ANGEWANDTE MATHEMATIK UND INFORMATIK
UNIVERSITÄT ZU KÖLN

Report No. 92.114

Provably good solutions for the traveling salesman problem

by

Michael Jünger
Gerhard Reinelt
Stefan Thienel

1992

Institut für Informatik
UNIVERSITÄT ZU KÖLN
Pohligstraße 1
D-5000 Köln 51

Addresses of the authors:

Michael Jünger
Institut für Informatik
Universität zu Köln
Pohligstraße 1
D-5000 Köln 51
Germany
E-mail: mjuenger@informatik.uni-koeln.de
Telephone: 49 221 3671113

Gerhard Reinelt
Institut für Angewandte Mathematik
Universität Heidelberg
Im Neuenheimer Feld 293
D-6900 Heidelberg
Germany
E-mail: reinelt@ares.iwr.uni-heidelberg.de
Telephone: 49 6221 563171

Stefan Thienel
Institut für Informatik
Universität zu Köln
Pohligstraße 1
D-5000 Köln 51
Germany
E-mail: thienel@informatik.uni-koeln.de
Telephone: 49 221 3671107

Abstract

The determination of true optimum solutions of combinatorial optimization problems is seldomly required in practical applications. The majority of users of optimization software would be satisfied with solutions of guaranteed quality in the sense that it can be proven that the given solution is at most a few percent off an optimum solution. This paper presents a general framework for practical problem solving with emphasis on this aspect. A detailed discussion along with a report about extensive computational experiments is given for the traveling salesman problem.

Recent research in algorithm design for hard combinatorial optimization problems follows two trends. One aims at a continuously better understanding of structural properties of a given problem in the pursuit of solving larger instances to optimality. For a few prominent hard combinatorial optimization problems such as the traveling salesman problem (TSP), the linear ordering problem or the max-cut problem, branch & cut algorithms are clearly the state-of-the-art. Every few years, increasingly larger instances are solved. For example, the largest (reasonable, non-random) solved instance of the TSP (in this paper we denote by TSP always the symmetric traveling salesman problem) was 49 in 1954 (Dantzig, Fulkerson and Johnson (1954)), 120 in 1977 (Grötschel (1977, 1980)), 318 in 1980 (Crowder and Padberg (1980)), 666 in 1988 (Grötschel and Holland (1991)), 2392 in 1988 (Padberg and Rinaldi (1991)) and 3038 in 1992 (Applegate, Bixby, Chvátal and Cook (personal communications, 1992)). However, in all cases, enormous amounts of computer time (in view of the available hardware and software technology of the respective time) had to be invested. The operations researcher who is interested in practical problem solving correctly criticizes the described algorithms as impractical, even more, since none of the above mentioned published results indicates the solvability of all or, at least, most instances of similar size. Consequently, a second trend evolved, aiming at finding reasonably good solutions for large instances within given computation time bounds. The quality of the solutions of the proposed heuristics can be determined in several ways. For demonstration, we again use the traveling salesman problem. One way consists of proving that the solution is never worse than a fixed fraction of an optimum solution. In the case of the Euclidean traveling salesman problem, where all cities have Cartesian coordinates and the length of a connection is the Euclidean distance between the two connected cities, Christofides (1976) gave a heuristic which produces a tour whose length is at most 1.5 times as long as an optimum one. Unfortunately, results like this, although theoretically appealing, are of little practical value since the guaranteed quality is poor. On the other hand, many good heuristics have been developed which can compute very high quality solutions for large real world instances, such as the heuristic by Lin and Kerninghan (1973), yet the quality of their solutions cannot be assessed directly. Instead, sophisticated methods for obtaining good lower bounds on the length of a tour must be applied to demonstrate their good performance in each case. For a survey of such methods, see Johnson (1990), Bentley (1991), Reinelt (1992).

This paper is an attempt to combine the two trends in one algorithmic framework, which produces a sequence of increasingly shorter tours and increasingly better lower bounds on the length of an optimum tour, such that a user interested in the solution of a prespecified guarantee can stop the computational process as soon as he is satisfied with a guarantee of the kind: "The currently best known tour is at most $p\%$ longer than the shortest one". We would like to demonstrate that a branch & cut algorithm enhanced with good heuristics for constructing and improving tours guided by structural information produced in the "ordinary" branch & cut part of the algorithm can achieve this goal.

We believe that the only way to substantiate a claim like ours is to give a detailed description of the implemented computer program, to provide convincing computational results on problem instances of the general accessible TSPLIB (Reinelt (1991a, 1991b)), and to show the limitations of the proposed approach as well by explaining why we cannot solve all of the library problems satisfactorily.

Section 1 gives a general outline of our extended notion of the branch & cut algorithm as indicated above, section 2 gives details of our implementation for the traveling salesman problem, and section 3 describes computational experiments with various parameter settings.

We would like to close this introduction with some disclaimers. We do not believe that the proposed method is good for all, or even many, combinatorial optimization problems. The structural properties of most combinatorial optimization problems arising in practical applications are much too complicated for such an approach. It seems that due to our limited knowledge such problems are, at least at present, in practice much better approached by heuristic methods and algorithms which simulate biological and physical phenomena, such as genetic or evolutionary algorithms or simulated annealing. In such cases one often has to drop the requirement of a provable guarantee. The traveling salesman problem belongs to a few problems for which enough structural knowledge is available to justify the branch & cut approach. Other combinatorial optimization problems are too "easy", e.g., combinatorial algorithms such as Edmonds' algorithm for the matching problem (Edmonds (1965)) clearly outperform approaches like ours in practice (Derigs and Metz (1991), Applegate and Cook (personal communications, 1991)).

Although we deal with a branch & cut algorithm for the TSP, we do not explain details of the facial structure of the TSP polytope nor give new separation algorithms. For this, we refer to the paper of Padberg and Rinaldi (1990), and the survey by Jünger, Reinelt and Rinaldi (1992). The fact that our separation routines are not as sophisticated as those by Padberg and Rinaldi is the reason that we cannot solve as large instances to optimality as they can. As explained above, our emphasis is a different one. Also, we do not explain linear programming terminology which can be found in, e.g., Chvatal (1983), Nemhauser and Wolsey (1988).

This paper gives first results of a large project of implementing and evaluating branch & cut type algorithms for well-studied, hard, but not too hard, combinatorial optimization problems such as max-cut, linear ordering, sequential ordering, maximum planar subgraph, etc., on conventional single processor and multi-processor computer systems.

1 Branch & cut

The branch & cut approach to combinatorial optimization problems is a variant of the branch & bound approach. The latter is so well known that we can omit a formal definition here. Instead we will outline the general scheme and use the example of the traveling salesman problem to summarize the refinements that lead to our notion of the branch

& cut approach. The flowchart of Figure 1 gives the basic control structure of a branch & bound algorithm for a combinatorial optimization problem with an objective function which is to be minimized.

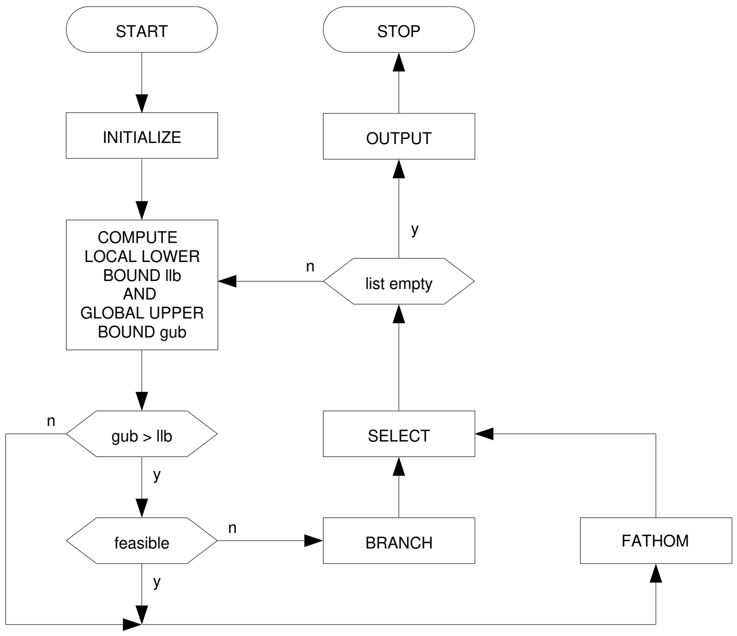

Figure 1. Flowchart of a branch & bound algorithm.

A branch & 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 step the algorithm selects a current subproblem from this list and tries to "solve" it in either of the following ways: a lower bound for the value of an optimum solution of the current problem 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 to be fathomed is empty.

The crucial part of a successful branch & bound algorithm is the computation of lower bounds. Here one uses the fundamental concept of relaxation. A **relaxation** of a combinatorial minimization problem is another minimization problem whose set of feasible solutions includes the feasible solutions of the original problem and assigns the same objective function values to them. Hence solving the relaxed problem gives a lower bound on the optimum objective function value of the problem it was derived from. The tighter the relaxation, the better this bound will be.

We introduce some terminology concerning lower bounds (derived from solving relaxations) and upper bounds (obtained by finding feasible solutions). We call a lower bound **local**, if it is only valid for a subproblem, and **global**, if it is a bound for the original problem. By solving a relaxation of the current problem we obtain a **local lower bound** `llb` for the objective function value of the original problem. It is also a **global lower bound** `glb` when this is done the first time for the initial relaxation and also later in the computation under certain conditions, which we outline in section 2. If the solution found for the relaxation happens to be feasible for the original problem (in which case it is also the optimum solution of the subproblem) and has lower objective function value than any feasible solution found so far, it is memorized and the **global upper bound** `gub` for the objective function value is decreased accordingly. If the global upper bound does not exceed the local lower bound, the current subproblem is fathomed.

If the global upper bound exceeds the local lower bound and no feasible solution was found for the current problem, we perform a branching step which consists of adding to the list a collection of new subproblems whose union of feasible solutions contains all feasible solutions of the current problem. The simplest **branching rule** consists of defining two new subproblems in one of which a fixed edge is required to be in the solution and in the other forbidden to be in the solution. More elaborate branching rules and selection rules can be found in the general survey of Balas and Toth (1985).

If the list of subproblems to be fathomed becomes empty, then the memorized feasible solution whose objective function value is equal to the global upper bound can be output as the optimum solution.

