odd). An *odd cut* of G^*_z is a cut whose shores have an odd number of odd nodes. Padberg and Rao (1982) propose an algorithm that finds the minimum weight odd cut of a labeled weighted graph in polynomial time. They also prove that the vector z satisfies all 2-matching inequalities if and only if the minimum odd cut in G^*_z has weight at least 1. Consequently, Problem 5.3 can be solved in polynomial time for the 2-matching inequalities.

Suppose that G^*_z has an odd cut with weight less than 1. Let us see how a 2-matching inequality violated by z can be generated. At most one of the two edges e' and e'' produced by splitting an edge e of G_z belongs to the cut. If such an edge has been complemented, then the endpoints of e are taken as a tooth of the inequality. All nodes of the original graph G_z that belong to one of the two shores of the cut are taken as the handle of the inequality. It may happen that after this construction two teeth of the 2-matching intersect in a node u, thus violating the condition (iii) of the definition of comb inequalities. In this case the two teeth are removed from the set of teeth. In addition, if u belongs to the handle, then it is removed from it; if it does not, then it is added to it.

The Padberg-Rao algorithm, which is based on the Gomory-Hu algorithm, requires as many max-flow calculations on G^*_z as the number of its odd nodes. As observed for the separation of the subtour elimination inequalities, this algorithm may be very time consuming. Therefore some reductions of the number of nodes, odd nodes, and edges of the graph to which the Padberg-Rao algorithm is applied have been proposed (see Padberg and Grötschel (1985), Grötschel and Holland (1987), and Padberg and Rinaldi (1990b)). A detailed description of an implementation of the Padberg-Rao algorithm is given in Grötschel and Holland (1987). A simple implementation of the Gomory-Hu algorithm is described in Gusfield (1987).

Although the reductions applied to G^*_z may produce a sensible speed up in the solution of Problem 5.3, an exact solution of this problem may still be too expensive for large TSP instances. For this reason heuristic separation procedures are often used for 2-matching inequalities.

A first procedure, similar to the exact one sketched above, is proposed in Padberg and Rinaldi (1990b). In this procedure all nodes of G_z are labeled *even* and all edges with weight greater than or equal to 0.5 are complemented. Then the reductions mentioned before are applied. Finally the Padberg-Rao algorithm is applied to the resulting graph, which is smaller (in terms of nodes, odd nodes and edges) than G_z^*.

Another heuristic, proposed first in Padberg and Hong (1980), follows a completely different approach. All 1-edges (or all edges with weight close to 1) are removed from G_z. The resulting graph is decomposed into its biconnected components. Each biconnected component with at least three nodes is considered as the handle of a possibly violated 2-matching inequality. The teeth of the inequality are the endpoints of edges of the original graph G_z with only one endpoint in the handle and with "big" weight (usually greater than or equal to 0.5). The procedure is fast and quite successful. Implementations and variations of this heuristic separation algorithm are described in Padberg and Grötschel (1985), Grötschel and Holland (1987), and Padberg and Rinaldi (1990b). In the last paper a variation is given that also produces violated Chvatál comb inequalities.

The separation for comb inequalities

As said above there is no known exact polynomial separation procedure for comb inequalities. The heuristic procedures proposed in the literature exploit the following two facts:

(a) comb inequalities expressed in TT form (i.e., 2-regular PWB inequalities) can be obtained by zero node lifting of 2-matching inequalities (i.e., simple 2-regular PWB inequalities)

(b) a separation procedure for 2-matching inequalities is available.

Let us assume that for a family $\mathcal{F}$ of simple inequalities in TT form, a separation procedure is available and that we want to exploit this procedure to find violated inequalities that are obtained by zero node-lifting of inequalities in $\mathcal{F}$. We can proceed as follows. Let $S \subset V_n$ be a set of nodes whose corresponding subtour elimination inequality is satisfied by z with equality, i.e.,

$$z(\delta(S)) = 2. \tag{5.15}$$

Let $\hat{G}_{\hat{z}} = (V_{\hat{n}}, \hat{E}, \hat{z})$, with $\hat{n} = n - |S| + 1$, be the weighted graph obtained by recursively contracting pairs of nodes in S until S is replaced by a single node s. It is easy to see that $\hat{z} \in \mathbb{R}^{E_{\hat{n}}}$ satisfies all inequalities (5.7), (5.8), and (5.9), and so it can be thought of as the solution of an LP relaxation of a TSP on a complete graph of $\hat{n}$ nodes. Suppose that we are able to separate $\hat{z}$ from STSP($\hat{n}$) with an inequality $\hat{h}\hat{x} \geqslant h_0$ in $\mathcal{F}$. It follows that $\hat{h}\hat{z} < h_0$. Let $hx \geqslant h_0$, with $h \in \mathbb{R}^{E_n}$, be the inequality obtained by zero lifting of node s. It follows that $hz = \hat{h}\hat{z}$, and so the inequality $hx \geqslant h_0$ is violated by z.

In conclusion, a separation procedure for inequalities obtained by zero node-lifting of inequalities in $\mathcal{F}$ can be devised by contracting any number of sets satisfying (5.15) by applying the separation procedure for the inequalities in the family $\mathcal{F}$ to the reduced graph, and finally by producing a violated inequality in $\mathbb{R}^{E_n}$ by zero lifting all nodes that are produced by contraction.

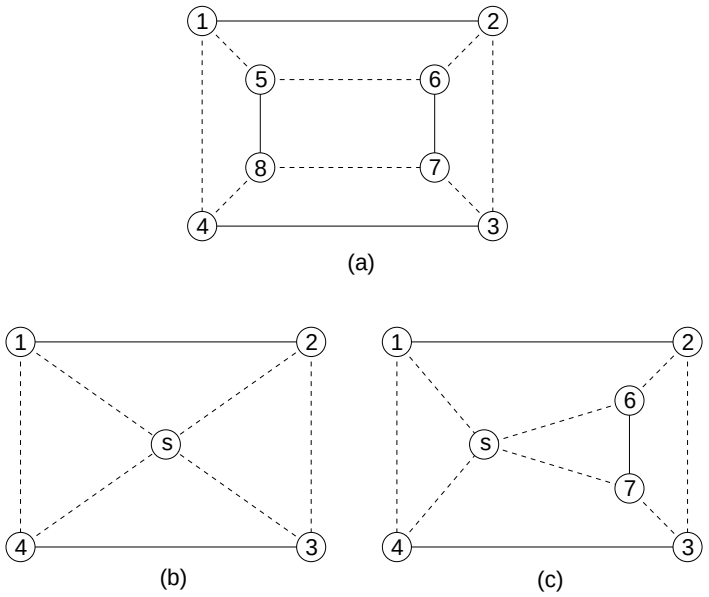

Figure 16. An example of a nonshrinkable set satisfying (5.15).

Unfortunately this procedure does not work in all cases. It may be the case that $z \notin \mathrm{STSP}(n)$, i.e., there exists a valid inequality for $\mathrm{STSP}(n)$ that is violated by z, but the vector $\hat{z}$, produced by contracting a set S satisfying (5.15), belongs to $\mathrm{STSP}(\hat{n})$, and so all valid inequalities for $\mathrm{STSP}(\hat{n})$ are satisfied by $\hat{z}$. Put differently, the operation associated with the contraction of a set S may map a point outside $\mathrm{STSP}(n)$ to a point inside $\mathrm{STSP}(\hat{n})$, which is an undesirable situation. Figure 16 gives an example of this bad case. The graph in Figure 16(a) is the support graph of a point z (solid lines are edges with weight 1, while dotted lines are edges with value 0.5). The set $S = \{5, 6, 7, 8\}$ satisfies (5.15). If we contract this set we get the graph of Figure 16(b). This is the support graph of a vector $\hat{z}$ which is the convex combination $z^1/2 + z^2/2$ of the incidence vectors z^1 and z^2 of the two Hamiltonian cycles $\{1, 2, s, 3, 4, 1\}$ and $\{1, 2, 3, 4, s, 1\}$, respectively. Consequently, $\hat{z}$ is a point of $\mathrm{STSP}(\hat{n})$. However if in the graph of Figure 16(a) the set $S = \{5, 8\}$ is contracted, in the resulting graph shown in Figure 16(c) there is a violated Chvátal comb with handle $\{1, s, 4\}$ and teeth $\{1, 2\}$, $\{4, 3\}$, and $\{s, 6, 7\}$. By zero-lifting node s we obtain a violated comb with handle $\{1, 4, 5, 8\}$ and teeth $\{1, 2\}$, $\{4, 3\}$, and $\{5, 6, 7, 8\}$.

In Padberg and Rinaldi (1990b) a set S satisfying (5.15) is called *shrinkable* if the graph obtained by contracting S is not the support graph of a point of $\mathrm{STSP}(\hat{n})$. Therefore, the shrinkable sets are those that can be safely contracted. In the same paper sufficient conditions are given for a set to be shrinkable and some cases of shrinkable sets are provided (see Figure 17). Finally a heuristic separation procedure for comb inequalities is described, which is based on the scheme outlined before and utilizes those special cases of shrinkable sets. The

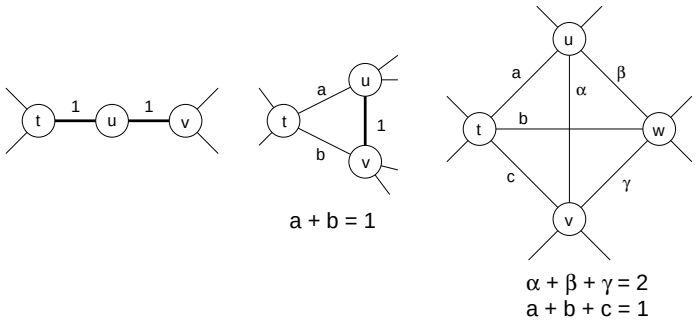

Figure 17. Shrinkable set.

heuristic separation procedure for comb inequalities described in Grötschel and Holland (1991) is similar.

The separation for clique-tree inequalities

The separation problem for clique-tree inequalities has not been studied to the same extent as for comb inequalities. Two simple procedures are published in Padberg and Rinaldi (1990b). One finds violated clique-tree inequalities with two handles. The other is an extension of the procedure for the 2-matching inequalities, based on the decomposition of a subgraph of G_z into biconnected components.

The improvement in the relaxation due to these procedures is not impressive.

The separation for PWB inequalities

A simple heuristic procedure for finding violated PWB inequalities is described in Clochard and Naddef (1993). The procedure starts by finding a violated comb inequality (a 2-regular PWB) and then tries to "extend" its paths (that all have length 2) to paths of bigger length. The procedure works well when the point z to be separated has many integer components. This is often the case when z is the optimal solution over the subtour elimination polytope. The support graph G_z of such a point often has long paths of consecutive 1-edges. These edges are good candidates to become path-edges of a violated PWB inequality.

The results obtained by applying this procedure, as reported in Clochard and Naddef (1993), are very promising.

5.7. Comparison of LP relaxations

To conclude this section, we show how different LP relaxations behave in terms of the value of the lower bound that they produce. To do so, we have computed the lower bound associated with each relaxation for each of the instances of our standard set of test problems. The results of this computation are reported in Table 5. For each test problem we have computed the *fractional 2-matching* relaxation, i.e., the relaxation obtained by solving Problem 5.1 with only the constraints (5.7) and (5.9). This relaxation is not properly an LP relaxation as

defined in Subsection 5.1, because it may have an integer solution that is the incidence vector of a nonconnected 2-matching, since the constraints (5.8) are not imposed. The fractional 2-matching is often the first relaxation to be produced in any cutting-plane algorithm (see Section 6) since it requires only a small number (polynomial in n) of constraints and its polytope properly contains the polytope of any other LP relaxation considered in this section. In addition, we have computed the lower bound of the subtour relaxation, of the 2-matching relaxation and of an LP relaxation that includes subtour elimination, 2-matching, comb and clique-tree inequalities. The latter relaxation has been computed with the algorithm described in Padberg and Rinaldi (1991) that is based on heuristic separation procedures for comb and clique-tree inequalities. Consequently, the lower bound that we report is inferior to the one that would be obtained by optimizing over the polytope defined by subtour elimination, comb, and clique-tree inequalities. The lower bound has been computed from the relaxation constructed by the Padberg-Rinaldi algorithm just before going to the branching phase. Since the algorithm tries to minimize the overall computation time, it may resort to the branching phase even though more inequalities can be added to the relaxation. Therefore, this lower bound is an underestimate of the bound obtainable with the separation procedures described in Padberg and Rinaldi (1990b) and used in the algorithm. We denote this relaxation by "all cuts" in the header of Table 5.

In the first five columns of the table the names of the test problems are reported along with the value $r = 100 \times (LB/OPT)$ for each lower bound, where LB is the lower bound value and OPT is the objective function value of an optimal solution. As mentioned in Section 2, two TSP instances are equivalent if one is obtained by adding a constant C to each component of the objective function of the other. However, the value r of any of the four bounds is not the same for two equivalent instances and tends to 100 when C tends to infinity.

To overcome this problem, we have also computed a different index to compare the bounds. This index is the same for two equivalent instances and shows, better than the previous one, how the different relaxations contribute to covering the gap between the fractional 2-matching bound and the optimal solution value. The index is given by the ratio $R = 100 \times (LB - F2M)/(OPT - F2M)$, where $F2M$ is the value of the fractional 2-matching bound. A value of R equal to 100 shows that the relaxation is sufficient to provide an optimal solution to the problem, i.e., that no recourse to branch and cut is necessary.