It is important to note, that during the execution of a branch & 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 optimum solution as soon as the best lower bound coincides with the value of the best feasible solution found. If we take the point of view that practical problem solving consists of producing a solution whose objective function value exceeds the objective function value of an optimum solution by at most $g\%$, then a branch & bound algorithm can achieve this goal if it stops as soon as it has found a feasible solution of value `gub` and global lower bound `glb` such that $\frac{\texttt{gub}-\texttt{glb}}{\texttt{glb}} \cdot 100 \leq g$.

Essential in the design of a branch & bound algorithm is the choice of suitable relaxations to make bounding efficient. A popular variant for the traveling salesman problem, which has been elaborated in many subsequent publications, is described in Held and Karp (1971). They use the **1-tree-relaxation**. A 1-tree in a graph $G = (V, E)$ with $V = 1, 2, \ldots, n$ consists of the edges of a spanning tree on the node set $2, \ldots, n$ plus two more edges incident to node 1. Minimum length 1-trees are easily computed, and any tour is clearly a special 1-tree. However, using 1-trees directly as a relaxation turns out to yield poor bounds. Instead, a strengthening is used which essentially amounts to the repeated computation of 1-trees in a Lagrangian relaxation framework.

In fact, Christofides (1979) and Volgenant and Jonker (1982) report lower bounds of about 99% on randomly generated problems. However, this does not carry over to real world problems and the "poor bound" we get by this procedure is probably the reason why 1-tree based branch & bound computer codes cannot compete with cutting plane based branch & bound codes, which we will discuss below.

To complete this quick survey on branch & bound algorithms, we mention a branch & bound algorithm that uses the **2-matching relaxation** and has been implemented by Miller, Pekny and Thompson (1991), and a lower bound based on an **additive bounding** procedure proposed by Carpaneto, Fischetti and Toth (1989). However, for these relaxations, no convincing results on non-random problems have been published.

The quality of the bounds obtained by solving the relaxed subproblems are essential for the performance of the algorithm where computation can amount to the almost complete enumeration with poor bounds in one extreme or the consideration of only a few or even only one subproblem in the other extreme. Published computational results show that the latter is best approached by using **linear programming relaxations** (**LP-relaxations**).

Let $K_n = (V_n, E_n)$ denote the complete undirected graph on n nodes. For $F \subseteq E_n$ let $x(F) = \sum_{e \in F} x_e$. For $W \subseteq V_n$ let $E_n(W) = \{uv \in E_n \mid u, v \in W\}$ denote the edges in E_n with endnodes in W and $\delta(W) = \{uv \in E_n \mid u \in W, v \in V_n - W\}$ the cut with shores W and $V_n - W$. For $v \in V_n$, we abbreviate $\delta(v)$ for $\delta(\{v\})$. A Hamiltonian cycle in K_n is also called a **tour**. With every tour $T \subseteq E_n$ we associate its incidence vector $\chi^T \in \{0,1\}^{E_n}$ defined by

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

If $c \in \mathbb{R}^{E_n}$ is the vector of edge weights of a TSP instance, then the following is an **integer programming formulation** of the TSP in the sense that its solutions are exactly the incidence vectors of tours:

$$\begin{aligned} \min \quad & c'x && \\ \text{s.t.} \quad & x(\delta(v)) = 2 && \text{for all } v \in V_n && (1.1) \\ & x(\delta(W)) \geq 2 && \text{for all } \emptyset \neq W \subset V_n && (1.2) \\ & 0 \leq x \leq 1 && && (1.3) \\ & x \text{ integer} && && (1.4) \end{aligned}$$

The equations (1.1) require that any node is incident to exactly two edges and the inequalities (1.2) ensure that there is only one cycle. The latter are called the **subtour elimination constraints** (given here in their cut version).

A first LP-relaxation, the **subtour relaxation**, is obtained by removing the integrality conditions (1.4). The polytope defined by (1.1), (1.2) and (1.3) is called the **subtour polytope**. The subtour relaxation can be improved by adding further inequalities. The ultimate strengthening would be the set of equations and inequalities defining the **traveling salesman polytope**

$$P_{TSP}^n = \text{conv}\left\{\chi^T \mid T \subseteq E_n \text{ is a tour in } K_n\right\}.$$

However, while it is known that no equations have to be added to those in (1.1), no complete knowledge of all necessary inequalities is available and it is unlikely that such a complete description can be given for any n. Small instances were examined, e.g., it is known that for $n = 7$ the number of required inequalities is 3,437 (Boyd and Cunningham (1991)) and for $n = 8$ it is 194,187 (Christof, Jünger, Reinelt (1991)).

In practical computation, it has turned out that facet defining inequalities for P^n_{TSP} (these are valid inequalities for P^n_{TSP} which are not dominated by any other inequalities and are therefore called required above) are very useful. Many researchers have studied P^n_{TSP} and found facet defining inequalities. An up-to-date survey on such results is given in Jünger, Reinelt and Rinaldi (1992). For our purposes it is sufficient to list only those known facets defining inequalities which we use in our implementation.

The subtour elimination inequalities (1.2) define facets of the traveling salesman problem on n cities (TSP(n)) for $3 \leq |W| \leq n-3$, see Grötschel and Padberg (1979).

Let $H_1, H_2, \ldots, H_r, T_1, T_2, \ldots, T_k \subseteq V_n$ such that $(V_n, H_1, H_2, \ldots, H_r, T_1, T_2, \ldots, T_k)$ is a connected hypergraph. Call the node sets $H_1, H_2, \ldots, H_r$ **handles** and the node sets $T_1, T_2, \ldots, T_k$ **teeth**. The handles and the teeth define a **clique tree** in $K_n = (V_n, E_n)$ if 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 $n-4$ nodes;
(iv) each tooth contains at least one node not belonging to any handle;
(v) each handle intersects an odd number (≥ 3) of teeth;
(vi) if a tooth T and a handle H have a nonempty intersection, then $H \cap T$ is an articulation set of the clique tree, i.e., the removal of the nodes in $H \cap T$ from K_n disconnects the clique tree.

The associated **clique tree inequality**

$$\sum_{i=1}^{r} x(E_n(H_i)) + \sum_{j=1}^{k} x(E_n(T_j)) \leq \sum_{i=1}^{r} |H_i| + \sum_{j=1}^{k} (|T_j| - h(T_j)) - \frac{k+1}{2},$$

where $h(T_j)$ is the number of handles with nonempty intersection with tooth T_j, is facet defining for TSP(n) with $n \geq 11$, see Grötschel and Pulleyblank (1986). A clique tree with two handles and seven teeth is displayed in Figure 2.

A clique tree inequality with only one handle ($r = 1$) is called a **comb inequality** and was shown to be facet defining for TSP(n) with $n \geq 6$ by Grötschel and Padberg (1979). A comb inequality where each tooth has exactly one common node with the handle is called a **Chvâtal comb** and was discovered by Chvatal (1973). A comb inequality in which all teeth have cardinality 2 is also called a **2-matching inequality**.

Good LP-relaxations contain an enormous number of inequalities. For TSP(n) there are already $O(2^n)$ subtour elimination constraints. This makes it prohibitive to store

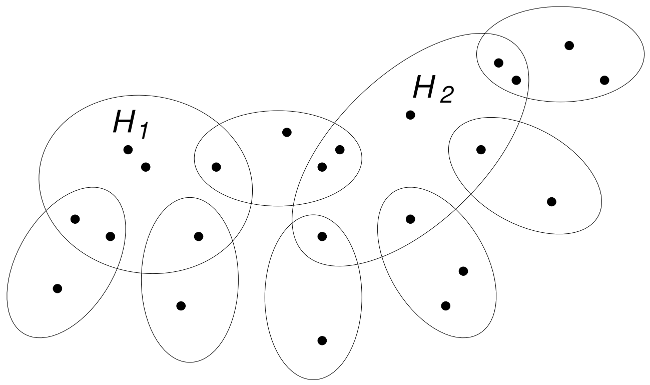

Figure 2. Handles and teeth of a clique tree.

the constraint matrix and apply an LP-algorithm. Rather, inequalities are produced in a **cutting plane approach** which starts with some initial relaxation, say, consisting of (1.1) and (1.3) and then adding known facet defining inequalities "according to need". More precisely, as soon as the optimal solution $\overline{x}$ of some relaxation is determined which is not the incidence vector of a tour, it is tried to identify by **separation algorithms** facet defining inequalities which are violated by $\overline{x}$. As long as such inequalities are found, they are added to the current relaxation and the new LP is solved, etc. If the optimal solution $\overline{x}$ violates some inequalities of a class of facets, an **exact separation algorithm** for a class of facets finds at least one of them. But if we apply a **heuristic separation algorithm** for a class of facets, no inequality might be found, even though there are constraints of this class violated by $\overline{x}$. Polynomial time exact separation algorithms are known for the subtour elimination inequalities (CROWDER AND PADBERG (1980)) and the 2-matching inequalities (PADBERG AND RAO (1982)). For comb and clique tree inequalities several heuristics have been proposed in the literature (GRÖTSCHEL AND HOLLAND (1991), PADBERG AND RINALDI (1990)).

Let us return to the flowchart of Figure 1. When we use a cutting plane approach to determine the local lower bounds the algorithm becomes a **branch & cut algorithm**. Requiring or forbidding edges in the definition of subproblems becomes setting the corresponding variables to 1 or 0, respectively. In the implementation to be described in the next section, the local lower bound and global upper bound computations are combined, since any fractional LP-solution occuring during the cutting plane procedure not only provides a local lower bound but also information that can be utilized to improve the shortest known tour by heuristic methods.

We close this section by giving a short historical review of the developments that eventually lead to our current notion of a branch & cut algorithm. 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 49-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 until the ideas of Dantzig, Fulkerson and Johnson were taken up 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 further development is due to the systematic study of the traveling salesman polytope P^n_{TSP}. Grötschel used the knowledge of this 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 tour. He needed 13 iterations of this process (see GRÖTSCHEL (1977, 1980)). Since the early eighties, then, 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 tours generated in the following way. An initial tour is determined by a heuristic described in LIN AND KERNIGHAN (1973), and the first linear programming problem is given by (1.1) and (1.3). The initial basis corresponds to the initial tour. 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 tour, the pivot is carried out, and the algorithm proceeds with the new tour. Otherwise, it is tried to identify a violated inequality which is satisfied with equality by the current tour but violated by the adjacent fractional solution. If such an inequality can be 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 tour. Out of 74 sample problems ranging from 15 to 318 cities, 54 problems could be solved to optimality this way. The whole computer code was written by the authors including an implementation of the simplex algorithm in rational arithmetic.

In the second paper, IBM's MPSX LP-software is used instead, and IBM's MPSX-MIP integer programming software is used to find the incidence vector of an optimum tour as follows. MIP is applied to the final LP to find an optimal integral solution. If this solution is the incidence vector of a tour, this tour is returned as the optimum solution. Otherwise the solution is necessarily a collection of subtours, and the corresponding subtour elimination constraints 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 the 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 (1988). 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. 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 ca 9 and 16 hours of CPU time, and the latter between ca 23 and ca 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 & bound software for integer programming could be used to find integer solutions. Only if such an integer solution contained subtours, the corresponding subtour elimination constraints were added to the LP-relaxation and the branch & bound part was started from scratch, again using a fixed linear programming relaxation in each node of the branch & bound tree.

On the other hand, the iterated "solving from scratch", whenever the addition of further subtour elimination constraints 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 & bound tree is not utilized. Furthermore, general purpose branch & 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 recursively 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 & 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 could not be solved with other available techniques.

The first state-of-the-art branch & 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 & 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.

Recently the solution of a 3038-node instance with an enhanced branch & cut algorithm is reported (Applegate, Bixby, Chvátal and Cook (personal communications, 1992)). However, no written report has been published yet.

2 Algorithm

This section discusses our implementation of a branch & cut algorithm for the TSP in detail. It resembles the implementation of PADBERG AND RINALDI (1991), but there are some differences:

- During the branch & cut algorithm, we exploit fractional LP solutions not only for getting lower bounds, but also for obtaining good tours.
- The subset of active variables is generated and managed in a different way.
- The pricing of nonactive variables is done in a hierarchical fashion.
- Separation of comb and clique tree inequalities is not as sophisticated.

We used the programming language C. The powerful builtin functions for dynamic memory allocation simplify the efficient use of memory space.

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. Up to this point, our notion of a branch & cut algorithm coincides with the definition which is common in the literature. The upper bounds give a new feature. They are obtained by exploiting fractional LP solutions in the construction of tours which are improved by heuristics.

The branch & cut algorithm for the TSP is outlined in the flowchart of Figure 3. 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.

But 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 **branch & cut tree**. Hence we call a subproblem also **branch & cut node**. We distinguish between three different types of branch & cut nodes. The node which is currently processed is called the **current branch & cut node**. The other unfathomed leaves of the branch & 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

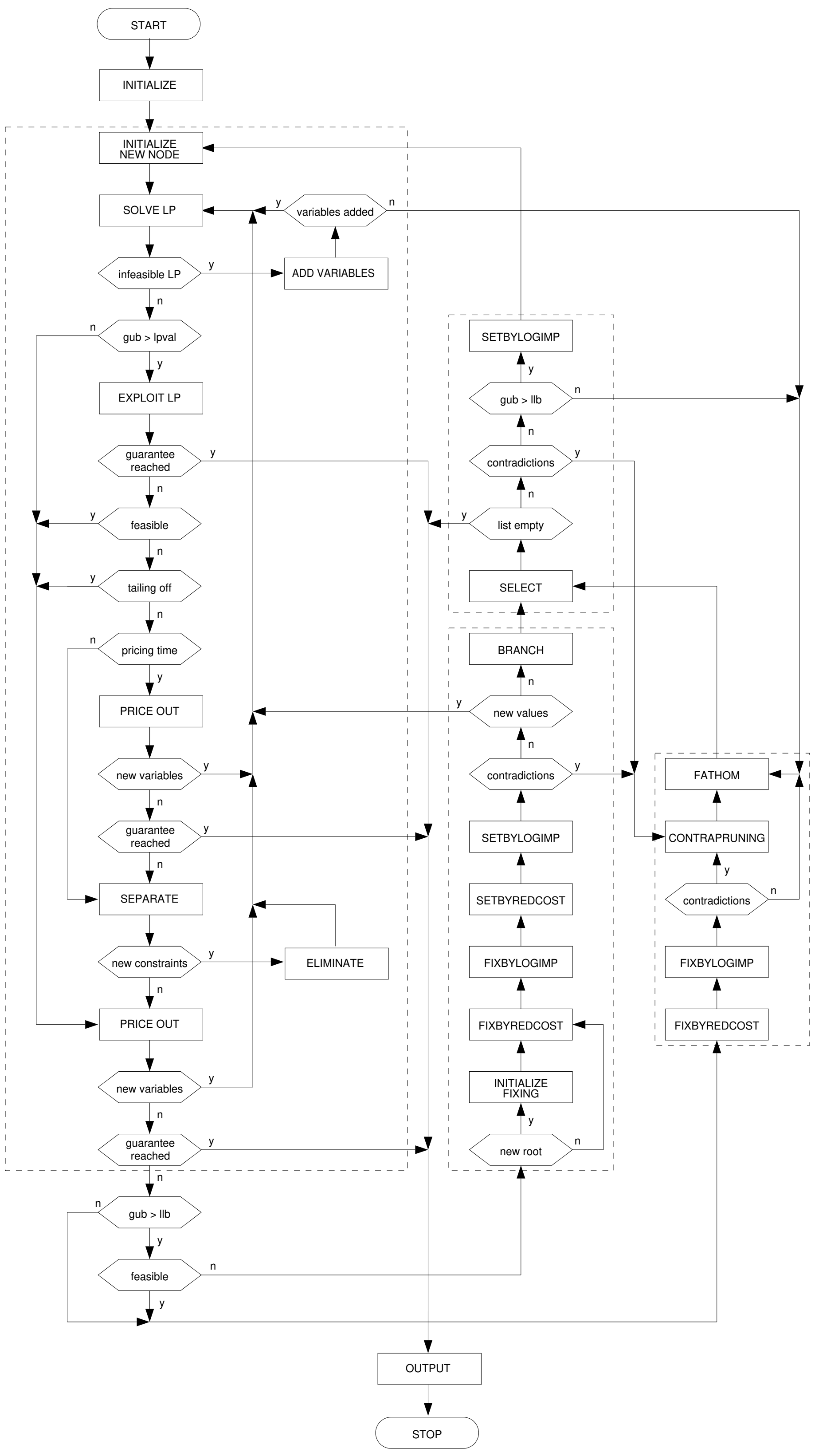

Figure 3. Flowchart of the branch & cut algorithm.

are used interchangeably, because they are in a one to one correspondence. Each variable (edge) has one of the following statuses during the computation: `atlowerbound`, `basic`, `atupperbound`, `settolowerbound`, `settoupperbound`, `fixedtolowerbound`, `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 & cut node and all branch & cut nodes in the subtree rooted at the current one in the branch & cut tree. The meanings of the other statuses are obvious: As soon as an LP has been solved, each variable which has not been fixed or set receives one of the statuses `atlowerbound`, `basic` or `atupperbound` by the revised simplex method with lower and upper bounds. Finally, the global variable `lpval` always denotes the optimal value of the last LP that has been solved, the global variable `llb` is a lower bound for the currently processed node, the global variable `gub` (global upper bound) gives the value of the currently best known tour. The minimal lower bound of all active branch & cut nodes and the current branch & 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 & 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 & cut tree** we denote the highest common ancestor in the branch & cut of tree of all branch & 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. Like in branch & bound terminology we call a subproblem **fathomed**, if the local lower bound `llb` of this subproblem is greater than 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 tour). 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 $\lceil \texttt{llb} \rceil \geq \texttt{gub}$. 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 1 with the dashed boxes of the flowchart of Figure 3. The upper bounding is done in EXPLOIT LP, the lowerbounding 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. The ordinary way of leaving the bounding part takes place if no variables are added by PRICE OUT at the end of the bounding part. 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 tour, if our guarantee requirement is fulfilled (`guarantee reached`), but we ignore this, if we want to find the optimum solution.

2.1 Enumerative frame

In this paragraph we explain our 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. Preliminary experience has already been obtained for the maximum planar subgraph problem (JÜNGER AND MUTZEL (1993)) and the sequential ordering problem by Ascheuer, Jünger and Reinelt.

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 already 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 optimum tour connect near neighbors. Often, optimum tours are contained in the 10-nearest neighbor subgraph of K_n. In any case, a very large fraction of optimum tour edges are already contained in the 5-nearest neighbor subgraph of K_n. Depending on two parameters k_s and k_r (the choice of k_s and k_r is discussed in section 3) we compute the k_s-nearest neighbor subgraph and augment it by the edges of a tour found by a simple heuristic so that the resulting **sparse graph** $G = (V, E)$ is Hamiltonian. Using this tour, 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. Afterwards the root node of the complete branch & cut tree is processed by the bounding part.

BOUNDING

The computation of the lower and upper bounds is outlined in the subsections 2.2 and 2.3. 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 constraints, all integral LP solutions are incidence vectors of tours, as soon as no more subtour elimination constraints are generated.

We check if the current branch & cut node cannot contain a better tour than the currently best known one ($\texttt{gub} \leq \texttt{llb}$). If this is the case, the current branch & cut node can be fathomed (rightmost column of the flowchart), and if no further branch & cut nodes have to be considered, the currently best known tour must be optimum (`list empty` after SELECT). Otherwise we have to check if the current LP-solution is already a tour. If this is the case (`feasible`) we can fathom the node, otherwise we prepare a branching operation and the selection of another branch & cut node for further processing (third column of the flowchart).

INITIALIZE FIXING, FIXBYREDCOST

If we are preparing a branching operation, and the current branch & cut node is the root node of the currently remaining branch & 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 $\texttt{rootlb} + r_e > \texttt{gub}$ and we can fix x_e to one if $x_e = 1$ and $\texttt{rootlb} - r_e > \texttt{gub}$.

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 & 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 & 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 statuses of the variables of this recomputation, but also the corresponding local lower bound as `rootlb` in the following calls of the routine FIXBYREDCOST. If we initialize the basis by the variables contained in the best known tour 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 & 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 & cut node to the root of the branch & 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 upon 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). Like 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 disregard it.