In Subsection 5.5 we have seen many inequalities that define facets of STSP. Only for a few of them a satisfactory separation procedure is available at present. It would be interesting to know for which inequalities it would be profitable to invest in further research to solve the corresponding separation problem. In other words, given a class of valid inequalities $\mathcal{C}$, it would be interesting to know the improvement on the lower bound that we would get by adding them all to the constraints of the subtour relaxation. This improvement would measure the strength of the inequalities contained in the class $\mathcal{C}$. In Goemans (1993) this analysis is carried out for GTSP. For a class of inequalities $\mathcal{C}$ a theoretical computation is provided for the improvement that would result when all the

Table 5

Comparison of LP relaxations

Problem	r f2-m.	r 2-m.	r subtour	r all cuts	R 2-m.	R subtour	R all cuts
lin105	98.8	99.6	99.9	100.0	67.4	94.9	100.0
pr107	55.5	55.5	100.0	100.0	0.0	100.0	100.0
pr124	85.0	87.2	98.4	99.9	14.8	89.1	99.4
pr136	92.2	92.3	99.1	99.8	1.0	88.9	97.0
pr144	56.0	57.5	99.4	99.9	3.4	98.7	99.8
pr152	60.4	60.4	99.4	99.6	0.0	98.4	99.1
u159	96.7	97.8	99.6	100.0	34.3	88.9	100.0
rat195	97.8	98.9	99.0	99.8	48.5	53.0	90.6
d198	74.8	75.3	99.7	100.0	2.0	98.7	100.0
pr226	68.7	71.1	99.7	100.0	7.7	98.9	100.0
gil262	93.5	94.5	99.0	99.9	16.4	84.9	99.0
pr264	74.7	75.7	99.8	100.0	4.2	99.1	99.9
pr299	93.3	94.7	98.3	99.9	21.0	74.9	98.9
lin318	92.7	93.5	99.7	99.9	10.7	95.4	99.1
rd400	95.2	96.0	99.2	99.8	17.3	83.0	96.7
pr439	87.4	89.0	98.8	99.7	12.4	90.4	97.9
pcb442	98.7	99.2	99.5	99.9	40.5	58.7	95.8
d493	93.3	94.6	99.5	99.9	18.7	92.6	99.2
u574	92.8	93.3	99.5	100.0	6.4	92.8	100.0
rat575	98.1	98.7	99.3	99.9	29.9	61.0	93.6
p654	82.5	82.8	99.9	100.0	1.5	99.2	100.0
d657	95.4	96.4	99.1	99.8	21.2	79.6	96.7
u724	96.7	96.8	99.4	100.0	3.6	83.0	98.7
rat783	97.3	97.5	99.6	100.0	6.5	86.1	99.2
pr1002	93.0	93.3	99.1	99.9	4.7	87.5	98.9
pcb1173	97.7	98.6	99.0	99.9	40.1	58.1	96.9
rl1304	90.9	91.1	98.5	99.9	1.8	83.2	98.9
nrw1379	98.2	98.5	99.6	100.0	21.1	76.9	98.1
u1432	99.2	99.5	99.7	100.0	38.7	66.4	99.3
pr2392	95.1	96.5	98.8	100.0	27.4	75.3	100.0

inequalities are added to the subtour relaxation of GTSP. The subtour relaxation of GTSP has the nonnegativity constraints and the constraints (5.6). The results, which can serve as a useful indication also for STSP, are summarized in Table 6.

6. Finding optimal and provably good solutions

The discussion of the last two sections gives us a variety of tools for computing Hamiltonian cycles and lower bounds of the length of Hamiltonian cycles. The lower bounds enable us to make statements of the form: "The solution found

Table 6

Quality of GTSP facet-defining inequalities

Class of inequalities	strength
Comb	10/9
Clique tree	8/7
Path	4/3
Crown	11/10

by the heuristic algorithm is at most $p\%$ longer than the shortest Hamiltonian cycle". Probably most practitioners would be completely satisfied with such a quality guaranteed solution and consider the problem solved, if the deviation p is only small enough. In this section, we will consider algorithms which can achieve any desired quality (expressed in terms of p), including the special case $p = 0$, i.e., optimality.

6.1. Branch and bound

All published methods satisfying the above criterion are variants of the branch and bound principle. Branch and bound algorithms are well known so that we can omit a formal definition here. The flowchart of Figure 18 gives the basic control structure of a branch and bound algorithm for a combinatorial optimization problem whose objective function has to be minimized.

A branch and bound algorithm maintains a list of subproblems of the original problem whose union of feasible solutions contains all feasible solutions of the original problem. This list is initialized with the original problem itself.

In each major iteration the algorithm selects a current subproblem from this list and tries to "fathom" it in either of the following ways: a lower bound for the value of an optimal solution of the current subproblem is derived that is at least as high as the value of the best feasible solution found so far, or it is shown that the subproblem does not contain any feasible solution, or the current subproblem is solved to optimality. If the current subproblem cannot be fathomed according to one of these criteria, then it is split into new subproblems whose union of feasible solutions contains all feasible solutions of the current problem. These newly generated problems are added to the list of subproblems. This iteration process is performed until the list of subproblems is empty.

Whenever a feasible solution is found in the process, its value constitutes a global upper bound for the value of an optimal solution. Similarly, the minimum of the local lower bounds at any point of the computation is a global lower bound for the optimal value of the objective function. For the TSP, this means that

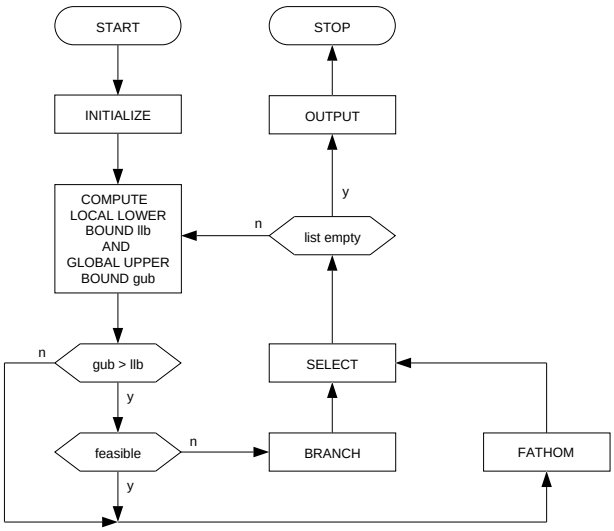

Figure 18. Flowchart of a branch and bound algorithm.

during the execution of a branch and bound algorithm a sequence of feasible solutions of decreasing lengths and a sequence of lower bounds of increasing values is produced. The algorithm terminates with the optimal solution as soon as the lower bound coincides with the length of the shortest Hamiltonian cycle found. If we take the point of view that practical problem solving consists of producing a solution of a prescribed quality $p\%$, then a branch and bound algorithm can achieve this goal if it stops as soon as it has found a solution T of length $c(T)$ and lower bound l such that $(c(T) - l)/l \leqslant p/100$. A general survey of the branch and bound method applied to the TSP has been given in Balas and Toth (1985). One of the crucial parts of such an algorithm is the lower bounding technique. Lower bounds are usually computed by solving an appropriate relaxation of the TSP.

Several relaxations have been considered for the TSP. Among them are *n-path relaxation*, *assignment relaxation*, or also so-called *additive bounding procedures*. For information on these approaches see Balas and Toth (1985) and Carpaneto, Fischetti and Toth (1989). A branch and bound algorithm that uses the *2-matching relaxation* has been implemented by Miller, Pekny and Thompson (1991).

We shall discuss branch and bound algorithms based on the two relaxation methods discussed in Section 5, the *1-tree relaxation* and the *LP relaxations*.

6.2. 1-tree relaxation

First we discuss the method of determining lower bounds based on the approximation of the optimum value of the Lagrangean dual based on the 1-tree relaxation (see Subsection 5.1). This technique is due to Held and Karp (1970, 1971).

Recall that the corresponding Lagrangean problem is of the form $\max_\lambda\{L(\lambda)\}$. Determining $L(\lambda)$ for a given vector of node multipliers λ amounts to computing a 1-tree with respect to the modified edge weights $c_{ij} + \lambda_i + \lambda_j$ and subtracting $2\sum_{i=1}^{n} \lambda_i$.

The function L is piecewise linear, concave, and nondifferentiable. A method for maximizing L is the *subgradient method*. This method is a generalized gradient ascent method where in each iteration step the next iterate (i.e., the next vector of node multipliers) is determined by taking a step in the direction of a *subgradient*.

A minimum 1-tree readily supplies a subgradient as follows. Namely, let δ_i be the degree of node i in the minimum 1-tree with respect to node multipliers λ. Then the vector $(\delta_1 - 2, \delta_2 - 2, \ldots, \delta_n - 2)$ is a subgradient of L at λ.

If L is bounded from above (which is the case for the TSP) and if the step lengths α_k satisfy both $\lim_{k\to\infty} \alpha_k = 0$ and $\sum_{k=0}^{\infty} \alpha_k = \infty$, then the method converges to the maximum of L (Polyak (1978)). However, it turned out in practice that such step lengths formulae lead to very slow convergence. There are update formulas for the α_k that do not satisfy the requirement $\sum_{k=0}^{\infty} \alpha_k = \infty$, but lead to better convergence in practice.

Based on the references Balas and Toth (1985), Volgenant and Jonker (1982), Christofides (1979) and on own experiments we use the following implementation.

procedure 1TREE_BOUND

(1) Let α_1 be the initial step length and γ a decrement factor for the step length.
(2) Set $\lambda_i^1 = 0$ for every node i, and $k = 1$.
(3) Perform the following steps for a specified number of iterations or until no significant increase in the lower bound can be observed for several iterations.
  (3.1) Compute a minimum spanning tree with respect to the edge weights $c_{ij} + \lambda_i + \lambda_j$.
  (3.2) Compute the best 1-tree obtainable from this spanning tree as follows. For each leaf v of the spanning tree we determine its second shortest incident edge. The length of the minimum spanning tree plus the length of this edge gives the length of the minimum 1-tree with v as fixed degree-2 node.
  (3.3) Define the vector d^k by $d_i^k = \delta_i - 2$ where δ_i is the degree of node i in the 1-tree computed in Step (3.2).
  (3.4) For every node i set $\lambda_i^{k+1} = \lambda_i^k + \alpha_k(0.7d_i^k + 0.3d_i^{k-1})$ (where $d_i^0 = 0$).
  (3.5) Set $\alpha_{k+1} = \gamma\alpha_k$ and increment k by 1.
(4) Return the best bound computed.

Differences to straightforward realizations are that the direction of the subgradient step is a convex combination of the current and the preceding subgradient, that the direction vector is not normalized, and that the special node for the 1-tree computations is not fixed. In theory the same optimal value of the Lagrange dual is attained whatever node is fixed. But practical experiments have shown that better bounds are obtained if additional time is spent for computing

various 1-trees. The chosen value of γ influences the running time of the method. The closer γ is to 1, the more iterations are performed and better bounds are obtained. Note that, since the step length is fixed a priori, there is no guarantee that each iteration step improves the bound.

Some authors propose to update the multipliers according to the formula

$$\lambda_i^{k+1} = \lambda_i^k + t^k(U - L(\lambda^k))\frac{d_i^k}{||d^k||}$$

where U is an estimate for the optimal solution value. For our set of sample problems, we found this formula inferior to the one above.

The running times are not very encouraging. Since edge weights are arbitrary we can compute the best tree in Step (3.1) only in time $O(n^2)$ and the best 1-tree in Step (3.2) in time $O(ln)$ where l is the number of leaves of the spanning tree. To obtain reasonable lower bounds more quickly, we use the following approach. The subgradient method is only performed on a sparse subgraph (e.g., for geometric instances a 10 nearest neighbor subgraph augmented by the Delaunay graph). This has the consequence that the computed bound may not be valid for the original problem. To get a valid lower bound, we compute a minimum 1-tree in the complete graph with respect to the node multipliers obtained with the subgradient method in the sparse graph.

Bounds obtained for problems *pcb442*, *rd783*, *pr1002*, and *pr2392* were 0.6%, 0.4%, 1.4%, and 1.3%, respectively, below the length of a shortest Hamiltonian cycle. The final iteration changed the bounds only slightly. In practical applications we can safely omit the final step and assume that the bound determined in the first phase is correct.

The subgradient method is not the only way for attacking nondifferentiable optimization problems occurring in the context of Lagrangean relaxation. A more elaborate approach is the so-called *bundle method* (Kiwiel (1989)). It is also based on subgradients, but in every iteration the new direction is computed as a convex combination of several (10–20) previous subgradients. Moreover, line searches are performed to determine the step lengths. In this sense our approach is a rather simple version of the bundle method keeping only a "bundle" of two subgradients (which are combined in a fixed way) and not performing line searches.

In Schramm (1989) an extension of this principle is discussed which combines the bundle approach with trust-region methods. From a theoretical point of view, it is interesting that finite convergence of this method to the optimal solution of the Lagrangean dual can be shown. Therefore, in the case of the TSP, this approach provides a finite procedure for computing the exact subtour elimination lower bound. But the running time is considerable, due to the many costly evaluations of $L(\lambda)$ in the line search.

6.3. LP relaxations

The first successful attempt to solve a "large" instance of the traveling salesman problem is reported in the seminal paper by Dantzig, Fulkerson and Johnson

(1954) who solved a 48-city instance. This paper is one of the cornerstones on which much of the methodology of using heuristics, linear programming and separation to attack combinatorial optimization problems is founded. It took a long time for the ideas of Dantzig, Fulkerson and Johnson to be pursued again, and this must probably be attributed to the fact that the systematic way of using cutting planes in integer programming, which had been put on a solid basis by the work of Gomory (1958, 1960, 1963), was not successful in practice. An important development is the systematic study of the traveling salesman polytope described in the previous section. Grötschel used the knowledge of the polytope to solve a 120 city instance to optimality, using IBM's linear programming package MPSX to optimize over relaxations of the traveling salesman polytope, visually inspecting the fractional solutions, adding violated facet defining inequalities, resolving etc., until the optimal solution was the incidence vector of a Hamiltonian cycle. He needed 13 iterations of this process (see Grötschel (1977, 1980)). Since the early eighties, more insight into the facial structure of the traveling salesman polytope and improved cutting plane based algorithms developed gradually.

On the computational side, the next steps were the papers Padberg and Hong (1980) and Crowder and Padberg (1980).

In the first paper, a primal cutting plane approach is used to obtain good bounds on the quality of solutions generated in the following way. An initial Hamiltonian cycle is determined by the Lin-Kernighan heuristic, and the first linear programming problem is given by (5.7) and (5.9) (i.e., by the fractional 2-matching relaxation). The initial basis corresponds to the initial Hamiltonian cycle. Then a pivoting variable is selected by the steepest edge criterion. If the adjacent basic solution after the pivot is the incidence vector of a Hamiltonian cycle, the pivot is carried out, and the algorithm proceeds with the new Hamiltonian cycle. Otherwise, one tries to identify a violated inequality which is satisfied with equality by the current solution but violated by the adjacent fractional solution. If such an inequality is found, it is appended to the current LP, a degenerate pivot is made on the selected pivoting variable, and the next pivoting variable is selected. Otherwise, the current (final) linear program is solved to optimality in order to obtain a lower bound on the length of the shortest Hamiltonian cycle. Out of 74 sample problems ranging from 15 to 318 cities, 54 problems could be solved to optimality in this way. The whole algorithm was written by the authors including an implementation of the simplex algorithm in rational arithmetic.

In the second paper, IBM's MPSX LP-package is used instead, and IBM's MPSX-MIP integer programming package is used to find the incidence vector of an optimal solution as follows. MIP is applied to the final LP to find an optimal integral solution. If this solution is the incidence vector of a Hamiltonian cycle, this cycle is returned as the optimal solution. Otherwise the solution is necessarily a collection of subtours, and the corresponding subtour elimination inequalities are appended to the integer program and the process is iterated. Thus for the first time a fully automatic computer program involving no human interaction was available to solve traveling salesman problems by heuristics, linear programming, separation and enumeration in the spirit of Dantzig, Fulkerson and Johnson.

Using their computer code, the authors were able to solve all 74 sample problems to optimality. The 318 city instance was solved in less than an hour of CPU time on an IBM 370/168 computer under the MVS operating system.

A similar, yet more sophisticated approach using MPSX/MIP is described in Grötschel and Holland (1991). They use a (dual) cutting plane procedure to obtain a tight linear programming relaxation. Then they proceed as Crowder and Padberg. An additional important enhancement is the use of sparse graphs, a prerequisite for attacking larger problem instances. Furthermore, improved separation routines are used, partly based on new results by Padberg and Rao (1982) on the separation of 2-matching inequalities as described in the previous section. The code was used to solve geometric instances with up to 666 nodes and random instances with up to 1000 nodes. Depending on parameter settings, the former took between 9 and 16 hours of CPU time, and the latter between 23 and 36 minutes of CPU time on an IBM 3081D under the operating system VM/CMS. Random problems where the edge weights are drawn from a uniform distribution appear to be much easier than geometric instances. From a software engineering point of view, the codes by Padberg and Hong, Crowder and Padberg, and Grötschel and Holland had the advantage that any general purpose branch and bound software for integer programming could be used to find integer solutions. However, if such an integer solution contained subtours, the corresponding subtour elimination inequalities were added to the LP-relaxation and the branch and bound part was started from scratch, again using a fixed linear programming relaxation in each node of the branch and bound tree.

On the other hand, the iterated "solving from scratch", whenever the addition of further subtour elimination inequalities was necessary, is a definite disadvantage. An even bigger drawback is the fact that the possibility of generating further globally valid cutting planes in non-root nodes of the branch and bound tree is not utilized. Furthermore, general purpose branch and bound software typically allows very little influence on the optimization process such as variable fixing based on structural properties of the problem. Such disadvantages are eliminated by the natural idea of applying the cutting plane algorithm with globally valid (preferably facet defining) inequalities in every node of the enumeration tree.

Such an approach was first published for the linear ordering problem by Grötschel, Jünger and Reinelt (1984). In Padberg and Rinaldi (1987) a similar approach was outlined for the TSP and called "branch and cut". By reporting the solution to optimality of three large unsolved problems of 532, 1002, and 2392 cities, it was shown for the first time in this paper how the new approach could be successfully used to solve instances of the traveling salesman problem that probably could not be solved with other available techniques.

The first state-of-the-art branch and cut algorithm for the traveling salesman problem is the algorithm published in Padberg and Rinaldi (1991). The major new features of the Padberg-Rinaldi algorithm are the branch and cut approach in conjunction with the use of column/row generation/deletion techniques, sophisticated separation procedures and an efficient use of the LP optimizer. The

LPs are solved using the packages XMP of Marsten (1981) on the DIGITAL computers microVAX II, VAX 8700 and VAX 780, as well as on the Control Data computer CYBER 205, and the experimental version of the code OSL by John Forrest of IBM Research on an IBM 3090/600 supercomputer. With the latter version of the code, the 2392-node instance is solved to optimality in about 4.3 hours of CPU time.

The rest of this section is devoted to a more detailed outline of such a state-of-the-art branch and cut algorithm for the TSP. Our description is based on the original implementation of Padberg and Rinaldi (1991) and a new implementation of Jünger, Reinelt and Thienel (1992). We use the terminology and the notation of the latter paper in which a new component is added to the Padberg-Rinaldi algorithm: a procedure that exploits the LP solution of the current relaxation to improve the current heuristic solution.

As in the original version of the algorithm, in the implementation described here, subtour elimination, 2-matching, comb and clique-tree inequalities are used as cutting planes. Since we use exact separation of subtour elimination inequalities, all integral LP solutions are incidence vectors of Hamiltonian cycles, as soon as no more subtour elimination inequalities are generated.

In our description, we proceed as follows. First we describe the enumerative part of the algorithm, i.e., we discuss in detail how branching and selection operations are implemented. Then we explain the work done in a subproblem of the enumeration. Finally we explain some important global data structures. There are two major ingredients of the processing of a subproblem, the computation of local lower and global upper bounds. The lower bounds are produced by performing an ordinary cutting plane algorithm for each subproblem. The upper bounds are obtained by exploiting fractional LP solutions in the construction of Hamiltonian cycles which are improved by heuristics.

The branch and cut algorithm for the TSP is outlined in the flowchart of Figure 19. Roughly speaking, the two leftmost columns describe the cutting plane phases within a single subproblem, the third column shows the preparation and execution of a branching operation, and in the rightmost column, the fathoming of a subproblem is performed. We give informal explanations of all steps of the flowchart.

Before going into detail, we have to define some terminology. Since in a branching step two new subproblems are generated, the set of all subproblems can be represented by a binary tree, which we call the *branch and cut tree*. Hence we call a subproblem also a *branch and cut node*. We distinguish between three different types of branch and cut nodes. The node which is currently processed is called the *current branch and cut node*. The other unfathomed leaves of the branch and cut tree are called the *active nodes*. These are the nodes which still must be processed. Finally, there are the already processed *nonactive nodes*.

The terms *edge* of a graph and *variable* in the integer programming formulation are used interchangeably, as they are in a one to one correspondence. Each variable (edge) has one of the following status values during the computation: *atlowerbound*, *basic*, *atupperbound*, *settolowerbound*, *settoupperbound*, *fixed-*

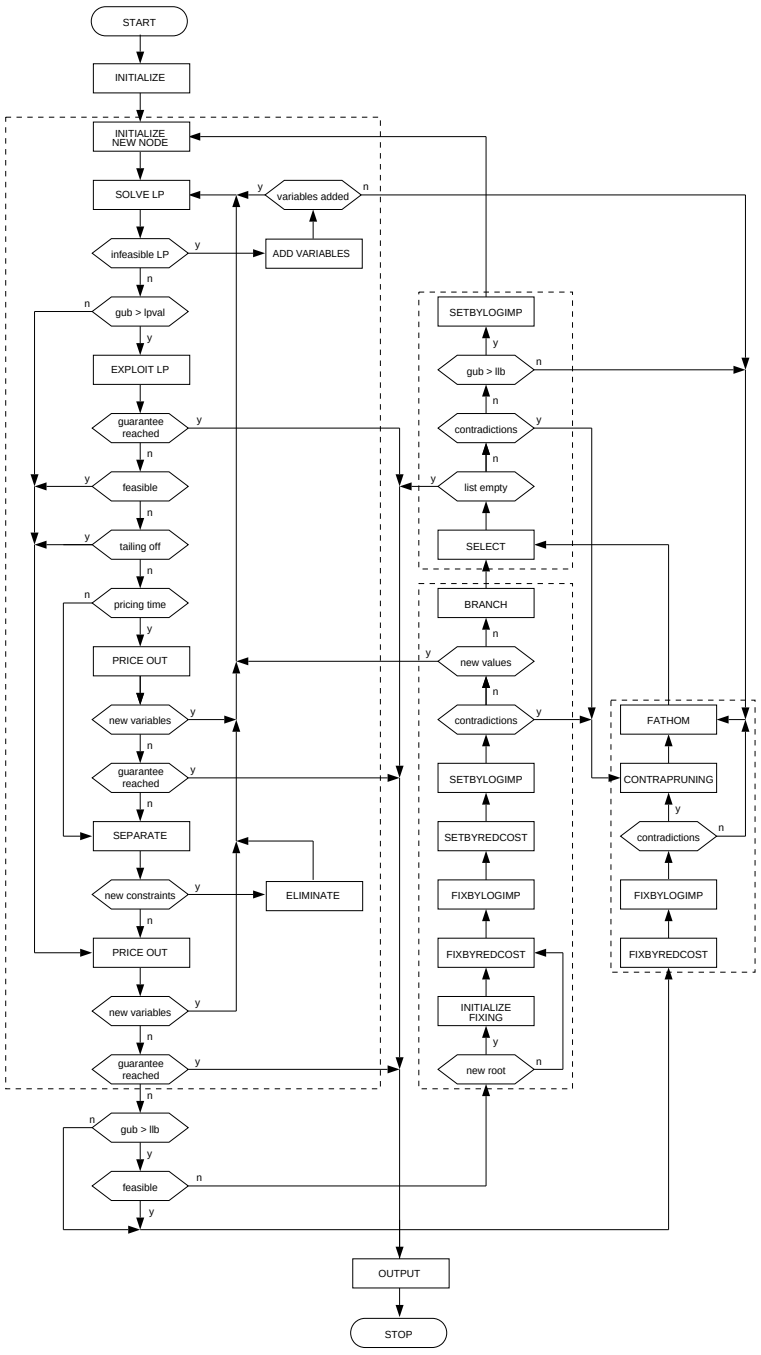

Figure 19. Flowchart of the branch and cut algorithm.

tolowerbound, *fixedtoupperbound*. When we say that a variable is *fixed* to zero or one, it means that it is at this value for the rest of the computation. If it is *set* to zero or one, this value remains valid only for the current branch and cut node and all branch and cut nodes in the subtree rooted at the current one in the branch and cut tree. The meanings of the other status values are obvious: As soon as an LP has been solved, each variable which has not been fixed or set receives one of the values *atlowerbound*, *basic* or *atupperbound* by the revised simplex method with lower and upper bounds.

The global variable *lpval* always denotes the optimal value of the last LP that has been solved, the global variable *llb* (local lower bound) is a lower bound for the currently processed node, the global variable *gub* (global upper bound) gives the value of the currently best known solution. The minimal lower bound of all active branch and cut nodes and the current branch and cut node is the global lower bound *glb* for the whole problem, whereas the global variable *rootlb* is the lower bound found while processing the root node of the remaining branch and cut tree. As we will see later, *lpval* and *llb* may differ, because we use sparse graph techniques, i.e., the computation of the lower bounds is processed only on a small subset of the edges and only those edges are added which are necessary to guarantee the validity of the bounds on the complete graph.

By the *root of the remaining branch and cut tree* we denote the highest common ancestor in the branch and cut tree of all branch and cut nodes which still must be processed. The values of *gub* and *glb* can be used to terminate the computation as soon as the guarantee requirement is satisfied. As in branch and bound terminology we call a subproblem *fathomed*, if the local lower bound *llb* of this subproblem is greater than or equal to the global upper bound *gub* or the subproblem becomes infeasible (e.g., branching variables have been set in a way that the graph does not contain a Hamiltonian cycle). Following TSPLIB (Reinelt (1991a, 1991b)) all distances are integers. So all terms of the computation which express a lower bound may be rounded up, e.g., one can fathom a node with global upper bound *gub* and local lower bound *llb*, if $\lceil llb \rceil \geqslant gub$. Since this is only correct for the distances defined in TSPLIB we neither outline this feature in the flowchart nor in the following explanations.

The algorithm consists of three different parts: The enumerative frame, the computation of upper bounds and the computation of lower bounds. It is easy to identify the boxes of the flowchart of Figure 18 with the dashed boxes of the flowchart of Figure 19. The upper bounding is done in EXPLOIT LP, the lower bounding in all other parts of the dashed bounding box. There are three possibilities to enter the bounding part and three to leave it. Normally we perform the bounding part after the startup phase in INITIALIZE or the selection of a new subproblem in SELECT. Furthermore it is advantageous, although not necessary for the correctness of the algorithm, to reenter the bounding part if variables are fixed or set to new values by FIXBYLOGIMP or SETBYLOGIMP, instead of creating two new subproblems in BRANCH. Normally, the bounding part is left if no variables are added by PRICE OUT. In this case we know that the bounds for the just processed subproblem are valid for the complete graph.

Sometimes an infeasible subproblem can be detected in the bounding part. This is the second way to leave the bounding part after ADD VARIABLES. We also stop the computations of bounds and output the currently best known solution, if our guarantee requirement is satisfied (*guarantee reached*), but we ignore this, if we want to find the optimal solution.

6.4. Enumerative frame

In this paragraph we explain the implementation of the implicit enumeration. Nearly all parts of this enumerative frame are not TSP specific. Hence it is easy to adapt it to other combinatorial optimization problems.

INITIALIZE

The problem data is read. We distinguish between several problem types as defined in Reinelt (1991a, 1991b) for the specifications of TSPLIB data. In the simplest case, all edge weights are given explicitly in the form of a triangular matrix. In this case very large problems are prohibitive because of the storage requirements for the problem data. But very large instances are usually generated by some algorithmic procedure, which we utilize. The most common case is the metric TSP instance, in which the nodes defining the problem correspond to points in d-dimensional space and the distance between two nodes is given by some metric distance between the respective points. Therefore, distances can be computed as needed in the algorithm and we make use of this fact in many cases.

In practical experiments it has been observed that most of the edges of an optimal Hamiltonian cycle connect near neighbors. Often, optimal solutions are contained in the 10-nearest neighbor subgraph of K_n. In any case, a very large fraction of the edges contained in an optimal Hamiltonian cycle are already contained in the 5-nearest neighbor subgraph of K_n. Depending on two parameters k_s and k_r we compute the k_s-nearest neighbor subgraph and augment it by the edges of a Hamiltonian cycle found by a simple heuristic so that the resulting *sparse graph* $G = (V, E)$ is Hamiltonian. Using this solution, we can also initialize the value of the global upper bound *gub*. We also compute a list of edges which have to be added to E to contain the k_r-nearest neighbor subgraph. These edges form the *reserve graph*, which is used in PRICE OUT and ADD VARIABLES. We will start working on G, adding and deleting edges (variables) dynamically during the optimization process. We refer to the edges in G as *active* edges and to the other edges as *nonactive* edges. All global variables are initialized. The set of active branch and cut nodes is initialized as the empty set. Afterwards the root node of the complete branch and cut tree is processed by the bounding part.

BOUNDING

The computation of the lower and upper bounds will be outlined in Subsection 6.5. We continue the explanation of the enumerative frame at the ordinary exit of the bounding part (at the end of the first column of the dashed bounding

box). In this case it is guaranteed that the lower bound on the sparse graph *lpval* becomes a local lower bound *llb* for the subproblem on the complete graph.

Since we use exact separation of subtour elimination inequalities, all integral LP solutions are incidence vectors of Hamiltonian cycles, as soon as no more subtour elimination inequalities are generated.

We check if the current branch and cut node cannot contain a better solution than the currently best known one ($gub \leqslant llb$). If this is the case, the current branch and cut node can be fathomed (rightmost column of the flowchart), and if no further branch and cut nodes have to be considered, the currently best known solution must be optimal (*list empty* after SELECT). Otherwise we have to check if the current LP-solution is already a Hamiltonian cycle. If this is the case (*feasible*) we can fathom the node (possibly giving a new value to *gub*), otherwise we prepare a branching operation and the selection of another branch and cut node for further processing (third column of the flowchart).

INITIALIZE FIXING, FIXBYREDCOST

If we are preparing a branching operation, and the current branch and cut node is the root node of the currently remaining branch and cut tree, the reduced cost of the nonbasic active variables can be used to fix them forever at their current values. Namely, if for an edge e the variable x_e is nonbasic and the reduced cost is r_e, we can fix x_e to zero if $x_e = 0$ and $rootlb + r_e > gub$ and we can fix x_e to one if $x_e = 1$ and $rootlb - r_e > gub$.