SETBYREDCOST

While fixings of variables are globally valid for the whole computation, variable settings are only valid for the current branch & cut node and all branch & cut nodes in the subtree rooted at the current branch & 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 & cut node, so that they can be undone when necessary. All set variables are inserted together with the branch & cut node into the hash table of the set variables, which is explained in subsection 2.4.

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). Like in SETBYREDCOST, all settings are associated with the current branch & cut node. If variables are fixed to new values, we proceed as explained in FIXBYREDCOST. As in SETBYREDCOST, the set variables are stored in the hash table.

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 & cut nodes, which are the two sons of the current branch & cut node, are created and added to the set of active branch & 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. Some strategies for choosing the branching variable are discussed in section 3.

SELECT

A branch & cut node is selected and removed from the set of active branch & cut nodes. We discuss different selection strategies in the next section. If the list of active branch & cut nodes is empty, we can conclude optimality of the best known tour. 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 & 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 & 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 or equal to the global upper bound `gub`, we fathom the node immediately and continue the selection process. A branch & cut node has pointers to his father and his 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 & 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 & 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 & cut nodes where e has been set. If in a branch & 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` exceeds the local lower bound `llb`, or a contradiction occured, or an infeasible branch & cut node has been generated, the current branch & 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 & cut tree for the remaining active nodes.

OUTPUT

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

2.2 Computation of lower 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 constraints** 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. Further details of the pool are outlined in subsection 2.4.

INITIALIZE NEW NODE

Let A_n be the adjacency matrix corresponding to the sparse graph. If the node is the root node of the branch & cut tree the LP is initialized to

$$\begin{aligned} \min \quad & c'x \\ \text{s.t.} \quad & A_n x = 2 \\ & 0 \leq x \leq 1 \end{aligned}$$

and the feasible basis obtained from the initial tour is used as a starting basis. The set of active branch & cut nodes is initialized as the empty set. In subsequent subproblems, we initialize the constraint matrix by the equations induced by the adjacency matrix of the sparse graph and by the inequalities which have been active, when the last LP of the father of the branch & cut has been solved. These inequalities can be regenerated from the pool. Since the 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. The columns of nonbasic set and fixed variables are removed from the constraint matrix and if their status is `settoupperbound` or `fixedtoupperbound`, the right hand side of the constraint has to be adjusted and the corresponding coefficients 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). As LP solver we use CPLEX by R.E. Bixby.

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 & 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, also this means that the branch & cut node is fathomed.

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 & cut node. So if `lpval` $\geq$ `gub`, we can fathom the branch & 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` $+ r_e \leq$ `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

$$x_b < 0 \text{ and } \overline{a}_e(b) < 0$$

  or

$$x_b > 1 \text{ and } \overline{a}_e(b) > 0$$

- x_b is a slack variable and

$$x_b < 0 \text{ and } \overline{a}_e(b) < 0.$$

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 described below in PRICE OUT.

If variables can be added, we regenerate the constraint structure and solve the new LP, otherwise we fathom the branch & 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 tour. If this is the case, the variable `feasible` is set to true. Otherwise, we try to improve the upper bound as discussed in 2.3. Afterwards we normally proceed with the generation of additional inequalities in SEPARATE.

We can leave the bounding part and output the best known solution if the guarantee requirements are satisfied.

Often it is reasonable to abort the cutting plane part, if no significant increase of `lpval` in the last LP-solutions has taken place. This phenomenon is called **tailing-off**. If during the last k iterations in the bounding part, `lpval` did not increase by more than $p\%$, we perform a pricing step and create two new subproblems, if no variables have been added, otherwise we resolve the LP. Tailing off is only possible, if there are fractional variables in the current optimal LP-solution to find a branching variable. This is guaranteed , e.g., if the current LP solution is not the incidence vector of a tour and no subtour elimination constraint is violated. Different parameters k and p are discussed in section 3, where we also show that it is advantageous to process a pricing step after every l-th call of the LP-solver (`additional pricing`). These two algorithmic details have already been used by Padberg and Rinaldi (1991). We also discuss the choice of the parameter l in section 3.

SEPARATE

The separation phase is a central part of our algorithm. We use hierarchical separation in three stages. The second and third stage are only executed if the respective previous stages did not generate any inequality. In the first stage we call the separation routines for subtour elimination and 2-matching constraints. The second stage is the pool separation. We check if an inactive comb or clique tree constraint is violated. Of course, we could do pool separation also with subtour elimination constraints and 2-matching constraints before calling the respective separation routines, but usually there are so many constraints in the pool that direct separation is more efficient than scanning the pool. However, the disadvantage of this strategy is that constraints might be stored more than once in the pool. In the final stage of the separation we try to identify violated comb and clique tree constraints. In the current implementation we use an exact subtour elimination constraint separator (Crowder and Padberg (1980)), a heuristic for the separation of 2-matching constraints of Padberg and Rinaldi (1990), which is based on the exact separation algorithm of Padberg and Rao (1982) and heuristics for separating comb constraints and clique tree constraints (based on ideas of Grötschel and Holland (1991)). The generated constraints are both stored in the pool and added to the current constraint matrix of the LP by the post-optimization routines of CPLEX.

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, in order to save memory and to accelerate the solution of the subsequent linear programs. We can safely do this to keep the constraint structure as small as possible, because we can restore the constraints from the pool (if they were not removed in the meantime).

PRICE OUT

Pricing is necessary before a branch & 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 tour.

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 optimum tour tend to be added to the sparse graph early in the computational process. We discuss the choice of k in section 3.

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 cost of all nonactive variables which takes a lot of computational effort. But this second step of PRICE OUT can be processed more efficiently by an idea of PADBERG AND RINALDI (1991). If our current branch & cut node is the root of the remaining branch & cut tree, we can check if the reduced cost r_e of a nonactive variable e satisfies the relation $\texttt{lpval} + r_e > \texttt{gub}$. 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 2.4, provides us with an efficient way to compute the constraint lists and the coefficients.

2.3 Computation of global upper bounds

The cutting plane part together with the enumeration scheme provides an algorithm that is capable of solving a traveling salesman problem instance in "finite time". But, not even a rough estimation on the necessary running time for a particular instance can be given. The size of a problem is only an insufficient characterization of its hardness.

According to our philosophy, an algorithm, that provides successively improving lower bounds on the objective function value of an optimum solution and only guarantees that eventually an optimum solution is found, is not adequate for practical problem solving. In fact, it may even turn out to be useless. What is required in practice is in addition that, on the first hand, a reasonable feasible solution is given quickly and that, on the second hand, better solutions become available as more running time is spent.

For the traveling salesman problem a host of heuristics is available. Usually, they are employed independent of lower bound computations. They are used to give a good feasible solution before a branch & bound algorithm is started. Then it is left to the branch & bound algorithm to find further tours. We take a different point of view in our approach. Rather than running heuristics in an isolated way we think that the fractional LP solutions occuring in the lower bound computations give hints on the structure of optimum or near optimum tours. We therefore prefer an algorithmic framework that integrates upper and lower bound computations. As we shall see below there are also influences of the tour heuristics to the lower bound part.

The implementation we present here is designed for a sequential computer. The basic requirement for the upper bound computations is therefore efficiency in order not to inhibit the optimization process. While in the first stages high emphasis is laid on providing good tours, this emphasis is less in the later stages of the computation process. On the other hand, computing upper bounds can always be reasonable since new knowledge about the structure of optimum tours is acquired (e.g. because of fixed and set variables in the branch & cut tree).

We discuss the various aspects of using heuristics to find good tours based on the information in LP solutions coming up in the algorithm. In our flowchart the corresponding computations are performed in EXPLOIT LP, some initializations are done in INITIALIZE.

Candidate subgraph

The basic idea that we apply for our heuristics 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 good tour. These edges are taken with priority in the various heuristics, thus avoiding the consideration of the majority of edges which are assumed to be of no importance. Various candidate subgraphs and the question of how to compute them efficiently are discussed in Reinelt (1992) and Jünger, Reinelt & Rinaldi (1992).

In the present context we relate the candidate subgraph to the set of active variables in the linear programming problems. Basically, we start with some candidate subgraph (which may consist of the sparse graph used to initialize the cutting plane part, or the sparse graph enhanced by the reserve graph, or just the empty graph) and then add edges whose corresponding values are close to one. In order to avoid too extensive growing of the candidate subgraph and to avoid being biased by LPs that were not recently solved, we clear the candidate subgraph in certain intervals (e.g. every 20th cutting plane phase) and reinitialize it.

Exploiting the LP solution

Integer optimal solutions, i.e., incidence vectors of tours, will almost never result from the LPs occuring in the branch & bound algorithm. But, as can be seen on a graphics display, these solutions, although having many fractional components, give information on good tours. They have a certain number of variables equal to 1 and also a certain number of variables whose values are close to 1. We exploit this to form a starting tour for subsequent improvement heuristics as follows.

First, we check if the current LP solution is the incidence vector of a tour. If this is the case, the variable `feasible` is set to true and we proceed with the pricing routine. Otherwise, edges are sorted according to their values in the current LP solution. We give decreasing priorities to edges as follows:

- edges that are fixed or set to 1,
- edges equal to 1 or close to 1 in the current LP,
- edges occuring in several successive LPs.

This list is scanned and edges become part of the tour if they do not produce a subtour with the edges selected so far. This gives a system of paths which now have to be connected. To this end a savings heuristic (Clarke and Wright (1964)), originally developed for vehicle routing problems, is used. It can be applied here since the traveling salesman problem can be considered as a special vehicle routing problem involving only one vehicle.

This heuristic basically consists of successively merging partial tours to eventually obtain a Hamiltonian tour. In our case we proceed as follows. We select one node as base node and form partial tours by connecting this base node to the end nodes of each of the paths obtained in the selection step and also adding a pair of edges to nodes not contained in any path. Then, as long as more than one subtour is left, we compute for every pair of tours the savings that is achieved if the tours are merged by deleting in each tour an edge to the base node and connecting the two open ends. The two tours giving the largest savings are merged. In our implementation, we try to avoid edges for connecting paths that are fixed or set to 0 in the branch & cut tree.

Some remarks concerning the implementation are in order. The crucial point is the update of the minimum merge possibilities. We can consider the system of tours as a system of paths 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 tours). This best possibility may change if two tours are merged. 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.

For large problem instances and in particular in our framework, we can neither afford $O(n^3)$ nor $O(n^2 \log n)$ running time. We therefore make use of the candidate set. Namely, merge operations are preferred that use a candidate edge for connecting 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 merge possibility by enumeration.