During the computational process, the value of *gub* decreases, so that at some later point in the computation, one of these criteria can be satisfied, even though it is not satisfied at the current point of the computation. Therefore, each time when we get a new root of the remaining branch and cut tree, we make a *list of candidates for fixing* of all nonbasic active variables along with their values (0 or 1) and their reduced costs and update *rootlb*. Since storing these lists in every node, which might eventually become the root node of the remaining active nodes in the branch and cut tree, would use too much memory space, we process the complete bounding part a second time for the node, when it becomes the new root. If we could initialize the constraint system for the recomputation by those constraints, which were present in the last LP of the first processing of this node, we would need only a single call of the simplex algorithm. However, this would require too much memory. So we initialize the constraint system with the constraints of the last solved LP. As some facets are separated heuristically, it is not guaranteed that we can achieve the same local lower bound as in the previous bounding phase. Therefore we not only have to use the reduced costs and status values of the variables of this recomputation, but also the corresponding local lower bound as *rootlb* in the subsequent calls of the routine FIXBYREDCOST.

If we initialize the basis by the variables contained in the best known Hamiltonian cycle and call the primal simplex algorithm, we can avoid phase 1 of the simplex method. Of course this recomputation is not necessary for the root of the complete branch and cut tree, i.e., the first processed node. The list of candidates for fixing is checked by the routine FIXBYREDCOST whenever it has

been freshly compiled or the value of the global upper bound *gub* has improved since the last call of FIXBYREDCOST.

FIXBYREDCOST may find that a variable can be fixed to a value opposite to the one it has been set to (*contradiction*). This means that earlier in the computation, somewhere on the path of the current branch and cut node to the root of the branch and cut tree, we have made an unfavorable decision which led to this setting either directly in a branching operation or indirectly via SETBYREDCOST or SETBYLOGIMP (to be discussed below). Contradictions are handled by CONTRAPRUNING, whenever FIXBYREDCOST has set *contradiction* to *true* using such a condition.

Before starting a branching operation and if no contradiction has occurred, some fractional (basic) variables may have been fixed to new values (0 or 1). In this case we solve the new LP rather than performing the branching operation.

FIXBYLOGIMP

After variables have been fixed by FIXBYREDCOST, we call FIXBYLOGIMP. This routine tries to fix more variables by logical implication as follows: If two edges incident to a node v have been fixed to 1, all other edges incident to v can be fixed to 0 (if not fixed already). As in FIXBYREDCOST, contradictions to previous variable settings may occur. Upon this condition the variable *contradiction* is set to *true*. If variables are fixed to new values, we proceed as explained in FIXBYREDCOST.

In principle also fixing or setting variables to zero could have logical implications. If all incident edges of a node but two are fixed or set to zero, these two edges can be fixed or set to one. However, as we work on sparse graphs, this occurs quite rarely so that we omit this check.

SETBYREDCOST

While fixings of variables are globally valid for the whole computation, variable settings are only valid for the current branch and cut node and all branch and cut nodes in the subtree rooted at the current branch and cut node. SETBYREDCOST sets variables by the same criteria as FIXBYREDCOST, but based on the local reduced cost and the local lower bound *llb* of the current subproblem rather than "globally valid reduced cost" and the lower bound of the root node *rootlb*. Contradictions are possible if in the meantime the variable has been fixed to the opposite value. In this case we go to CONTRAPRUNING. The variable settings are associated with the current branch and cut node, so that they can be undone when necessary. All set variables are inserted together with the branch and cut node into the hash table of the set variables, which is explained in Subsection 6.6.

SETBYLOGIMP

This routine is called whenever SETBYREDCOST has successfully fixed variables, as well as after a SELECT operation. It tries to set more variables by logical implication as follows: If two edges incident to a node v have been set or fixed to 1, all other edges incident to v can be set to 0 (if not fixed already). As in

SETBYREDCOST, all settings are associated with the current branch and cut node. If variables are set to new values, we proceed as explained in FIXBYREDCOST. As in SETBYREDCOST, the set variables are stored in a hash table, see Subsection 6.6.

After the selection of a new node in SELECT, we check if the branching variable of the father is set to 1 for the selected node. If this is the case, SETBYLOGIMP may also set additional variables.

BRANCH

Some fractional variable is chosen as the branching variable and, accordingly, two new branch and cut nodes, which are the two sons of the current branch and cut node, are created and added to the set of active branch and cut nodes. In the first son the branching variable is set to 1 in the second one to 0. These settings are also registered in the hash table.

SELECT

A branch and cut node is selected and removed from the set of active branch and cut nodes. Our strategy is to select the candidate with the minimal local lower bound, a variant of the "best first search" strategy which compares favorably with commonly used strategies such as "depth first search" or "breadth first search". If the list of active branch and cut nodes is empty, we can conclude optimality of the best known Hamiltonian cycle. Otherwise we start processing the selected node. After a successful selection, variable settings have to be adjusted according to the information stored in the branch and cut tree. If it turns out that some variable must be set to 0 or 1, yet has been fixed to the opposite value in the meantime, we have a contradiction similar as discussed above. In this case we prune the branch and cut tree accordingly by going to CONTRAPRUNING and fathom the node in FATHOM. If the local lower bound *llb* of the selected node is greater than or equal to the global upper bound *gub*, we fathom the node immediately and continue the selection process. A branch and cut node has pointers to its father and its two sons. So it is sufficient to store a set variable only once in any path from the root to a leaf in the branch and cut tree. If we select a new problem, i.e., proceed with the computation at some leaf of the tree, we only have to determine the highest common ancestor of the old node and the new leaf, reset the set variables on the path from the old node to the common ancestor and set the variables on the path from the common ancestor to the new leaf.

CONTRAPRUNING

Not only the current branch and cut node, where we have found the contradiction, can be deleted from further consideration, but all active nodes with the same "wrong" setting can be fathomed. Let the variable with the contradiction be e. Via the hash table of the set variables we can efficiently determine all branch and cut nodes where e has been set. If in a branch and cut node b the variable e is set to the "wrong" bound we remove all active nodes (unfathomed leaves) in the subtree below b from the set of active nodes.

FATHOM

If for a node the global upper bound *gub* does not exceed the local lower bound *llb*, or a contradiction occurred, or an infeasible branch and cut node has been generated, the current branch and cut node is deleted from further consideration. Even though a node is fathomed, the global upper bound *gub* may have changed during the last iteration, so that additional variables may be fixed by FIXBYREDCOST and FIXBYLOGIMP. The fathoming of nodes in FATHOM and CONTRAPRUNING may lead to a new root of the branch and cut tree for the remaining active nodes.

OUTPUT

The currently best known Hamiltonian cycle, which is either optimal or satisfies the desired guarantee requirement, is written to an output file.

6.5. Computation of lower and upper bounds

The computation of lower bounds consists of all elements of the dashed bounding box except EXPLOIT LP, where the upper bounds are computed.

During the whole computation, we keep a *pool* of active and nonactive facet defining inequalities of the traveling salesman polytope. The *active inequalities* are the ones in the current LP and are both stored in the pool and in the constraint matrix, whereas the *inactive inequalities* are only present in the pool. An inequality becomes inactive, if it is nonbinding in the last LP solution. When required, it is easily regenerated from the pool and made active again later in the computation. The pool is initially empty. If an inequality is generated by a separation algorithm, it is stored both in the pool and added to the constraint matrix.

INITIALIZE NEW NODE

Let A_G be the node-edge incidence matrix corresponding to the sparse graph G. If the node is the root node of the branch and cut tree the LP is initialized to

$$\begin{aligned} \min \quad & cx \\ \text{s.t.} \quad & A_G x = \mathbf{2} \\ & \mathbf{0} \leqslant x \leqslant \mathbf{1} \end{aligned}$$

and the feasible basis obtained from the initial Hamiltonian cycle is used as a starting basis. In subsequent subproblems, we initialize the constraint matrix by the equations induced by the node-edge incidence matrix of the sparse graph and by the inequalities which were active when the last LP of the father of the branch and cut node was solved. These inequalities can be regenerated from the pool. Since the final basis of the father is dual feasible for the initial LP of its sons, we start with this basis to avoid phase 1 of the simplex method. Every column of a nonbasic set or fixed variable is removed from the constraint matrix and if its status value is *settoupperbound* or *fixedtoupperbound*, the right hand side of

every constraint containing it has to be adjusted. The corresponding coefficient of the objective function must be added to the optimal value returned by the simplex algorithm in order to get the correct value of the variable *lpval*. Set or fixed basic variables are not deleted, because this would lead to an infeasible basis and require phase 1 of the simplex method. We perform the adjustment of these variables by adapting their upper and lower bounds.

SOLVE LP

The LP is solved, either by the two phase primal simplex method, if the current basis is neither primal nor dual feasible, by the primal simplex method, if the basis is primal feasible (e.g., if variables have been added) or by the dual simplex method if the basis is dual feasible (e.g., if constraints have been added or more variables have been set). As LP solver we use CPLEX by R.E. Bixby (see CPLEX (1993)).

If the LP has no feasible solution we go to ADD VARIABLES, otherwise we proceed downward in the flowchart.

ADD VARIABLES

Variables have to be added to the sparse graph if indicated by the reduced costs (handled by PRICE OUT) or if the current LP is infeasible. The latter may be caused by three reasons. First, equations may not be satisfiable because the variables associated with all but at most one edge incident to a node v in the sparse graph may be fixed or set to 0. Such an infeasibility can either be removed by adding an additional edge incident to v, or, if all edges are present already, we can fathom the branch and cut node.

Second, suppose that all equations are satisfiable, yet some active inequality has a void left hand side, since all involved variables are fixed or set, but is violated. As is clear from our strategy for variable fixings and settings, this also means that the branch and cut node is fathomed, since all constraint coefficients are nonnegative in our implementation.

Finally, neither of the above conditions may apply, and the infeasibility is detected by the LP solver. In this case we perform a pricing step in order to find out if the dual feasible LP solution is dual feasible for the entire problem. We check for variables that are not in the current sparse graph (i.e., are assumed to be at their lower bound 0) and have negative reduced cost. Such variables are added to the current sparse graph. An efficient way of the computation of the reduced costs is outlined in PRICE OUT.

If variables have been added, we solve the new LP. Otherwise, we try to make the LP feasible by a more sophisticated method. The LP value *lpval*, which is the objective function value corresponding to the dual feasible basis where primal infeasibility is detected, is a lower bound for the objective function value obtainable in the current branch and cut node. So if $lpval \geqslant gub$, we can fathom the branch and cut node.

Otherwise, we try to add variables that may restore feasibility. First we mark all infeasible variables, including negative slack variables.

Let e be a nonactive variable and r_e be the reduced cost of e. We take e as a candidate only if $lpval + r_e \leqslant gub$. Let B be the basis matrix corresponding to the dual feasible LP solution, at which the primal infeasibility was detected. For each candidate e let a_e be the column of the constraint matrix corresponding to e and solve the system $B\overline{a}_e = a_e$. Let $\overline{a}_e(b)$ be the component of $\overline{a}_e$ corresponding to basic variable x_b. Increasing x_e reduces some infeasibility if one of the following holds.

- x_b is a structural variable (i.e., corresponding to an edge of G) and
  $x_b < 0$ and $\overline{a}_e(b) < 0$
  or
  $x_b > 1$ and $\overline{a}_e(b) > 0$
- x_b is a slack variable and
  $x_b < 0$ and $\overline{a}_e(b) < 0$.

In such a case we add e to the set of active variables and remove the marks from all infeasible variables whose infeasibility can be reduced by increasing x_e. We do this in the same hierarchical fashion as in the the procedure PRICE OUT that is described below.

If variables can be added, we regenerate the constraint structure and solve the new LP, otherwise we fathom the branch and cut node. Note that all systems of linear equations that have to be solved have the same matrix B, and only the right hand side a_e changes. We utilize this by computing a factorization of B only once, in fact, the factorization can be obtained from the LP solver for free. For further details on this algorithm, see Padberg and Rinaldi (1991).

EXPLOIT LP

We check if the current LP solution is the incidence vector of a Hamiltonian cycle. If this is the case, the variable *feasible* is set to *true*. Otherwise, the LP solution is exploited in the construction of a Hamiltonian cycle. To this end we use the following heuristic. Edges are sorted according to decreasing values in the current LP solution. This list is scanned and edges become part of the Hamiltonian cycle if they do not produce a subtour. Then the savings heuristic as described in Section 4 is used to combine the produced system of paths to form a Hamiltonian cycle. Then the Lin-Kernighan heuristic is applied. If the final solution has smaller cost than the currently best known one, it is made the incumbent solution, *upperbound* is updated and *improved* is set to *true*. For details of this step, see Jünger, Reinelt and Thienel (1992).

SEPARATE

This part implements separation for the TSP as described in the previous section. In a first phase, the pool is checked for inactive violated inequalities. If an inactive inequality is violated, it is added to the active set of constraints. While checking the pool, we remove, under certain conditions, all those inequalities from the pool which have been inactive for a long time. If violated inequalities have been added from the pool, we terminate the separation phase. Otherwise, we try to identify new violated constraints as outlined in the previous section,

store them as active inequalities in the pool and add them to the LP. For details of the separation process, we refer to the original articles mentioned in Section 5.

ELIMINATE

Before the LP is solved after a successful cutting plane generation phase, all active inequalities which are nonbinding in the current LP solution are eliminated from the constraint structure and marked inactive in the pool. We can safely do this to keep the constraint structure as small as possible, because as soon as the inequality becomes violated in a later cutting plane generation phase, it can be generated anew from the pool (if it has not been removed in the meantime).

PRICE OUT

Pricing is necessary before a branch and cut node can be fathomed. Its purpose is to check if the LP solution computed on the sparse graph is valid for the complete graph, i.e., all nonactive variables "price out" correctly. If this is not the case, nonactive variables with negative reduced cost are added to the sparse graph and the new LP is solved using the primal simplex method starting with the previous (now primal feasible) basis, otherwise we can update the local lower bound *llb* and possibly the global lower bound *glb*. If the global lower bound has changed, our guarantee requirement might be satisfied and we can stop the computation after the output of the currently best known Hamiltonian cycle.

Although the correctness of the algorithm does not require this, we perform additional pricing steps every k solved LPs (see Padberg and Rinaldi (1991)). The effect is that nonactive variables which are required in a good or optimal Hamiltonian cycle tend to be added to the sparse graph early in the computation.