Improving the first solution

Having applied the savings heuristic we try to improve this tour by local modifications. Here we use variants of the 3-opt heuristic and of the Lin-Kernighan heuristic.

A move of the 3-opt heuristic consists of removing three edges from the current tour and reconnecting the three resulting paths in the best possible way. The number of such 3-opt moves is $\binom{n}{3}$ and there are eight ways to connect three paths to form a tour (if each of them contains at least one edge). The simpler node insertion, edge insertion, and 2-opt exchange are special 3-opt moves. Node (edge) insertion is obtained if one path of the 3-opt move consists of just one node (edge). A 2-opt move is a 3-opt move where one eliminated edge is used again for reconnecting the paths. Out of the eight 3-opt moves only four are real 3-opt moves (three moves are just 2-opt moves and one move leaves the tour unchanged). To examine all 3-opt moves whether they can contribute to decrease the tour length takes time $O(n^3)$, and is therefore infeasible in our situation. Tour update after a 3-opt move is also complicated because the direction of the tour may change on all but the longest of the three involved paths. We have therefore limited the general 3-opt procedure by requiring that at least one candidate edge is introduced by a 3-opt move.

The motivation for the Lin-Kernighan heuristic (LIN AND KERNIGHAN (1973)) is based on experience gained from practical computations. Namely, one observes that the more flexible and powerful the possible tour modifications are, the better results are obtained and that simple moves quickly run into local optima of only moderate quality that cannot be left anymore. The natural consequence of simply applying k-opt for larger k cannot be realized due to increasing running time. A complete check of the existence of an improving k-move takes time $O(n^k)$. One can, of course, design restricted searches for higher values of k in the same way as we did for $k = 3$. It is more powerful, however, to follow the approach suggested by Lin and Kernighan. Their idea is based on the observation that sometimes a modification slightly increasing the tour length can open up new possibilities for achieving considerable improvement afterwards. The basic principle is to build complicated tour modifications that are composed of simple moves where not all of these moves necessarily have to decrease the tour 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 apply the original version of this algorithm which contains a 3-opt component, but use a somewhat simpler version where the basic components are 2-opt and node insertion moves. There are no significant differences since we also have the usual

3-opt exchange at hand.

Further details about these heuristics and their computational performance can be found in Reinelt (1992) and Jünger, Reinelt & Rinaldi (1992).

If the final tour has smaller cost than the currently best known one, it is made the incumbent solution and `gub` is updated.

Strategies for employing the heuristics

Since we are running our algorithm on a sequential machine we must take care of a proper distribution of CPU time between lower bounding and upper bounding part. In addition, the amount of work that is spent in the heuristics has to be controlled. Various strategies are possible.

- Fixed percentage of CPU time

  We specify in advance a certain percentage of CPU time that is spent for upper bound computations. Whenever, after having solved an LP, we have not reached this percentage then we initiate an upper bound computation. This strategy has the disadvantage that it is not flexible and can miss LP solutions that would lead to an improvement of the current tour. We observed that there is a significant difference among LP solutions with respect to suitability for our heuristic approach.

- Fixed iteration number

  In this case we would start an upper bound computation whenever a certain number of LPs is solved. This strategy has the same disadvantage as the previous one and, in addition, does not take care of the CPU time spent for the heuristics.

- Dynamic strategy

  Here we try to specify some guidelines for increasing the chance that an upper bound computation is promising and should be initiated. In any case, after every LP the candidate subgraph is updated, i.e., every active variable whose value is above some certain threshold in the current LP (for example at least 0.6) is added to the candidate set, and after a certain number of iterations the candidate set is reinitialized. This way, important variables introduced by pricing can also enter the candidate set. Also, after every LP, we perform the savings heuristic to exploit the LP solution. We avoid trying to improve the same tour several times by using a hashing scheme to detect identical tours. As hash key we use the length of the tour and the name of the heuristic which is applied to this tour. We also inhibit the improvement heuristic if the starting solution is much inferior than the best solution found so far, because we anticipate in this case that our limited heuristics will not yield a better solution within moderate time limits. Depending on the progress of the improvement heuristics we also decide how much effort is spent in this part. For example, if we come close to the length of the best known solution very fast, then we spent more CPU time and extend the modification possibilities of the Lin-Kernighan heuristic and the 3-opt exchange. If progress is slow, then we terminate improvement early.

Feedback to the cutting plane part

It should be noted that the tours found by the heuristics are not restricted to only using edges of the candidate set. These edges are only considered with priority and lead to an acceptable CPU time. Usually, heuristics will introduce edges that are not active in the LP. We add these edges to the set of active variables. This is based on the assumption that these edges are also important for the lower bound computations and would be added to the LP in the pricing step anyway. This way we augment the set of active variables without pricing.

We will report on computational aspects in the final section of this report.

2.4 Data Structures

A CPU time and memory sensitive implementation of data structures is crucial for an efficient branch & cut algorithm. Our general design philosophy is to rather spend more memory space than to inhibit improvements in running time. Nevertheless some global data structures have to be implemented very carefully.

Sparse graph

Essential for efficient TSP solving by a branch & cut algorithm is the use of sparse graph techniques. We select in INITIALIZE only a very small subset of the edges for our computations: the set of active edges. This set is augmented by additional edges if either this is necessary to guarantee correctness of the lower bounds on the complete graph by PRICE OUT or ADD VARIABLES, or if an upper bounding method in EXPLOIT LP considers some nonactive edges useful.

For the representation of the sparse graph we have to choose some format which saves memory and enables us to perform efficiently 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. We store the sparse graph in the six arrays `tail`, `head`, `nxtt`, `nxth`, `frst`, `cost`.

`tail[e]` is the tail of edge `e`.
`head[e]` is the head of edge `e`.
`nxtt[e]` is the next edge incident to `tail[e]`.
`nxth[e]` is the next edge incident to `head[e]`.
`cost[e]` is the cost of edge `e`.
`frst[v]` is the first edge incident to node `v`.

All incidence lists terminate with a zero. If we initialize `frst[v] = 0` for all nodes `v`, it is easy to add an edge, if `nvar` variables have already been inserted into the sparse graph. This is achieved by the following lines of C-code.

```
nxtt[nvar] = frst[t]; frst[t] = nvar; tail[nvar] = t;
nxth[nvar] = frst[h]; frst[h] = nvar; head[nvar] = h;
nvar++;
```

The next program fragment shows how all adjacent nodes and all incident edges of a node v can be scanned.

```
incidentedge = frst[v];
while (incidentedge) {
  if (v == tail[incidentedge]) {
    adjacentnode = head[incidentedge];
    incidentedge = nxtt[incidentedge];
  }
  else {
    adjacentnode = tail[incidentedge];
    incidentedge = nxth[incidentedge];
  }
}
```

For many data structures, e.g., some of the arrays of the sparse graph, we do not know the actual required size at run time in advance. So we initially allocate memory, which is sufficient according to our experience and extend it, if necessary.

Branch & cut nodes

Although a subproblem is completely defined by the fixed variables and the variables that were set temporarily, it is necessary to store additional information at each node for an efficient implementation. Every branch & cut node has pointers to its father and sons. A branch & cut node contains the arrays `set` of its set variables and `setstat` with the corresponding statuses (`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 & 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 & 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 the TSP-polytope. 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 & 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 statuses 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, which 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 & 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 (1990)) 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 & 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 and SETBYLOGIMP. In CONTRAPRUNING it is essential to determine efficiently all nodes where a certain variable is set. To avoid scanning the complete branch & 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 & 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)).

Contradictions occur quite rarely in the case of the traveling salesman problem, yet CONTRAPRUNING might become more important for other problems.

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. Furthermore 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 TSP(n) can range from four entries for a subtour elimination constraint to about $2n$ entries for a comb constraint or a clique tree constraint.

A subtour elimination constraint 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 constraints, comb constraints and clique tree constraints 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 constraint 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 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 constraint 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 constraint 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 constraints and clique tree constraints. 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 & 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.

3 Computational experiments

We have performed various computational tests in order to evaluate the practical performance of our implementation. The parameters and strategies mentioned in the previous section are tested in subsection 3.1. In subsection 3.2 we show computational results for all instances of TSPLIB, which have between 100 and 4461 cities. For all of these problems we give solutions with a prespecified guarantee and prespecified time bounds.

All computational experiments were carried out on a SUN SPARCstation 2 with 64 MB main memory under the operating system SunOS 4.1.2. We used the GNU C compiler with optimization option O2.

3.1 Strategies and parameters

To test different parameters and strategies we chose a test base of 17 problems from TSPLIB and tried to solve these instances to optimality. We selected these problems by the following criteria. First, the instances should have more than 100 cities, because all smaller instances are very easy to solve except for `pr76`. Second, the problems should be of different size and structure. Third, the selected problems should be solvable in reasonable time with our branch & cut code, because we wanted to perform a very large number of tests. Of course, we cannot guarantee that the results would be the same for another test set, especially if it consists of bigger instances. But we think that the results of these experiments provide good base for further developments of branch & cut algorithms. In addition to the obvious performance measure in terms of CPU time (in minutes:seconds, abbreviated by T in the tables of this section) we also give for each run of an instance the number of generated branch & cut nodes not including the root node (abbreviated by N in the tables of this section). So the actual number of branch & cut nodes is N + 1. For every tested parameter or strategy we chose a default value which is given in the respective subsection. For the experiments with a single strategy or parameter all other strategies and parameters kept their default value.

3.1.1 Enumeration strategies

There are three well-known enumeration strategies in branch & bound (branch & cut) algorithms: depth-first search (DFS), breadth-first search (BRFS) and best-first search (BEFS). We define the **level** of a branch & cut node B as the number of edges on the path from the root of the branch & cut tree to B. In case of depth-first search a branch & cut node with maximum level in the branch & cut tree is selected from the set of active nodes in SELECT, whereas in breadth-first search a subproblem with minimum level is selected. In best-first search the "most promising" node becomes the current branch & cut node. We think that the node with minimal local lower bound among all active nodes is most promising. If in any of these three enumeration strategies two nodes have the same priority, we select one for which the branching variable of its father is set to one, because setting a variable to one has more influence on the structure of a subproblem than setting a variable to zero.