In a first phase, only the variables in the reserve graph are considered. If the "partial pricing" considering only the edges of the reserve graph has not added variables, we have to check the reduced costs of all nonactive variables which takes a lot of computational effort. But this second step of PRICE OUT can be processed more efficiently. If our current branch and cut node is the root of the remaining branch and cut tree, we can check if the reduced cost r_e of a nonactive variable e satisfies the relation $lpval + r_e > gub$. In this case we can discard this nonactive candidate edge forever. During the systematic enumeration of all edges of the complete graph, we can make an explicit list of those edges which remain possible candidates. In the early steps of the computation, too many such edges remain, so that we cannot store this list completely with reasonable memory consumption. Rather, we predetermine a reasonably sized buffer and mark the point where the systematic enumeration has to be resumed after considering the edges in the buffer. In later steps of the computation there is a good chance that the complete list fits into the buffer, so that later calls of the pricing routine become much faster than early ones.

To process PRICE OUT efficiently, for each node v a list of those constraints containing v is made. Whenever an edge $e = vw$ is considered, we initialize the reduced cost by c_e, then v's and w's constraint lists are compared, and the value of the dual variable y_f times the corresponding coefficient is subtracted from the

reduced cost whenever the two lists agree in a constraint f. The format of the pool, which is explained in Subsection 6.6, provides us with an efficient way to compute the constraint lists and the coefficients.

6.6. Data structures

A suitable choice of data structures is essential for an efficient implementation of a branch and cut algorithm. This issue is discussed in detail in Jünger, Reinelt and Thienel (1992).

Sparse graphs

In INITIALIZE we select only a very small subset of the edges for our computations: the set of active edges, which remains small during the computations. For the representation of the resulting sparse graph we choose a data structure which saves memory and enables us to efficiently perform the operations scanning all incident edges of a node, scanning all adjacent nodes of a node, determining the endnodes of an edge and adding an edge to the sparse graph.

Branch and cut nodes

Although a subproblem is completely defined by the fixed variables and the variables that are set temporarily, it is necessary to store additional information at each node for an efficient implementation. Every branch and cut node has pointers to its father and sons. A branch and cut node contains the arrays *set* of its set variables and *setstat* with the corresponding status values (*settolowerbound, settoupperbound*). The first variable in this array is the branching variable of the father. There may be further entries to be made in case of successful calls of SETBYREDCOST and SETBYLOGIMP while the node is processed. The set variables of a branch and cut node are all the variables in the arrays *set* of all nodes in the path from the root to the node.

In a branch and cut node we store the local lower bound of the corresponding subproblem. After creation of a new leaf of the tree in BRANCH this is the bound of its father, but after processing the node we can in general improve the bound and update this value.

Of course it would be correct to initialize the constraint system of the first LP of a new selected node with the inequalities of the last processed node, since all generated constraints are facets of STSP. However, this would lead to tedious recomputations, and it is not guaranteed that we can regenerate all heuristically separated inequalities. So it is preferable to store in each branch and cut node pointers to those constraints in the pool, which are in the constraint matrix of the last solved LP of the node. We initialize with these constraints the first LP of each son of that node.

As we use an implementation of the simplex method to solve the linear programs, we store the basis of the last processed LP of each node, i.e., the status values of the variables and the constraints. Therefore we can avoid phase 1 of the simplex algorithm, if we carefully restore the LP of the father and solve this first LP with the dual simplex method. Since the last LP of the father and the

first LP of the son differ only by the set branching variable, variables set by SETBYLOGIMP, and variables that have been fixed in the meantime, the basis of the father is dual feasible for the first LP of the son.

Active nodes

In SELECT a node is extracted from the set of active nodes for further processing. Every selection strategy defines an order on the active nodes. The minimal node is the next selected one. The representing data structure must allow efficient implementations of the operations *insert*, *extractmin* and *delete*. The operation *insert* is used after creation of two new branch and cut nodes in BRANCH, *extractmin* is necessary to select the next node in SELECT and *delete* is called if we remove an arbitrary node from the set of active nodes in CONTRAPRUNING. These operations are very well supported by a height balanced binary search tree. We have implemented a red-black tree (Bayer (1972), Guibas and Sedgewick (1978), see also Cormen, Leiserson and Rivest (1989)) which provides $O(\log m)$ running time for these operations, if the tree consists of m nodes. Each node of the red-black tree contains a pointer to the corresponding leaf of the branch and cut tree and vice versa.

Temporarily set variables

A variable is either set if it is the branching variable or it is set by SETBYREDCOST or SETBYLOGIMP. In CONTRAPRUNING it is essential to determine efficiently all nodes where a certain variable is set. To avoid scanning the complete branch and cut tree, we apply a hash function to a variable right after setting and store in the slot of the hash table the set variable and a pointer to the corresponding branch and cut node. So it is quick and easy to find all nodes with the same setting by applying an appropriate hashing technique. We have implemented a Fibonacci hash with chaining (see Knuth (1973)).

Constraint pool

The data structure for the pool is very critical concerning running time and memory requirements. It is not appropriate to store a constraint in the pool just as the corresponding row of the constraint matrix, because we also have to know the coefficients of variables which are not active. This is necessary in PRICE OUT, to avoid recomputation from scratch after addition of variables and in INITIALIZE NEW NODE. Such a format would require too much memory. We use a node oriented sparse format. The pool is represented by an array. Each component (constraint) of the pool is again an array, which is allocated dynamically with the required size. This last feature is important, because the required size for a constraint of STSP(n) can range from four entries for a subtour elimination constraint to about $2n$ entries for a comb or a clique-tree inequality.

A subtour elimination inequality is defined by the node set $W = \{w_1, \ldots, w_l\}$. It is sufficient to store the size of this node set and a list of the nodes.

2-matching inequalities, comb inequalities and clique-tree inequalities are defined by a set of handles $\mathcal{H} = \{H_1, \ldots, H_r\}$ and a set of teeth $\mathcal{T} = \{T_1, \ldots, T_k\}$,

with the sets $H_i = \{h_{i_1}, \ldots, h_{i_{n_i}}\}$ and $T_j = \{t_{j_1}, \ldots, t_{j_{m_j}}\}$. In our pool format a clique-tree inequality with h handles and t teeth is stored as:

$$r,n_1,h_{1_1},\ldots,h_{1_{n_1}},\ldots,n_r,h_{r_1}\ldots,h_{r_{n_r}},k,m_1,t_{1_1},\ldots,t_{1_{m_1}},\ldots,m_k,t_{k_1}\ldots,t_{k_{m_k}}$$

For each constraint in the pool, we also store its storage type (subtour or clique-tree).

This storage format of a pool constraint provides us with an easy method to compute the coefficient of every involved edge, even if it is not present in the sparse graph at generation time. In case of a subtour elimination inequality, the coefficient of an edge is 1 if both endnodes of the edge belong to W, otherwise it is zero. The computation of the coefficients of other constraints is straightforward. A coefficient of an edge of a 2-matching inequality is 1 if both endnodes of the edge belong to the handle or to the same tooth, 0 otherwise. Some more care is needed for comb inequalities and clique-tree inequalities. The coefficient of an edge is 2 if both endnodes belong to the same intersection of a handle and a tooth, 1 if both endnodes belong to the same handle or (exclusive) to the same tooth and 0 in all other cases.

Since the pool is the data structure using up the largest amount of memory, only those inactive constraints are kept in the pool, which have been active, when the last LP of the father of at least one active node has been solved. These inequalities are used to initialize the first LP of a new selected node. In the current implementation the maximal number of constraints in the pool is $50n$ for TSP(n). After each selection of a new node we try to eliminate those constraints from the pool which are neither active at the current branch and cut node nor necessary to initialize the first LP of an active node. If, nevertheless, more constraints are generated than free slots of the pool are available, we remove nonactive constraints from the pool. But now we cannot restore the complete LP of the father of an active node. In this case we proceed as in INITIALIZE FIXING to initialize the constraint matrix and to get a feasible basis.

7. Computation

Computational experience with the algorithmic techniques presented in the previous sections has been given along with the algorithms in Section 4 and parts of Sections 5 and 6. In this final section, we would like to report on computational results of linear programming based algorithms, in particular, the branch and cut algorithm, for various kinds of problem instances. We report both on optimal and provably good solutions.

7.1. Optimal solutions

For most instances of moderate size arising in practice, optimal solutions can indeed be found with the branch and cut technique. On the other hand, there are small instances that have not been solved yet.

Some Euclidean instances from TSPLIB

Computational results for a branch and cut algorithm for solving symmetric traveling salesman problems to optimality have been published by Padberg and Rinaldi (1991). In order to have a common basis for comparison, the performance of their algorithm on a SUN SPARCstation 10/20 for our standard set of test problems defined in Section 4 is presented in Table 7. For each instance we show the number of nodes of the tree (not including the root node), the total number of distinct cuts generated by the separation algorithm, the maximum cardinality of the set of active constraints (including the degree equations), the maximum cardinality of the set of active edges, the number of times the LP solver is executed, the percentage of time spent in computing and improving a heuristic solution, the percentage of time spent by the LP solver, and the overall computation time in seconds.

All the problem instances have been solved with the same setting for the parameters that can be used to tune the algorithm. Tailoring parameters for each instance individually often gives better results. E.g., with a different setting the instance *pr2392* is solved without branching. The fact that all instances of Table 7 are Euclidean is not exploited in the implementation. For other computational results, in particular for non Euclidean instances, see Padberg and Rinaldi (1991).

Further computational results for branch and cut algorithms for solving TSP to optimality have been reported in Jünger, Reinelt and Thienel (1992) and Clochard and Naddef (1993).

It has been announced by Applegate, Bixby, Chvátal and Cook (1992) that several problem instances from TSPLIB have been solved with their branch and cut implementation, the largest being *fnl4461*, whose computation time was the equivalent of about 1.9 years of SUN SPARCstation 10. However, at the time of writing these results are not yet published.

We do not know of any algorithmic approach other than the polyhedral branch and cut method which is able to solve even moderately sized instances from TSPLIB to optimality.

From the results presented above one may get the erroneous impression that todays algorithmic knowledge is sufficient to solve instances with up to a few thousand cities to optimality. Unfortunately, there are small instances that cannot be solved to optimality in a reasonable amount of time. See, for example, some non Euclidean instances described below. This is not surprising at all since the TSP is an $\mathcal{NP}$-hard combinatorial optimization problem. Still the impression might remain that Euclidean instances of size up to, say, 1000 nodes can be solved routinely to optimality. Also this impression is wrong.

Some difficult Euclidean instances

Already from a quick look of Table 7 it is clear that, unlike in the case of the computation of heuristic solutions, there is a weak correlation between the computational effort and the instance size. Two small Euclidean instances from TSPLIB are not listed in Table 7, namely *pr76* and *ts225*. With the same implementation as used for the results of Table 7, solving *pr76* takes about 405 seconds

Table 7

Computation of optimal solutions

Problem	BC	Cuts	Mrow	Mcol	Nlp	% Heu	% LP	Time
lin105	0	50	137	301	10	89.4	8.5	11
pr107	0	111	148	452	19	73.8	23.8	10
pr124	2	421	199	588	74	74.1	16.7	77
pr136	10	786	217	311	102	70.8	14.8	101
pr144	2	273	238	1043	52	71.3	25.3	43
pr152	20	1946	287	2402	371	37.9	44.5	303
u159	0	139	210	395	23	82.2	15.1	17
rat195	16	1730	318	483	217	49.7	26.2	463
d198	2	563	311	1355	66	79.3	13.7	129
pr226	0	296	344	3184	31	72.4	25.9	87
gil262	4	950	409	668	90	74.6	16.1	197
pr264	0	70	305	1246	17	82.5	15.8	47
pr299	18	3387	554	800	281	1.0	89.5	2394
lin318	4	1124	497	875	100	56.8	20.1	344
rd400	54	8474	633	1118	852	36.5	45.1	2511
pr439	92	10427	741	1538	1150	26.5	55.0	3278
pcb442	50	2240	608	895	486	52.4	31.0	530
d493	70	20291	845	1199	1105	13.9	27.8	7578
u574	2	2424	910	1588	140	42.0	30.9	1134
rat575	110	24185	851	1455	1652	21.2	40.1	7666
p654	2	969	870	2833	55	59.7	35.1	449
d657	220	67224	1056	2154	3789	2.1	41.4	37642
u724	40	14146	1112	1962	766	4.5	32.8	9912
rat783	6	2239	1097	1953	126	64.8	25.6	1039
pr1002	20	14713	1605	2781	572	4.0	43.5	18766
pcb1173	324	165276	1686	3362	5953	6.7	61.7	91422
rl1304	46	38772	2101	5305	1377	2.0	84.5	160098
nrw1379	614	226518	1942	3643	7739	7.4	39.0	155221
u1432	2	4996	2044	2956	96	33.6	53.5	1982
pr2392	2	11301	3553	6266	145	23.6	57.9	7056

and 92 nodes of the tree. As far as we know, no algorithm has found a certified optimal solution to *ts225* yet. We report on the computation of a quality guaranteed solution for this problem in Subsection 7.2. Clochard and Naddef (1993) observe that both these problems have the same special structure that might be the reason for the poor performance of branch and cut algorithms. They propose a possible explanation for why these problems are difficult and describe a generator that produces random Euclidean instances with the same structural property. Applying new separation heuristics for path inequalities combined with an elaborate branching strategy they obtained very encouraging results for the hard instance *pr76*.

Some difficult non Euclidean instances

It is actually not very difficult to create artificially hard instances for a branch and cut algorithm. As an example, take as the objective function of an instance a facet defining inequality for the TSP polytope that is not included in the list of inequalities that the separation procedure can produce. To give numerical examples, we considered the crown inequality described in Section 5. Table 8 shows the computational results for a few instances of this type. The names of these instances have the prefix *cro*.

Another kind of instances that are expected to be difficult are those that arise from testing if a graph is Hamiltonian. To provide difficult numerical examples, we considered some hypohamiltonian graphs that generalize the Petersen graph. A graph is *hypohamiltonian* if it is not Hamiltonian but the removal of any node makes it Hamiltonian. We applied the transformation described in Section 2 to make the tests. The results are also listed in Table 8. The instance names have the prefix *NH*. Finally, we added one edge to each graph considered before that makes it Hamiltonian and we ran the test once more. In this case the computation was very fast as can be seen in Table 8, where the modified instances appear with the prefix *H*.

Table 8

Optimal solutions of non Euclidean instances

Problem	BC	Cuts	Nlp	% Heu	% LP	Time
cro12	38	57	83	9.7	41.0	2
cro16	204	277	390	6.0	40.9	16
cro20	1078	1657	1838	4.3	39.1	113
cro24	4064	10323	8739	3.5	32.8	1232
cro28	19996	182028	68010	4.8	21.1	8864
NH58	40	287	276	10.0	55.5	28
NH82	58	489	505	6.3	62.5	67
NH196	294	2800	2817	1.1	69.0	2168
H58	0	0	1	0.0	100.0	2
H82	0	0	1	0.0	100.0	4
H196	0	0	1	0.0	100.0	25

Randomly generated instances

It is common in the literature that the performance of algorithms is evaluated on randomly generated problem instances. This is often due to the fact that real world instances are not available to the algorithm designers. For some combinatorial optimization problems, randomly generated instances are generally hard, for other problems such instances are easy. The symmetric traveling salesman

problem seems to fall into the latter category. This is the case when, for example, the distances are drawn from a uniform distribution. To support this claim experimentally, we generated ten 10,000-city instances whose edge weights were taken from a uniform distribution of integers in the range $[0, 50000]$. We always stopped the computation after 5 hours. Within this time window, seven of them were solved to optimality. Table 9 contains the statistics of the successful runs. Since the computation of reduced costs of nonactive edges took a significant amount of time in some cases, we list the percentage of time spent for this in an extra column called "% Pricing". The unaccounted percentage of the time is essentially spent in the initialization process. In all cases separation took negligible time.

Table 9

Optimal solutions of 10,000 city random instances

BC	Cuts	Nlp	% Heu	% LP	% Pricing	Time
2	48	35	21.5	51.0	7.5	9080
22	88	64	16.0	58.8	4.2	9205
10	73	47	10.6	41.9	32.0	11817
0	43	31	8.9	62.7	5.0	7670
46	129	107	16.0	60.6	6.3	11825
52	132	115	4.2	30.4	56.2	22360
20	115	74	8.1	34.8	42.3	16318

However, in the Euclidean case, we could not observe a significant difference in difficulty between real-world and randomly created instances, whose coordinates are uniformly distributed on a square.

Instances arising from transformations

Recently Balas, Ceria and Cornuéjols (1993) reported on the solution of a difficult 43-city asymmetric TSP instance, which arises from a scheduling problem of a chemical plant. They solved the problem in a few minutes of a SUN SPARCstation 330 with a general purpose branch and cut algorithm that does no substantial exploitation of the structural properties of the asymmetric TSP. They also tried to solve the problem with a special purpose branch and bound algorithm for the asymmetric TSP, based on a additive bounding procedure described in Fischetti and Toth (1992), with an implementation of the authors. This algorithm could not find an optimal solution within a day of computation on the same computer. We transformed the asymmetric TSP instance to a symmetric one having 86 nodes, using the transformation described in Section 2 and solved it in less than a minute using only subtour elimination inequalities.

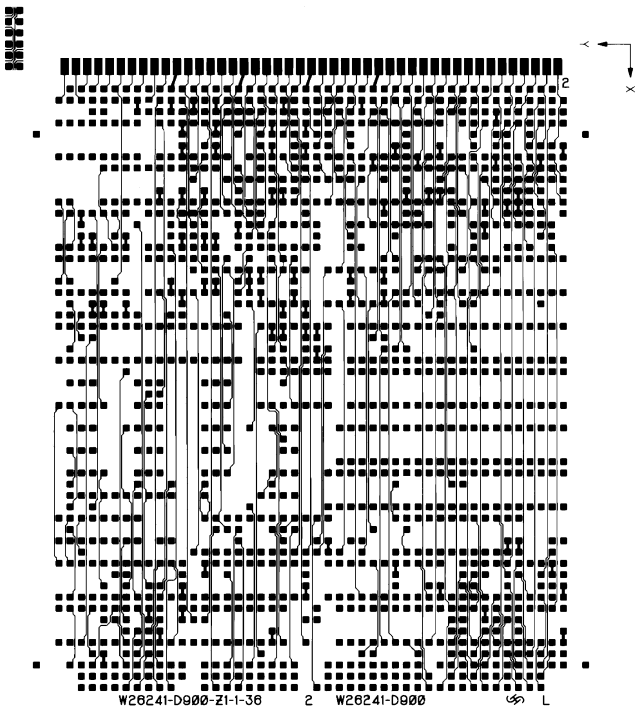

Figure 20. Mask for a printed circuit board.

In a paper about a polyhedral approach to the rural postman problem, Corberán and Sanchis (1991) describe two problem instances which are based on the street map of the city of Albaida (Valencia). The two instances are obtained by declaring two different randomly chosen sets of edges as required. The underlying graph has 171 edges and 113 nodes, which represent all streets and intersections of the streets, respectively. The first instance has 99 required edges giving rise to 10 connected components of required edges. The second has 83 required edges in 11 connected components. We applied the transformation described in Section 2, thus producing TSP instances of 198 and 176 nodes, respectively. The solution time was 89 seconds for the first and 22 seconds for the second instance.

Combinatorial optimization problems arising in the context of the control of plotting and drilling machines are described in Grötschel, Jünger and Reinelt (1991). While the drilling problems lead directly to TSP instances, the plotting problem is modeled as a sequence of Hamiltonian path and "rural postman path" problems. One of the problem instances is shown in Figure 20.

We use this mask to demonstrate the optimal solution of the three rural postman instances contained in it. The biggest of them has 258 required edges which correspond to the thin electrical connections between squares, the smallest of 10 required edges to the thick connections. The third instance has 87 required edges and corresponds to drawing the letters and digits at the bottom of the

mask. Since the movements of the light source (see Section 3) are carried out by two independent motors in horizontal and vertical directions, we choose the Maximum metric (L_∞) for distances between points. (The mask gives also rise to two TSP instances of 45 and 1432 nodes, the Euclidean version of the latter is contained in the TSPLIB under the name *u1432*.) We solve the three rural postman instances independently, starting and ending each time at an origin outside the mask, so in addition to the required edges we have one required node in each case. Table 10 gives the statistics, with a column labeled "Nre" for the number of required edges. All nodes except the origin have exactly one incident required edge in all three instances, so that the number of nodes in the TSP instance produced by the transformation is always $2\text{Nre} + 1$.

Table 10

Optimal solutions for mask plotting rural postman instances

Nre	BC	Time
10	0	0
87	2	28
258	224	9458

7.2. Provably good solutions

A branch and cut algorithm as outlined in the previous section produces a sequence of increasing lower bounds as well as a sequence of Hamiltonian cycles of decreasing lengths. Therefore, at any point during the computation we have a solution along with a quality guarantee. Looking more closely at the optimization process we observe that quality guarantees of, say, 5% are obtained quickly whereas it takes a very long time to close the last 1%.

A typical example of this behavior is shown in Figure 21 for the problem *pcb442*. The jumps in the lower bounds are due to the fact that the validity of the LP-value as a global lower bound for the length of a shortest Hamiltonian cycle is only guaranteed after a pricing step in which all nonactive variables price out correctly. The lower bound obtained after about 17 seconds is slightly increasing over time, although this is not visible in the picture. After about 10 seconds, a solution is found which can be guaranteed to deviate at most 5.220% from the optimum. At the end of the root branch and cut node, the quality guarantee is 0.302%. (For this and the following experiments we have disabled the enumerative part of the algorithm. The implementation used here is the one by Jünger, Reinelt and Thienel (1992) using the CPLEX LP-software.)

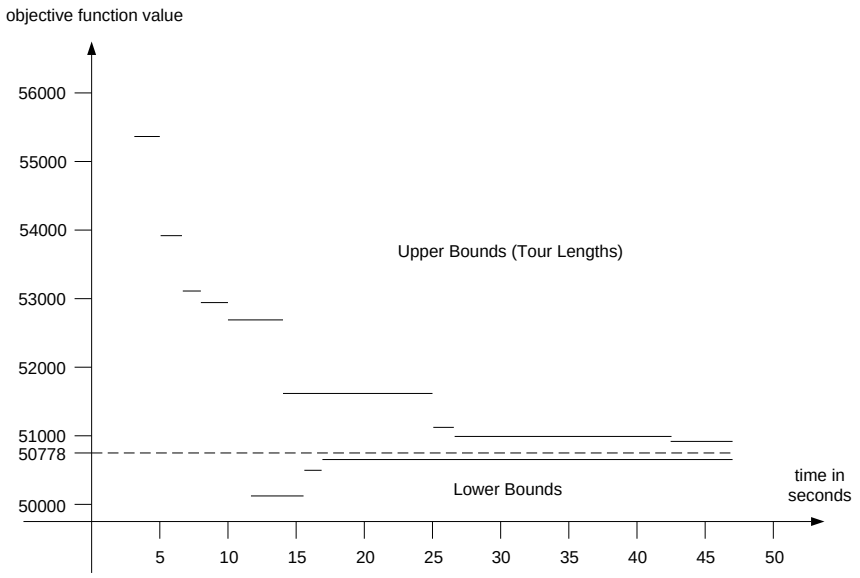

Figure 21. Gap versus time plot for *pcb442*.

The phenomenon depicted in Figure 21 is indeed typical as the computational results in Table 11 show. Here for all the problems in our list we show the number of LPs solved (before the enumerative part would have been entered), the computation time in seconds, the guaranteed quality in percent, and the actual quality (in terms of the deviation of the known optimal solution) in percent.

Our approach for computing solutions of certified good quality fails miserably on the artificial Euclidean instance *ts225*. Table 12 shows the number of branch and cut nodes (BC) and the lower bounds (LB) after 200, 400, 600, 800 and 1000 minutes of computation. A Hamiltonian cycle of length 126643 (which we believe is optimal) is found after 72 minutes. No essential progress is made as the computation progresses.

On the other hand, large real world instances can be treated successfully in this framework. As an example, we consider the Euclidean TSPLIB instance *d18512* whose nodes correspond to cities and villages in Germany. This instance was presented by Bachem and Wottawa (1991) along with a Hamiltonian cycle of length 672,721 and a lower bound on the value of an optimal solution of 597,832. Considering only subtour elimination and simple comb inequalities, we ran our standard implementation to the end of the root node computation, and obtained a Hamiltonian cycle of length 648,093 and a lower bound of 644,448 in 1295 minutes which results in a quality guarantee of about 0.57%. This Hamiltonian cycle is shown in Figure 22. Even using only subtour elimination inequalities, we obtained a lower bound of 642,082, i.e., a quality guarantee of less than 1%. In both cases we solved the first LP by the barrier method which was recently added to the CPLEX software.

Table 11

Computational results without branching

Problem	Nlp	Time	Guarantee	Quality
lin105	9	1	0.000	0.000
pr107	12	1	0.000	0.000
pr124	18	4	1.269	0.078
pr136	14	4	0.698	0.150
pr144	17	5	0.396	0.360
pr152	71	13	0.411	0.000
u159	21	7	0.202	0.000
rat195	73	60	0.430	0.130
d198	44	34	0.297	0.051
pr226	24	11	0.029	0.000
gil262	47	30	0.439	0.170
pr264	30	14	0.026	0.000
pr299	99	81	0.876	0.280
lin318	80	105	0.471	0.380
rd400	49	65	0.406	0.100
pr439	74	156	0.948	0.200
pcb442	32	39	0.302	0.185
d493	61	123	0.216	0.069
u574	86	173	0.182	0.073
rat575	60	128	0.444	0.207
p654	55	121	0.169	0.104
d657	80	248	0.779	0.033
u724	66	171	0.448	0.227
rat783	61	190	0.174	0.057
pr1002	110	485	0.249	0.024
pcb1173	92	520	0.361	0.030
rl1304	144	1239	1.025	0.421
nrw1379	92	736	0.386	0.290
u1432	132	1302	0.981	0.883
pr2392	148	3199	1.011	0.790

When the size of the instance gets even larger, memory and time consumption prohibit the application of our method. For very large Euclidean instances, Johnson (1992) reports tours found by his implementation of a variant of the Lin-Kernighan heuristic, together with lower bounds obtained with a variant of the 1-tree relaxation method described above, which constitute excellent quality guarantees. Among the instances he considered are the TSPLIB instances *pla33810* and *pla85900*. For *pla33810*, he reports a solution of length 66,138,592 and a lower bound of 65,667,327. We applied a simple strategy for this instance. Trying to exploit the clusters in the problem data, we preselected a set of subtour elimination inequalities, solved the resulting linear program containing them plus

Table 12

Lower bounds for *ts225*

Time	200 min	400 min	600 min	800 min	1000 min
BC	2300	4660	6460	7220	8172
LB	123437	123576	123629	123642	123656

the degree equations on the Delaunay graph, priced out the nonactive edges and resolved until global optimality on the relaxation was established. As LP-solver, we used the program LOQO of Vanderbei (1992), because we found the implemented interior point algorithm superior to the simplex method. Table 13 shows the results for different sets of subtours. The implementation is rather primitive, the running time can be improved significantly.

Table 13

Lower bounds for *pla33810*

# Subtours	0	4	466	1114
Lower bound	65,354,778	65,400,649	65,579,139	65,582,859
Time	51,485	36,433	47,238	104,161

7.3. Conclusions

In the recent years many new algorithmic approaches to the TSP (and other combinatorial optimization problems) have been extensively discussed in the literature. Many of them produce solutions of surprisingly good quality. However, the quality could only be assessed because optimal solutions or good lower bounds were known.

When optimization problems arise in practice we want to have confidence in the quality of the solutions. Quality guarantees become possible by reasonably efficient calculations of lower bounds. The branch and cut approach meets the goals of simultaneously producing good solutions as well as reasonable quality guarantees.

We believe that practical problem solving does not consist only of producing "probably good" but *provably good* solutions.

Figure 22. A 0.0057-guaranteed solution of *d18512*.

Acknowledgements

We are grateful to Martin Grötschel, Volker Kaibel, Denis Naddef, George Nemhauser, Peter Störmer, Laurence Wolsey, and an anonymous referee who took the time to read an earlier version of the manuscript and made many valuable suggestions. Thanks are due to Sebastian Leipert, who implemented the transformation of the rural postman to the traveling salesman problem. We are particularly thankful to Stefan Thienel who generously helped us with our computational experiments, and heavily influenced the contents of sections 6 and 7.

References

E.H.L. Aarts and J. Korst (1989), *Simulated Annealing and Boltzmann Machines*, (John Wiley & Sons, Chichester) [42].

P. Ablay (1987), Optimieren mit Evolutionsstrategien, *Spektrum der Wissenschaft* 7, 104–115 [43].

I. Althöfer and K.-U. Koschnick (1989), On the Convergence of "Threshold Accepting", Report, Universität Bielefeld [42].

D. Applegate, R.E. Bixby, V. Chvátal and W. Cook (1993), personal communication [92].

D. Applegate, V. Chvátal and W. Cook (1990), Data Structures for the Lin-Kernighan Heuristic, talk presented at the TSP-Workshop 1990, CRPC, Rice University [29].

J.L. Arthur and J.O. Frendeway (1985), A computational study of tour construction procedures for the traveling salesman problem, Research report, Oregon State University, Corvallis [22].

A. Bachem and M. Wottawa (1991), Ein 18512-Städte (Deutschland) Traveling Salesman Problem, Report 91.97, Mathematisches Institut, Universität zu Köln [98].

E. Balas and P. Toth (1985), Branch and bound methods, in: E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan and D.B. Shmoys, eds., *The Traveling Salesman Problem*, John Wiley & Sons, Chichester, pp. 361–401 [73, 74].