The results presented in Table I show that depth-first search is the worst enumeration strategy, because the "risc" of spending a lot of time in a branch of the tree, which is useless for computing upper and lower bounds, is very high. We have observed that often the local lower bound of the current subproblem exceeds the length of an optimum tour, however, this node cannot be fathomed, because no good upper bound is known. The same phenomenon occurs also sometimes when using breadth-first search, but it is very rare if we enumerate in best-first search order. Therefore we chose best-first search as default strategy.

In order to get more insight into the performance of our bounding part and the effi-

Table I
Enumeration strategies

Problem	BEFS T	BEFS N	BRFS T	BRFS N	DFS T	DFS N	GUB T	GUB N
gr120	0:14	2	0:14	2	0:14	2	0:12	2
bier127	0:17	0	0:17	0	0:17	0	0:10	0
pr152	2:21	24	1:57	24	2:04	24	4:03	28
rat195	6:52	56	3:58	24	5:34	40	3:33	24
d198	4:11	42	3:46	36	6:48	62	2:34	20
gr229	15:19	68	21:45	78	15:46	66	10:29	54
gil262	5:30	12	5:46	26	8:32	36	3:34	8
pr299	77:59	200	99:09	360	210:13	982	70:35	272
lin318	6:28	16	3:26	8	3:40	8	3:47	6
rd400	72:33	162	57:15	200	103:16	292	41:26	128
pr439	250:06	378	251:32	368	2590:31	4642	152:44	278
d493	88:09	60	117:05	212	1955:53	5014	84:09	66
att532	182:54	180	249:08	324	396:29	520	115:18	126
ali535	28:30	8	37:07	18	88:30	78	28:41	14
p654	11:52	34	7:31	22	13:10	34	13:39	66
gr666	148:46	70	115:57	76	213:10	174	49:30	22
rat783	25:28	12	49:03	44	30:10	24	21:10	6
Sum	927:29	1324	1025:00	1822	5644:22	11998	605:34	1120

ciency of the enumeration strategies we carried out the following experiment. We initialized the variable global upper bound gub with the length of the known optimum tour. Now the only task of the branch & cut algorithm was to prove optimality. The results are presented in the last two columns of Table I (GUB). It is obvious that in this case the enumeration strategy has no direct influence on the running time. The number of branch & cut nodes gives us an approximate lower bound on the number of branch & cut nodes in any enumeration strategy in some sense. Our experiments show that this is only an approximation for this lower bound, since in this experiment no variables are added to the sparse graph by the upper bounding part, which influences the cutting plane generation procedures. For this experiment the computation of upper bounds was turned off.

3.1.2 Separation strategies

Even though the separation of violated inequalities seems to be a very small part of the algorithm (just the one box SEPARATE of the flowchart of Figure 3), it is the most important part. Only further improvements of the separation algorithms, i.e., implementation of an exact separation method for 2-matching constraints, better separation heuristics for comb and clique tree constraints and the development of separation algorithms for other known facets of the traveling salesman polytope, can lead to a breakthrough to the next order of magnitude of instances, which can be solved to optimality or for which better lower bounds for the length of an optimum tour can be computed.

In Table II we compare the results achieved by the separation strategy outlined in subsection 2.2 (1) with an implementation without the separation of comb constraints and clique tree constraints from the pool (2) and with another implementation which does not generate comb constraints and clique tree constraints at all (3). Already the lack of pool separation made the performance worse, but without the separation of comb constraints and clique tree constraints some problems could not be solved (in reasonable time). The default strategy is therefore strategy (1).

Table II
Separation strategies

Problem	(1) T	(1) N	(2) T	(2) N	(3) T	(3) N
`gr120`	0:14	2	0:14	2	0:13	2
`bier127`	0:17	0	0:17	0	0:15	2
`pr152`	2:21	24	2:10	26	1:27	26
`rat195`	6:52	56	6:40	56	5:33	64
`d198`	4:11	42	3:17	30	3:54	112
`gr229`	15:19	68	24:37	84	40:19	344
`gil262`	5:30	12	7:19	22	26:32	386
`pr299`	77:59	200	173:46	580	859:59	7242
`lin318`	6:28	16	5:42	10	2:58	8
`rd400`	72:33	162	100:57	252	496:09	3916
`pr439`	250:06	378	262:06	438	—	—
`d493`	88:09	60	224:03	148	—	—
`att532`	182:54	180	664:10	656	—	—
`ali535`	28:30	8	41:48	18	98:15	136
`p654`	11:52	34	11:50	34	627:29	5682
`gr666`	148:46	70	116:28	44	398:58	466
`rat783`	25:28	12	38:02	30	774:03	1866
`Sum`	927:29	1324	1683:26	2430	3336:04	20252

3.1.3 Sparse graph

As the sparse graph we use the k_s-nearest neighbour graph and add a tour to guarantee that the resulting graph (denoted by $k_s$nn in the tables) is Hamiltonian. The parameter k_r for the reserve graph is always given by $k_r = k_s + 5$. The results for $k_s = 2, 3 \ldots, 10, 20$ of Table III and Table IV show that different sparse graphs led to different performances of the algorithm for a single problem but did not influence the sum of the CPU times of all 17 instances significantly for k_s less or equal than 10. The reason is that after several calls of the routines PRICE OUT, FIXBYREDCOST, SETBYREDCOST we automatically get an appropriate set of variables. However, if the initial sparse graph was too large, the number of those active variables, which were neither fixed nor set until to the termination of the algorithm, was very large (see Table V). Hence the solution of the LPs, the computation of upper bounds and the separation of violated inequalities was slowed down. As default value for k_s we chose 5.

3.1.4 Tailing-off

We try to leave the bounding part if during the last k solved LPs the local lower bound `llb` did not increase by at least p%. First we tested four different parameters $p = 10^{-1}$ (1), $p = 2.5 \cdot 10^{-3}$ (2), $p = 10^{-3}$ (3) and $p = 10^{-8}$ (4) for fixed $k = 6$ (see Table VI). For even smaller values of p we got the same results as for $p = 10^{-8}$. The default value is $p = 2.5 \cdot 10^{-3}$.

Table III
Sparse graph (part 1)

Problem	2nn T	2nn N	3nn T	3nn N	4nn T	4nn N	5nn T	5nn N	6nn T	6nn N
gr120	0:19	0	0:15	0	0:18	0	0:14	2	0:18	0
bier127	0:19	0	0:19	0	0:18	0	0:17	0	0:18	0
pr152	3:38	16	3:01	34	3:26	34	2:21	24	3:17	24
rat195	4:55	34	10:58	82	4:33	28	6:52	56	6:56	50
d198	3:08	26	2:57	26	2:51	32	4:11	42	2:40	26
gr229	16:46	56	22:15	84	26:27	68	15:19	68	15:21	58
gil262	9:04	30	5:21	18	9:32	38	5:30	12	3:25	8
pr299	86:18	288	71:59	170	86:13	260	77:59	200	60:40	182
lin318	5:10	12	4:02	8	4:32	8	6:28	16	6:32	16
rd400	73:40	180	49:09	142	48:40	132	72:33	162	81:25	190
pr439	135:51	168	141:47	228	173:03	260	250:06	378	281:15	418
d493	100:21	70	78:09	56	118:04	68	88:09	60	86:34	50
att532	295:22	264	256:36	258	418:30	384	182:54	180	211:10	182
ali535	47:41	18	35:25	12	47:11	14	28:30	8	89:11	22
p654	14:24	42	18:52	88	11:23	28	11:52	34	3:16	2
gr666	131:34	44	97:38	46	108:46	56	148:46	70	155:17	68
rat783	21:58	8	19:44	8	34:37	28	25:28	12	38:28	16
Sum	950:28	1256	818:27	1260	1098:24	1438	927:29	1324	1046:03	1324

Table IV
Sparse graph (part 2)

Problem	7nn T	7nn N	8nn T	8nn N	9nn T	9nn N	10nn T	10nn N	20nn T	20nn N
gr120	0:16	0	0:17	0	0:27	4	0:24	6	0:33	6
bier127	0:18	0	0:20	0	0:20	0	0:20	0	0:22	0
pr152	2:56	24	2:28	24	2:09	14	4:04	40	5:00	10
rat195	9:43	70	11:01	84	7:05	54	10:55	84	11:31	54
d198	3:43	32	4:16	38	3:58	44	3:38	26	8:09	28
gr229	18:37	72	20:11	74	18:54	68	35:58	138	26:27	68
gil262	5:53	14	5:59	16	10:13	26	7:57	10	11:57	20
pr299	64:14	180	96:22	238	120:57	326	106:34	238	135:58	248
lin318	5:24	12	6:17	12	10:19	16	6:06	12	8:36	12
rd400	68:52	194	74:01	184	56:05	108	68:01	130	105:02	132
pr439	164:48	218	166:52	202	185:52	256	150:42	192	186:44	164
d493	69:33	48	103:59	64	157:13	106	103:53	64	119:41	52
att532	216:14	186	482:52	344	380:49	314	366:34	272	285:23	164
ali535	40:52	14	34:44	10	53:38	14	53:27	8	82:34	10
p654	134:48	610	10:44	34	13:50	28	53:53	172	5:43	0
gr666	176:17	68	136:20	44	163:51	86	163:52	66	179:46	52
rat783	36:22	20	34:13	28	39:31	24	33:41	28	62:08	18
Sum	1018:50	1762	1190:56	1396	1225:11	1488	1169:59	1486	1235:34	1486

In a second experiment we tried to abort the computation of upper and lower bounds for a subproblem if during the last k solved LPs the local lower bound llb did not increase by at least $2.5 \cdot 10^{-3}\%$ for $k = 3, 6, 10, 20, 100$ (see Table VII). The default value is $k = 6$.

In contrast to Padberg and Rinaldi (1991) we could not observe any significant influence of the parameters p and k for tailing-off on the running time. However, we still use tailing-off in our implementation as "emergency exit" of the cutting plane phase, since the behaviour of the separation algorithms is not predictable. Better separation techniques, as used by Padberg and Rinaldi (1991), may make tailing-off more important.