E. Balas, S. Ceria and G. Cornuéjols (1993), A lift-and-project cutting plane algorithm for mixed 0–1 programs, *Mathematical Programming* 58, 295–324 [95].

J.J. Bartholdi and L.K. Platzman (1982), An $O(n \log n)$ Planar Travelling Salesman Heuristic Based on Spacefilling Curves, *Operations Research Letters* 4, 121–125 [33, 34].

R. Bayer (1972), Symmetric binary b-trees: Data structure and maintenance algorithms, *Acta Informatica* 1, 290–306 [90].

J. Beardwood, J.H. Halton and J.M. Hammersley (1959), The shortest path through many points, Proc. Cambridge Philos. Soc. 55, 299–327 [34].

J.L. Bentley (1992), Fast Algorithms for Geometric Traveling Salesman Problems, *ORSA Journal on Computing* 4, 387–411 [11, 16, 25, 26, 33, 39].

R.E. Bland and D.F. Shallcross (1987), Large traveling salesman problem arising from experiments in X-ray crystallography: a preliminary report on computation, Technical Report No. 730, School of OR/IE, Cornell University [8].

S.C. Boyd and W.H. Cunningham (1991), Small travelling salesman polytopes, *Mathematics of Operations Research* 16, 259–271 [51, 58, 63].

S.C. Boyd, W.H. Cunningham, M. Queyranne and Y. Wang (1993), Ladders for travelling salesman, Preprint, Department of Combinatorics and Optimization, University of Waterloo, Waterloo, Ontario, Canada [58].

S.C. Boyd and W.R. Pulleyblank (1990), Optimizing over the subtour polytope of the traveling salesman problem, *Mathematical Programming* 49, 163–187 [48].

R.E. Burkard (1989), Special Cases of Travelling Salesman Problems and Heuristics, Report 135-89, Technische Universität Graz [3].

G. Carpaneto, M. Fischetti and P. Toth (1989), New lower bounds for the symmetric travelling salesman problem, *Mathematical Programming* 45, 233–254 [73].

V. Cerny (1985), A Thermodynamical Approach to the Travelling Salesman Problem: An Efficient Simulation Algorithm, *J. Optimization Theory and Applications* 45, 41–51 [40].

T. Christof, M. Jünger and G. Reinelt (1991), A complete description of the traveling salesman polytope on 8 nodes, *Operations Research Letters* 10, 497–500 [51, 62].

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 [17].

N. Christofides (1979), The Travelling Salesman Problem, in: N. Christofides, A. Mingozzi, P. Toth and C. Sandi, eds., *Combinatorial Optimization*, John Wiley & Sons, Chichester, pp. 131–149 [74].

V. Chvátal (1973), Edmonds polytopes and weakly Hamiltonian graphs, *Mathematical Programming* 5, 29–40 [54, 62].

G. Clarke and J.W. Wright (1964), Scheduling of vehicles from a central depot to a number of delivery points, *Operations Research* 12, 568–581 [19].

J.M. Clochard and D. Naddef (1993), Using path inequalities in a branch and cut code for the symmetric traveling salesman problem, in: G. Rinaldi and L. Wolsey, eds., *Integer Programming and Combinatorial Optimization 3*, CORE, Louvain-la-Neuve, pp. 291–311 [69, 92, 93].

N.E. Collins, R.W. Eglese and B.L. Golden (1988), Simulated Annealing: An Annotated Bibliography, *Amer. J. Math. and Mgmt. Sci.* 8, 205–307 [42].

A. Corberán and J.M. Sanchis (1991), A polyhedral approach to the rural postman problem, Working paper, Facultad de Matemáticas, Universidad de Valencia [96].

T.H. Cormen, Ch.E. Leiserson and R.L. Rivest (1989), *Introduction to Algorithms,* (MIT Press, Cambridge) [11, 38, 90].

G. Cornuéjols, J. Fonlupt and D. Naddef (1985), The traveling salesman problem on a graph and some related polyhedra, *Mathematical Programming* 33, 1–27 [52, 57].

G. Cornuéjols and G.L. Nemhauser (1978), Tight bounds for Christofides' TSP heuristic, *Mathematical Programming* 14, 116–121 [18].

CPLEX (1993), *Using the CPLEX callable library and CPLEX mixed integer library,* (CPLEX Optimization, Inc.) [86].

T.M. Cronin (1990), The Voronoi diagram for the Euclidean Traveling Salesman Problem is Piecemeal Hyperbolic, CECOM Center for Signals Warfare, Warrenton [38].

H. Crowder and M.W. Padberg (1980), Solving large-scale symmetric traveling salesman problems to optimality, *Management Science* 26, 495–509 [65, 76].

W. Cunningham and A.B. Marsh III (1978), A Primal Algorithm for Optimum Matching, *Mathematical Programming Study* 8, 50–72 [50].

G.B. Dantzig, D.R. Fulkerson and S.M. Johnson (1954), Solution of a large scale traveling-salesman problem, *Operations Research* 2, 393–410 [54, 76].

B. Delaunay (1934), Sur la sphère vide, *Izvestia Akademia Nauk. SSSr, VII Seria, Otdelenie Matemtischeskii i Estestvennyka Nauk 7* 6, 793–800 [36].

M.B. Dillencourt (1987a), Traveling Salesman Cycles are not Always Subgraphs of Delaunay Triangulations or of Minimum Weight Triangulations, *Information Processing Letters* 24, 339–342 [39].

M.B. Dillencourt (1987b), A Non-Hamiltonian, Nondegenerate Delaunay Triangulation, *Information Processing Letters* 25, 149–151 [39].

W. Dreissig and W. Uebach (1990), personal communication [8].

G. Dueck (1990), New Simple Heuristics for Optimization, Research report, IBM Scientific Center Heidelberg [42].

G. Dueck and T. Scheuer (1988), Threshold Accepting: A General Purpose Algorithm Appearing Superior to Simulated Annealing, TR 88.10.011, IBM Scientific Center Heidelberg [42].

R. Durbin and D. Willshaw (1987), An analogue approach to the travelling salesman problem using an elastic net method, *Nature* 326, 689–691 [44].

J. Edmonds (1965), Maximum matching and a polyhedron with 0,1-vertices, *Journal of Research of the National Bureau of Standards B* 69, 125–130 [17, 51].

J. Edmonds and E.L. Johnson (1970), Matching: a Well-Solved Class of Integer Linear Programs, in Proceedings of the Calgary International Conference on Combinatorial Structures and Their Applications, R.K. Guy et al., eds., Gordon and Breach, pp. 89–92 [50].

J. Edmonds and E.L. Johnson (1973), Matching, Euler tours and the Chinese postman, *Mathematical Programming* 5, 88–124 [5].

M. Fischetti and P. Toth (1992), An additive bounding procedure for the asymmetric travelling salesman problem, *Mathematical Programming* 53, 173–197 [95].

B. Fleischmann (1987), Cutting planes for the symmetric traveling salesman problem, Research Report, Universität Hamburg [63].

B. Fleischmann (1988), A New Class of Cutting Planes for the Symmetric Travelling Salesman Problem, *Mathematical Programming* 40, 225–246 [63].

B. Fritzke and P. Wilke (1991), FLEXMAP – A Neural Network for the Traveling Salesman Problem with Linear Time and Space Complexity, Research Report, Universität Erlangen-Nürnberg [45].

R.S. Garfinkel (1985), Motivation and Modeling, in: E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan and D.B. Shmoys, eds., *The Traveling Salesman Problem* John Wiley & Sons, Chichester, pp. 307–360 [3].

M. Gendreau, A. Hertz and G. Laporte (1992), New Insertion and Postoptimization Procedures for the Traveling Salesman Problem, *Operations Research* 40, 1086–1094 [23].

P.C. Gilmore, E.L. Lawler and D.B. Shmoys (1985), Well-solved special cases, in: E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan and D.B. Shmoys, eds., *The Traveling Salesman Problem*, John Wiley & Sons, Chichester, pp. 87–143 [3].

F. Glover (1990), Tabu Search, *ORSA Journal on Computing* 1, 190–206 (Part I), 2, 4–32 (Part II) [44].

M.X. Goemans (1993), Worst-case Comparison of Valid Inequalities for the TSP, Preprint, Department of Mathematics, Massachusetts Institute of Technology, Cambridge [48, 70].

D.E. Goldberg (1989), *Genetic algorithms in search, optimization and machine learning,* (Addison-Wesley) [44].

A.V. Goldberg and R.E. Tarjan (1988), A new approach to the maximum flow problem, *Journal of the ACM* 35, 921–940 [65].

B.L. Golden and W.R. Stewart (1985), Empirical analysis of heuristics, in: E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan and D.B. Shmoys, eds., *The Traveling Salesman Problem* John Wiley & Sons, Chichester, pp. 207–249 [11].

R.E. Gomory (1958), Outline of an algorithm for integer solutions to linear programs, *Bull. Amer. Math. Soc.* 64, 275–278 [76].

R.E. Gomory (1960), Solving linear programming problems in integers, *Proc. Sympos. Appl. Math.* 10, 211–215 [76].

R.E. Gomory (1963), An algorithm for integer solutions to linear programs, in: R.L. Graves and P. Wolfe, eds., *Recent Advances in Mathematical Programming*, McGraw Hill, New York, pp. 269–302 [76].

R.E. Gomory and T.C. Hu (1961), Multi-terminal network flows, *SIAM Journal on Applied Mathematics* 9, 551-570 [65].

R. L. Graham (1972), An efficient algorithm for determining the convex hull fo a finite planar set, *Information Processing Letters* 1, 132–133 [36].

M. Grötschel (1977), *Polyedrische Charakterisierungen kombinatorischer Optimierungsprobleme,* (Hain, Meisenheim am Glan) [52, 76].

M. Grötschel (1980), On the symmetric traveling salesman problem: solution of a 120-city problem, *Mathematical Programming Studies* 12, 61–77 [76].

M. Grötschel and O. Holland (1987), A cutting plane algorithm for minimum perfect 2-matching, *Computing* 39, 327–344 [66, 67].

M. Grötschel and O. Holland (1991), Solution of Large-scale Symmetric Traveling Salesman Problems, *Mathematical Programming* 51, 141–202 [65, 69, 77].

M. Grötschel, M. Jünger and G. Reinelt (1984), A Cutting Plane Algorithm for the Linear Ordering Problem, *Operations Research* 32, 1195–1220 [77].

M. Grötschel, M. Jünger and G. Reinelt (1991), Optimal Control of Plotting and Drilling Machines: A Case Study, *Zeitschrift für Operations Research – Methods and Models of Operations Research* 35, 61–84 [7, 10, 96].

M. Grötschel, L. Lovász and A. Schrijver (1981), The ellipsoid methods and its consequences in combinatorial optimization, *Combinatorica* 1, 169–197 [64].

M. Grötschel, L. Lovász and A. Schrijver (1988), *Geometric Algorithms and Combinatorial Optimization,* (Springer-Verlag, Berlin-Heidelberg) [64].

M. Grötschel and M.W. Padberg (1974), Zur Oberflächenstruktur des Traveling Salesman Polytopen, in H.J. Zimmermann et al., eds., Proc. Operations Research 4, Physica, Würzburg, pp. 207–211 [52].

M. Grötschel and M.W. Padberg (1977), Lineare Charakterisierungen von Traveling Salesman Problemen, *Z. Oper. Res.* 21, 33–64 [52].

M. Grötschel and M.W. Padberg (1978), On the symmetric traveling salesman problem: theory and computation, in R. Henn et al., eds., *Optimization and Operations Research*, Lecture Notes in Economics and Mathematical Systems 157, Springer, Berlin, pp. 105–115 [52].

M. Grötschel and M.W. Padberg (1979a), On the symmetric traveling salesman problem I: inequalities, *Mathematical Programming* 16, 265–280 [52, 54].

M. Grötschel and M.W. Padberg (1979b), On the symmetric traveling salesman problem II: lifting theorems and facets, *Mathematical Programming* 16, 281–302 [52, 53, 54].

M. Grötschel and M.W. Padberg (1985), Polyhedral theory, in: E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan and D.B. Shmoys, eds., *The Traveling Salesman Problem*, John Wiley & Sons, Chichester, pp. 251–305 [51].

M. Grötschel and W.R. Pulleyblank (1986), Clique tree inequalities and the symmetric traveling salesman problem, *Mathematics of Operations Research* 11, 537–569 [55].

L.J. Guibas and R. Sedgewick (1978), A dicromatic framework for balanced trees, in: *Proceedings of the 19th annual symposium on foundations of computer science*, IEEE Computer Society, pp. 8–21 [90].

D. Gusfield (1987), Very simple algorithms and programs for all pairs network flow analysis, Preprint, Computer Science Division, University of California, Davis [66].

B. Hajek (1985), A Tutorial Survey of Theory and Applications of Simulated Annealing, Proc. 24th IEEE Conf. on Decision and Control, pp. 755–760 [42].

J. Hao and J.B. Orlin (1992), A Faster Algorithm for Finding the Minimum Cut in a Graph, Proceedings of the 3rd ACM-SIAM Symposium on Discrete Algorithms, Orlando, Florida, pp. 165–174 [65].

M. Held and R.M. Karp (1970), The Traveling Salesman Problem and Minimum Spanning Trees, *Operations Research* 18, 1138–1162 [50, 73].

M. Held and R.M. Karp (1971), The Traveling Salesman Problem and Minimum Spanning Trees: Part II, *Mathematical Programming* 1, 6–25 [50, 73].

A.J. Hoffman and P. Wolfe (1985), History, in: E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan and D.B. Shmoys, eds., *The Traveling Salesman Problem*, John Wiley & Sons, Chichester, pp. 1–15 [2].

J.J. Hopfield and D.W. Tank (1985), 'Neural' computation of decisions in optimization problems, *Biol. Cybern.* 52, 141–152 [45].

T.C. Hu (1965), Decomposition in Traveling Salesman Problems, Proc. IFORS Theory of Graphs, A34–A44 [35].

C.A.J. Hurkens (1991), Nasty TSP instances for classical insertion heuristics, University of Technology, Eindhoven [16].

D.S. Johnson (1990), Local Optimization and the Traveling Salesman Problem, Proc. 17th Colloquium on Automata, Languages and Programming, Springer Verlag, 446–461 [11, 13, 16, 19, 25, 26, 30, 33, 35, 39, 42].

D.S. Johnson (1992), personal communication [99].

D.S. Johnson, C.R. Aragon, L.A. McGeoch and C. Schevon (1991), Optimization by simulated annealing: An experimental evaluation, *Operations Research* 37, 865–892 (Part I), 39, 378–406 (Part II) [42].

D.S. Johnson and C.H. Papadimitriou (1985), Computational Complexity, in: E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan and D.B. Shmoys, eds., *The Traveling Salesman Problem*, John Wiley & Sons, Chichester, pp. 37–85 [12].

D.S. Johnson, C.H. Papadimitriou and M. Yannakakis (1988), How easy is local search?, *J. Comp. Syst. Sciences* 37, 79–100 [33].