Table V
Number of active variables

Problem	2nn	3nn	4nn	5nn	6nn	7nn	8nn	9nn	10nn	20nn
gr120	454	323	345	215	460	521	590	184	184	212
bier127	419	437	475	527	581	640	722	797	877	1744
pr152	647	678	664	697	707	675	680	683	773	732
rat195	502	551	535	550	646	631	704	701	704	748
d198	504	527	591	589	595	711	744	780	813	617
gr229	753	627	654	699	737	794	903	928	956	1017
gil262	672	636	654	692	674	694	691	709	722	782
pr299	992	1000	1002	1026	1032	974	1304	1495	1337	1814
lin318	676	719	733	782	803	764	781	763	771	775
rd400	1123	1245	1226	1381	1360	1391	1568	1724	1772	1804
pr439	1676	1643	1793	1858	2008	2102	2329	2566	2735	4282
d493	1483	1414	1481	1515	1652	1966	1105	1968	2079	2098
att532	1907	1541	1674	1795	2084	2212	2629	2873	3250	3862
ali535	1547	1456	1505	1521	1561	1815	1913	1876	2020	2793
p654	1309	1365	1256	1900	2169	2425	2079	2784	2990	7746
gr666	2223	1767	1895	2054	2331	2759	3000	3370	3375	4304
rat783	1547	1589	1702	1562	1776	1569	1890	1664	1890	1783
Sum	18434	17518	18185	19363	21176	22643	23632	25865	27248	37113

Table VI
Tailing-off, variation of p

Problem	(1) T	(1) N	(2) T	(2) N	(3) T	(3) N	(4) T	(4) N
gr120	0:14	2	0:14	2	0:14	2	0:14	2
bier127	0:17	0	0:17	0	0:17	0	0:17	0
pr152	2:21	24	2:21	24	2:21	24	2:21	24
rat195	6:52	56	6:52	56	6:52	56	6:52	56
d198	4:11	42	4:11	42	4:11	42	4:11	42
gr229	15:21	68	15:19	68	15:21	68	15:21	68
gil262	5:30	12	5:30	12	5:30	12	5:30	12
pr299	110:49	354	77:59	200	77:56	200	77:58	200
lin318	4:35	10	6:28	16	6:28	16	6:28	16
rd400	78:34	162	72:33	162	72:33	162	72:33	162
pr439	208:06	286	250:06	378	250:11	378	250:11	378
d493	88:10	60	88:09	60	88:11	60	88:13	60
att532	146:20	152	182:54	180	183:43	180	168:07	166
ali535	37:11	18	28:30	8	28:35	8	28:27	8
p654	11:52	34	11:52	34	11:52	34	11:52	34
gr666	161:14	78	148:46	70	162:53	82	162:52	82
rat783	36:33	18	25:28	12	25:32	12	25:25	12
Sum	918:10	1376	927:29	1324	942:40	1336	926:52	1322

3.1.5 Additional pricing

After every l LP-solutions during the computation of local lower bounds for a branch & cut node we perform an additional pricing step. In Table VIII results are given for $l = 1, 2, 5, 10, \infty$.

It turned out that additional pricing did not influence the running time of our implementation. However, there are three reasons why we still perform additional pricing steps. First, if no variable is added to the sparse graph after PRICE OUT, the global lower bound may change and hence the guarantee requirements may be reached, if no

Table VII
Tailing-off, variation of k

Problem	3 T	3 N	6 T	6 N	10 T	10 N	20 T	20 N	100 T	100 N
gr120	0:14	2	0:14	2	0:14	0	0:14	0	0:14	2
bier127	0:17	2	0:17	0	0:18	0	0:17	0	0:17	0
pr152	4:37	68	2:21	24	2:27	24	2:29	24	2:25	24
rat195	5:07	38	6:52	56	6:10	40	6:21	40	6:10	40
d198	3:43	52	4:11	42	4:37	42	4:31	42	4:30	42
gr229	20:37	86	15:19	68	16:32	68	17:30	82	18:04	66
gil262	7:08	30	5:30	12	9:46	36	9:49	36	9:40	36
pr299	153:41	436	77:59	200	79:03	200	81:03	200	60:28	170
lin318	4:13	12	6:28	16	6:36	16	6:41	16	6:39	16
rd400	56:37	176	72:33	162	74:09	162	75:34	162	73:05	162
pr439	183:58	316	250:06	378	257:16	378	258:33	378	151:32	270
d493	71:04	98	88:09	60	94:23	60	64:37	50	64:53	46
att532	207:28	298	182:54	180	174:03	166	221:55	206	356:02	306
ali535	41:46	40	28:30	8	30:32	8	57:58	18	36:49	14
p654	12:19	34	11:52	34	12:21	34	12:19	34	10:58	38
gr666	103:24	84	148:46	70	178:30	82	178:06	82	132:14	56
rat783	48:35	56	25:28	12	26:12	12	25:43	12	25:19	12
Sum	924:48	1828	927:29	1324	973:09	1328	1023:40	1382	959:19	1382

computation to optimality is desired. Second, if better separation algorithms are used, the number of branch & cut nodes decreases significantly. Often a problem can be solved (without branching) in the root node of the branch & cut tree. In this case additional pricing steps may be quite helpful. Third, the time necessary for PRICE OUT is quite small, i.e., between 1% and 2% of the total running time if we call PRICE OUT every 5 solved LPs. Therefore $l = 5$ is the default parameter.

Table VIII
Additional pricing

Problem	1 T	1 N	2 T	2 N	5 T	5 N	10 T	10 N	∞ T	∞ N
gr120	0:18	6	0:18	6	0:14	2	0:18	2	0:14	2
bier127	0:16	0	0:18	0	0:17	0	0:17	0	0:16	0
pr152	3:09	40	2:21	20	2:21	24	2:26	32	1:53	22
rat195	7:29	46	7:44	54	6:52	56	12:54	126	7:06	46
d198	3:28	24	4:13	36	4:11	42	3:27	22	4:32	32
gr229	17:36	50	17:16	58	15:19	68	19:48	88	16:21	58
gil262	6:16	18	9:07	32	5:30	12	5:32	20	9:06	46
pr299	59:08	146	71:27	184	77:59	200	120:23	296	114:21	384
lin318	7:00	14	6:51	16	6:28	16	3:40	10	6:59	16
rd400	61:42	170	51:53	114	72:33	162	77:37	190	60:40	154
pr439	184:26	246	168:47	222	250:06	378	152:21	212	176:33	254
d493	140:05	92	100:39	92	88:09	60	79:04	96	100:00	132
att532	255:17	194	169:59	142	182:54	180	177:14	178	256:57	260
ali535	58:34	14	36:26	16	28:30	8	34:53	24	50:11	28
p654	11:07	42	8:51	30	11:52	34	14:40	48	9:10	18
gr666	122:54	48	113:11	36	148:46	70	105:28	56	111:40	56
rat783	44:41	24	64:52	60	25:28	12	26:58	24	30:48	28
Sum	983:26	1174	834:13	1118	927:29	1324	837:00	1424	956:47	1424

3.1.6 Selection of the branching variable

We tested 4 different strategies for the selection of the branching variable. Let $\overline{x}$ be the solution of the last solved LP.

(1) Select a variable with value close to 0.5 which has a big objective function coefficient in the following way. Find L and H with $L = \max\{\overline{x}_e \mid \overline{x}_e \leq 0.5, e \in E\}$ and $H = \min\{\overline{x}_e \mid \overline{x}_e \geq 0.5, e \in E\}$. Let $C = \{e \in E \mid 0.75L \leq \overline{x}_e \leq H + 0.25(1 - H)\}$ be the set of variables with value "close" to 0.5. From the set C we select the variable with maximum cost, i.e., with maximum objective function coefficient. This method is similar to the one given in PADBERG AND RINALDI (1991).

(2) Select the fractional variable which has maximum objective function coefficient.

(3) If there are fractional variables which belong to the currently best known tour, select the one with maximum cost of them, otherwise, apply strategy (1).

(4) Select that fractional variable which is closest to one, i.e., find an edge $e^\star$ with $\overline{x}_{e^\star} = \max\{\overline{x}_e \mid \overline{x}_e \leq 0.999\}$.

Table IX
Selection of the branching variable

Problem	(1) T	(1) N	(2) T	(2) N	(3) T	(3) N	(4) T	(4) N
gr120	0:14	2	0:14	0	0:14	2	0:19	8
bier127	0:17	0	0:17	0	0:17	0	0:17	0
pr152	2:21	24	3:13	34	2:10	24	46:12	1102
rat195	6:52	56	10:49	90	10:25	82	30:10	388
d198	4:11	42	3:31	26	3:00	36	14:31	118
gr229	15:19	68	19:40	102	18:48	76	348:39	2522
gil262	5:30	12	7:30	22	18:24	102	25:15	208
pr299	77:59	200	160:56	624	193:03	966	—	—
lin318	6:28	16	10:55	36	10:35	30	145:01	908
rd400	72:33	162	115:45	390	262:36	924	—	—
pr439	250:06	378	552:56	1436	—	—	—	—
d493	88:09	60	114:19	120	73:12	58	—	—
att532	182:54	180	—	—	2732:21	3516	—	—
ali535	28:30	8	61:54	52	39:28	16	—	—
p654	11:52	34	18:34	64	44:17	154	—	—
gr666	148:46	70	168:23	134	169:44	110	—	—
rat783	25:28	12	61:00	100	35:03	30	—	—
Sum	927:29	1324	1309:56	3230	3613:37	6126	610:24	5254

The experiments (see Table IX) showed the good performance of the strategy (1). We assume that the reason is that there is a high "uncertainty" at a variable with value close to 0.5. The results of strategy (2) indicate that branching on an "expensive" variable is preferable, because this may have more influence on the objective function value of the sons of the current branch & cut node. The selection of the branching variable according to strategy (3) is certainly good if we have already found an optimum tour, but since this cannot be guaranteed, the risc to work in a "bad" branch of the branch & cut tree is quite high. With strategy (4) several problems could not be solved to optimality. Strategy (1) is the default strategy.

3.1.7 Upper bounding strategies

In view of the enormous choice of possible upper bounding strategies we had to limit our testing to a few reasonable experiments. Already for the Lin-Kernighan heuristic several parameters can be modified, e.g., the maximal number of iterations, the maximal number of exchanges in a sequence, the number of moves that are examined at the current terminal node of the exchange sequence and the level to which backtracking is used. We show the results of three different dynamic improvement strategies.