M. Jünger, G. Reinelt and S. Thienel (1992), Optimal and provably good solutions for the symmetric traveling salesman problem, Report 92.114, Angewandte Mathematik und Informatik, Universität zu Köln [78, 87, 89, 92, 97].

M. Jünger, G. Reinelt and D. Zepf (1991), Computing Correct Delaunay Triangulations, *Computing* 47, 43–49 [38].

V. Kaibel (1993), Numerisch stabile Berechnung von Voronoi-Diagrammen, Diplomarbeit, Universität zu Köln [38].

D.R. Karger (1993), Global min-cuts in $\mathcal{RNC}$, and other ramifications of a simple min-cut algorithm, Proceedings of the 4^{th} ACM-SIAM Symposium on Discrete Algorithms, pp. 21–30 [66].

D.V. Karger and C. Stein (1993), An $\tilde{O}(n^2)$ algorithm for minimum cuts, Proceedings of the 25th ACM Symposium on the Theory of Computing, San Diego, CA, pp. 757–765 [66].

R. Karp (1977), Probabilistic analysis of partitioning algorithms for the traveling-salesman in the plane, *Mathematics of Operations Research* 2, 209–224 [34].

R.M. Karp and J.M. Steele (1985), Probabilistic Analysis of Heuristics, in: E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan and D.B. Shmoys, eds., *The Traveling Salesman Problem*, John Wiley & Sons, Chichester, pp. 181–205 [35].

C. Kemke (1988), Der Neuere Konnektionismus; Ein Überblick, *Informatik Spektrum* 11, 143–162 [45].

S. Kirkpatrick (1984), Optimization by simulated annealing: quantitative studies, *J. Statist. Physics* 34, 975–986 [42].

S. Kirkpatrick, C.D. Gelatt Jr. and M.P. Vecchi (1983), Optimization by simulated annealing, *Science* 222, 671–680 [40].

K. C. Kiwiel (1989), A Survey of Bundle Methods for Nondifferentiable Optimization, in: M. Iri & K. Tanabe (eds.) *Mathematical Programming. Recent Developments and Applications*, Kluwer Academic Publishers, Dordrecht, 263–282 [75].

J. Knox and F. Glover (1989), Comparative Testing of Traveling Salesman Heuristics Derived from Tabu Search, Genetic Algorithms and Simulated Annealing, Center for Applied Artificial Intelligence, Univ. of Colorado [44].

D.E. Knuth (1973), *The art of computer programming, Volume 3: Sorting and searching*, Addison-Wesley, Reading, Massachusetts [90].

J.B. Kruskal (1956), On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem, *Proceedings of the American Mathematical Society* 7, 48–50 [38].

E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan and D.B. Shmoys, eds. (1985), *The Traveling Salesman Problem,* (John Wiley & Sons, Chichester) [2].

J.K. Lenstra and A.H.G. Rinnooy Kan (1974), Some Simple Applications of the Travelling Salesman Problem, BW 38/74, Stichting Mathematisch Centrum, Amsterdam [9, 10].

S. Lin and B.W. Kernighan (1973), An Effective Heuristic Algorithm for the Traveling-Salesman Problem, *Operations Research* 21, 498–516 [27].

J.D. Litke (1984), An improved solution to the traveling salesman problem with thousands of nodes, *Communications ACM* 27, 1227–1236 [35].

K.-T. Mak and A.J. Morton (1993), A Modified Lin-Kernighan Traveling Salesman Heuristic, *Operations Research Letters* 13, 127–132 [30].

M. Malek, M. Guruswamy, H. Owens and M. Pandya (1989), Serial and Parallel Search Techniques for the Traveling Salesman Problem, *Annals of OR: Linkages with Artificial Intelligence* [44].

M. Malek, M. Heap, R. Kapur and A. Mourad (1989), A Fault Tolerant Implementation of the Traveling Salesman Problem, Research Report, Dpmt. of Electrical and Computer Engineering, Univ. of Texas at Austin [44].

F. Margot (1992), Quick Updates for p-OPT TSP heuristics, *Operations Research Letters* 11 [27].

R. Marsten (1981), The design of the XMP linear programming library, *ACM Transactions of Mathematical Software* 7, 481–497 [78].

O. Martin, S.W. Otto and E.W. Felten (1992), Large-step Markov Chains for the TSP incorporating local search heuristics, *Operations Research Letters* 11, 219–224 [42].

J.F. Maurras (1975), Some results on the convex hull of Hamiltonian cycles of symmetric complete graphs, in B. Roy, ed., *Combinatorial Programming: Methods and Applications*, Reidel, Dordrecht, pp. 179–190 [62].

N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller and E. Teller (1953), Equation of state calculation by fast computing machines, J. Chem. Phys. 21, 1087–1092 [40].

D.L. Miller, J.F. Pekny and G.L. Thompson (1991), An exact branch and bound algorithm for the symmetric TSP using a symmetry relaxed two-matching relaxation, talk presented at International Symposium on Mathematical Programming, Amsterdam [73].

H. Mühlenbein, M. Gorges-Schleuter and O. Krämer (1988), Evolution algorithms in combinatorial optimization, *Parallel Computing* 7, 65–85 [44].

D. Naddef (1990), Handles and teeth in the symmetric traveling salesman polytope, in W. Cook and P.D. Seymour, eds., *Polyhedral Combinatorics* DIMACS series in Discrete Mathematics and Theoretical Computer Science 1, A.M.S., pp. 61–74 [63].

D. Naddef (1992), The binested inequalities for the symmetric traveling salesman polytope, *Mathematics of Operations Research* 17, 882–900 [63].

D. Naddef and G. Rinaldi (1988), The symmetric traveling salesman polytope: New facets from the graphical relaxation, Report R. 248, IASI-CNR Rome [57, 60, 62].

D. Naddef and G. Rinaldi (1991), The symmetric traveling salesman polytope and its graphical relaxation: Composition of valid inequalities, *Mathematical Programming 51, 359-400* [62].

D. Naddef and G. Rinaldi (1992), The crown inequalities for the symmetric traveling salesman polytope, *Mathematics of Operations Research* 17, 308–326 [58, 60].

D. Naddef and G. Rinaldi (1993), The graphical relaxation: a new framework for the symmetric traveling salesman polytope, *Mathematical Programming* 58, 53–88 [52, 53, 59, 60, 62].

H. Nagamochi and T. Ibaraki (1992a), A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph, *Algorithmica* 7, 583–596 [65].

H. Nagamochi and T. Ibaraki (1992b), Computing edge-connectivity in multigraphs and capacitated graphs, *SIAM Journal on Discrete Mathematics* 5, 54–66 [65].

G.L. Nemhauser and L.A. Wolsey (1988), *Integer and Combinatorial Optimization,* (John Wiley & Sons, Chichester) [50, 51].

R.Z. Norman (1955), On the convex polyhedra of the symmetric traveling salesman problem (abstract), *Bulletin of the A.M.S.* 61, 559 [51].

T. Ohya, M. Iri and K. Murota (1984), Improvements of the Incremental Method for the Voronoi Diagram with Computational Comparison of Various Algorithms, *Journal of the Operations Research Society of Japan* 27, 306–337 [37].

I. Or (1976), Traveling Salesman-Type Combinatorial Problems and Their Relation to the Logistics of Regional Blood Banking, Northwestern University, Evanston, Il [27].

M.W. Padberg and M. Grötschel (1985), Polyhedral computations, in: E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan and D.B. Shmoys, eds., *The Traveling Salesman Problem*, John Wiley & Sons, Chichester, pp. 307–360 [66, 67].

M.W. Padberg and S. Hong (1980), On the symmetric traveling salesman problem: a computational study, *Mathematical Programming Studies* 12, 78–107 [63, 67, 76].

M.W. Padberg and M.R. Rao (1981), The Russian method for linear inequalities III: Boundend integer programming, GBA Working paper 81–39 New York University, New York [64].

M.W. Padberg and M.R. Rao (1982), Odd minimum cut sets and b-matchings, *Math. Oper. Res.* 7, 67–80 [50, 66, 77].

M.W. Padberg and G. Rinaldi (1987), Optimization of a 532 City Symmetric Traveling Salesman Problem by Branch and Cut, *Operations Research Letters* 6, 1–7 [77].

M.W. Padberg and G. Rinaldi (1990a), An Efficient Algorithm for the Minimum Capacity Cut Problem, *Mathematical Programming* 47, 19–36 [65].

M.W. Padberg and G. Rinaldi (1990b), Facet Identification for the Symmetric Traveling Salesman Polytope, *Mathematical Programming* 47, 219–257 [66, 67, 68, 69, 70].

M.W. Padberg and G. Rinaldi (1991), A Branch and Cut Algorithm for the Resolution of Large-scale Symmetric Traveling Salesman Problems, *SIAM Review* 33, 60–100 [70, 77, 78, 87, 88, 92].

M. Padberg and T.-Y. Sung (1988), A polynomial-time solution to Papadimitriou and Steiglitz's 'traps', *Operations research Letters* 7, 117–125 [48].

C.H. Papadimitriou (1990), The Complexity of the Lin-Kernighan Heuristic for the Traveling Salesman Problem, University of California, San Diego [27].

R.D. Plante, T.J. Lowe and R. Chandrasekaran (1987), The Product Matrix Traveling Salesman Problem: An Application and Solution Heuristics, *Operations Research* 35, 772–783 [8].

B. T. Polyak (1978), Subgradient Methods: A Survey of Soviet Research, in: C. Lemarèchal & R. Mifflin (eds.), *Nonsmooth Optimization*, Pergamon Press, Oxford, 5–29 [74].

J.-Y. Potvin and J.-M. Rousseau (1990), Enhancements to the Clarke and Wright Algorithm for the Traveling Salesman Problem, Research report, University of Montreal [19].

R.C. Prim (1957), Shortest Connection Networks and Some Generalizations, *The Bell System Technical Journal* 36, 1389–1401 [17].

W.R. Pulleyblank (1983), Polyhedral Combinatorics, in *Mathematical Programming The State of the Art*, A. Bachem et al., eds., Springer-Verlag, pp. 312–345 [51].

M. Queyranne and Y. Wang (1990), Facet tree composition for symmetric travelling salesman polytopes, Working paper 90–MSC–001, Faculty of Commerce, University of British Columbia, Vancouver, B.C., Canada [62, 63].

M. Queyranne and Y. Wang (1993), Hamiltonian path and symmetric travelling salesman polytopes, *mathematical Programming* 58, 89–110 [60].

H.D. Ratliff and A.S. Rosenthal (1981), Order-Picking in a Rectangular Warehouse: A Solvable Case for the Travelling Salesman Problem, PDRC Report Series, No. 81–10 [8].

I. Rechenberg (1973), *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution,* (Frommann-Holzboog, Stuttgart) [43].

G. Reinelt (1991a), TSPLIB – A Traveling Salesman Problem Library, *ORSA Journal on Computing* 3, 376–384 [11, 80, 81].

G. Reinelt (1991b), TSPLIB – Version 1.2, Report No. 330, Schwerpunktprogramm der Deutschen Forschungsgemeinschaft, Universität Augsburg [11, 80, 81].

G. Reinelt (1992), Fast Heuristics for Large Geometric Traveling Salesman Problems, *ORSA Journal on Computing* 2, 206–217 [11].

G. Reinelt (1994), *Contributions to Practical Traveling Salesman Problem Solving,* (Lecture Notes in Scientific Computing, Springer) [11, 35].

D.J. Rosenkrantz, R.E. Stearns and P.M. Lewis (1977), An analysis of several heuristics for the traveling salesman problem, *SIAM J. Computing* 6, 563–581 [13, 16, 27].

P. Ruján (1988), Searching for optimal configurations by simulated tunneling, *Z. Physik B – Condensed Matter* 73, 391–416 [42].

P. Ruján, C. Evertsz and J.W. Lyklema (1988), A Laplacian Walk for the Travelling Salesman, *Europhys. Lett.* 7, 191–195 [38].

D.E. Rumelhart, G.E. Hinton and J.L. McClelland (1986), *The PDP Research Group: Parallel Distributed Processing: Explorations in the Microstructure of Cognition,* (MIT Press) [45].

S. Sahni and T. Gonzales (1976), P-complete approximation problems, *J. Assoc. Comp. Mach.* 23, 555-565 [11].

H. Schramm (1989), *Eine Kombination von Bundle- und Trust-Region-Verfahren zur Lösung nichtdifferenzierbarer Optimierungsprobleme,* (Bayreuther Mathematische Schriften, Heft 30) [75].

A. Segal, R. Zhang and J. Tsai (1991), A New Heuristic Method for Traveling Salesman Problem, University of Illinois, Chicago [38].

M. I. Shamos and D. Hoey (1975), Closest point problems, Proc. 16th IEEE Ann. Symp. Found. Comput. Sci. 151–162 [37].

D.B. Shmoys and D.P. Williamson (1990), Analyzing the Held-Karp TSP bound: A monotonicity property with application, *Information Processing Letters* 35, 281–285 [48].

R.E. Tarjan (1983), *Data Structures and Network Algorithms,* (Society for Industrial and Applied Mathematics, Philadelphia) [38].

N.L.J. Ulder, E. Pesch, P.J.M. van Laarhoven, H.-J. Bandelt and E.H.L. Aarts (1990), Improving TSP Exchange Heuristics by Population Genetics, Preprint, Erasmus Universiteit Rotterdam [44].

R. van Dal (1992), *Special Cases of the Traveling Salesman Problem,* (Wolters-Noordhoff, Groningen) [3, 8].

R. van der Veen (1992), Solvable Cases of the Traveling Salesman Problem with Various Objective Functions, Doctoral Thesis, Rijksuniversiteit Groningen, Groningen [3].

P.J.M. van Laarhoven (1988), Theoretical and Computational Aspects of Simulated Annealing, PhD Thesis, Erasmus Universiteit Rotterdam [42].

R.J. Vanderbei (1992), LOQO User's Manual, Preprint, Statistics and Operations Research, Princeton University [100].

T. Volgenant and R. Jonker (1982), A branch and bound algorithm for the symmetric traveling salesman problem based on the 1-tree relaxation, *European J. Oper. Res.* 9, 83–89 [74].

G. Voronoi (1908), Nouvelles applications des paramètres continus à la théorie des formes quadratiques. Deuxième memoire: Recherche sur les parallélloèdres primitifs, *Journal für Reine und Angewandte Mathematik* 3, 198–287 [36].

R.H. Warren (1993), Special cases of the traveling salesman problem, Preprint, Advanced Concepts Center, Martin Marietta Corporation, King of Prussia, PA, to appear in *Applied Mathematics and Computation* [3].

L.A. Wolsey (1980), Heuristic analysis, linear programming and branch and bound, *Mathematical Programming Study* 13, 121–134 [48].

INDEX OF DEFINITIONS