Table X
Upper bounding strategies

Problem	(1) T	(1) N	(2) T	(2) N	(3) T	(3) N
gr120	0:14	2	0:17	2	0:12	0
bier127	0:17	0	0:13	0	0:15	0
pr152	2:21	24	2:35	24	2:06	24
rat195	6:52	56	5:04	50	9:18	68
d198	4:11	42	3:29	28	3:28	36
gr229	15:19	68	18:27	68	16:30	70
gil262	5:30	12	7:19	22	5:13	16
pr299	77:59	200	60:11	216	65:03	180
lin318	6:28	16	4:47	12	6:26	20
rd400	72:33	162	52:49	140	62:01	198
pr439	250:06	378	282:21	398	338:05	436
d493	88:09	60	127:25	84	86:47	62
att532	182:54	180	462:47	430	235:41	254
ali535	28:30	8	35:52	16	45:23	20
p654	11:52	34	11:34	34	11:46	38
gr666	148:46	70	103:42	48	87:19	32
rat783	25:28	12	38:01	16	42:09	32
Sum	927:29	1324	1216:53	1588	1017:42	1486

We reinitialize the candidate set for the improvement heuristics every r LP solutions and start a series of improvement heuristics only every i solved LPs.

(1) We set $r = 10$ and $i = 5$. If the starting solution was more than 15% longer than the currently best known tour we aborted the improvement at once. Otherwise we called a version of the Lin-Kernighan heuristic, which used only 2-opt moves. Then another more sophisticated version of the Lin-Kernighan heuristic was applied. Afterwards the parameters of the last heuristic were modified for a more intensive search, where also node insertion moves might be used in an exchange.

(2) Like (1), but the improvement was aborted after the first application of the Lin-Kernighan heuristic.

(3) Like (1), but we did additional work every r solved LPs, if the tour after the three calls of the Lin-Kernighan heuristic was not longer than 0.5% but still worse than the currently best known tour. Then transitive edges were added to the candidate set, i.e., if the edges (j,k) and (k,l) belonged to the candidate set then also the edge (j,l) was added. Again a version of the Lin-Kernighan heuristic was applied and finally several three-opt exchanges were used to find a better tour. We set the parameters $r = 20$ and $i = 10$.

Strategy (2) worked quite well for some problems in comparison to strategy (1), but Table X shows that this quick strategy was not good enough to find good upper bounds for some hard instances. Strategy (3) was inferior to strategy (1) and further experiments showed that the degradation in performance by spending still more CPU time in the improvement part (e.g., by modifying the parameters r and i) was not compensated by a reduction of the number of branch & cut nodes. Strategy (1) is the default strategy.

3.1.8 Default settings of parameters

The outcome of the experiments might suggest that different default settings might be preferable, yet finding a good default setting might require many more iterations through the above described experiments.

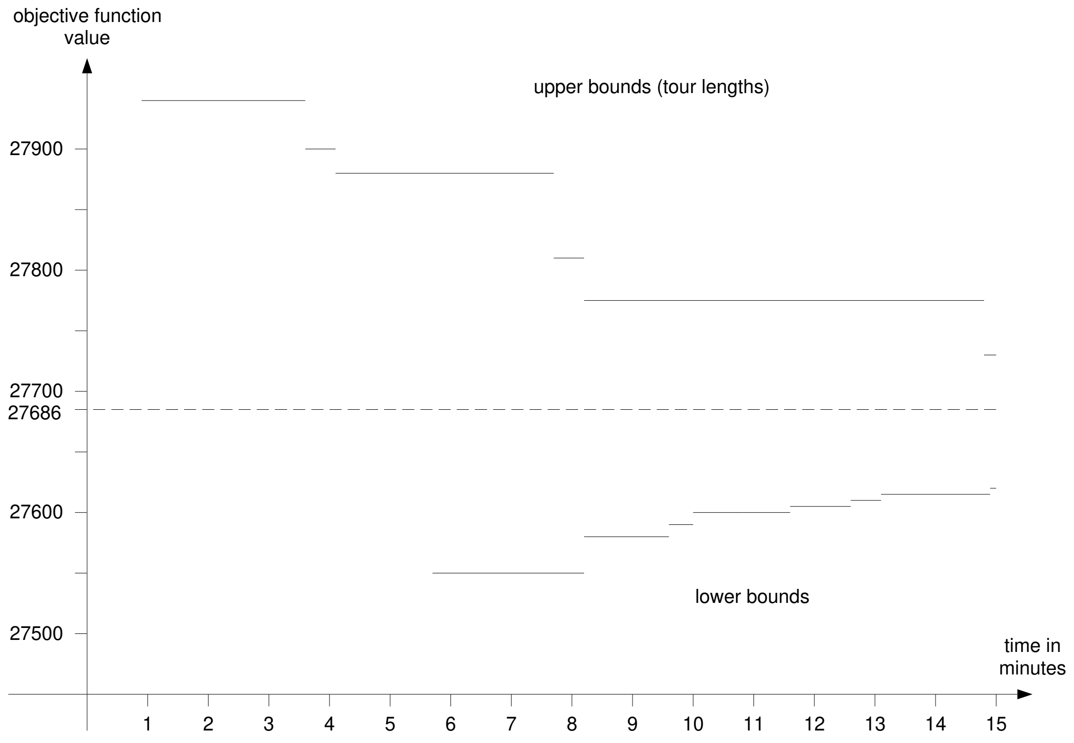

Figure 4. Gap versus time plot for `att532`

3.2 Guaranteed solutions

Let `glb` be a global lower bound and `gub` be a the length of a known tour as specified in section 2. We denote by `opt` the length of an optimum tour. In this section we distinguish between the guarantee and the quality of a given tour. The **guarantee** g is defined as

$$g = \frac{\texttt{gub} - \texttt{glb}}{\texttt{glb}} \cdot 100$$

and expresses that the tour with length `gub` is at most g% longer than an optimum tour. The **quality** q is defined as

$$q = \frac{\texttt{gub} - \texttt{opt}}{\texttt{opt}} \cdot 100$$

and expresses that the tour with length `gub` is exactly $q\%$ longer than an optimum tour. The length of an optimum tour of a problem (if it is known) can be found in TSPLIB.

A branch and cut algorithm as outlined in the previous section produces a sequence of increasing global lower bounds as well as a sequence of tours of decreasing lengths. Therefore, at any point during the computation we have a tour along with a certain guarantee.

We performed the following experiments using the default parameters and strategies of subsection 3.1. We stopped our branch & cut algorithm as soon as a guarantee which is smaller than g (`guarantee reached` in the flowchart of Figure 3) could be given. Table XI shows the results for $g = 10$ for all instances of TSPLIB between 100 and 4461 cities.

We can give a first global lower bound if after a pricing step no variable has to be added. Additional pricing steps are performed every 5 solved LPs per default. So it may last several minutes until a first guarantee can be computed. At this time the gap between the lower and upper bound is often already quite small. In Table XI the actual guarantee is less than 5% or even 2% for most problems, even though we required only a guarantee of 10%. For two instances we had to perform a second experiment for $g = 5\%$ (see Table XII). For those problems for which we could not give a guarantee of 1% we executed a third experiment with $g = 1\%$. The results are presented in Table XIII. For some problem we could not reach a guarantee less than 1% in less than four hours of CPU time. Since for some problems an optimum solution is not known, we could not give the quality for all instances.

For every problem it is easy to give a guarantee of 10%, and also the CPU time to give a guarantee of 1% is very small for most instances. But if we compare the time spent to give a 1%-guarantee and the time to solve a problem to optimality, we see that it is very time consuming to close this last gap.

A typical example of the decreasing upper and increasing lower bounds is shown in Figure 4 for the problem `att532`. We show the development of the upper and lower bounds during the first 15 minutes of the computation. 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 tour is only guaranteed after a pricing step in which all nonactive variables price out correctly.

In another experiment we computed upper and lower bounds within a given maximal CPU time. We stopped the computation after t minutes, if the problem had not been solved to optimality yet. In Table XIV we present quality and guarantee for $t = 7.5, 15, 30$ and 60. If we could solve a problem to optimality for a CPU time bound t, the following columns contain only blanks. If there is dash in a quality column, an optimum solution is not known.

For some problems no valid global lower bound was known, when we wanted to abort the computation. In this case we performed a pricing step. If variables had to be added we resolved the LP. This process was repeated until a first global lower bound was found. If we wanted to abort the computational process after 7.5 minutes, this iteration was necessary

Table XI
Guarantee 10% (part 1)

Problem	T	N	Guarantee	Quality
kroA100	0:05	0	0.45	0.11
kroB100	0:07	0	0.57	0.00
kroC100	0:03	0	2.03	1.29
kroD100	0:05	0	0.47	0.07
kroE100	0:03	0	1.38	0.28
rd100	0:04	0	0.00	0.00
eil101	0:03	0	0.36	0.16
lin105	0:03	0	0.00	0.00
pr107	0:04	0	0.00	0.00
gr120	0:07	0	0.26	0.13
pr124	0:37	4	0.46	0.08
bier127	0:11	0	0.19	0.00
pr136	0:03	0	3.58	2.51
gr137	0:16	0	0.63	0.34
pr144	0:24	2	0.12	0.00
kroA150	0:11	0	1.47	0.92
kroB150	0:12	0	1.95	1.01
pr152	0:14	0	0.71	0.00
u159	0:07	0	1.15	0.91
rat195	0:24	0	1.48	0.77
d198	1:09	0	0.47	0.25
kroA200	0:23	0	0.89	0.28
kroB200	0:16	0	0.90	0.59
gr202	0:22	0	0.15	0.05
ts225	0:08	0	8.81	2.15
pr226	0:30	0	0.01	0.00
gr229	0:53	0	1.39	0.69
gil262	0:33	0	0.59	0.13
pr264	0:35	2	0.02	0.00
pr299	0:41	0	1.92	0.72
lin318	1:51	0	0.56	0.49
rd400	0:31	0	1.46	0.84
fl417	1:38	0	0.80	0.42
gr431	4:35	0	0.57	0.23
pr439	4:46	0	0.74	0.20
pcb442	0:49	0	0.69	0.56
d493	1:50	0	0.24	0.02
att532	5:30	0	1.19	0.70
ali535	9:43	0	0.81	0.64
u574	4:12	0	0.20	0.08
rat575	1:50	0	1.00	0.62
p654	3:03	0	0.13	0.07
d657	2:44	0	1.08	0.65
gr666	8:46	0	0.42	0.17
u724	4:10	0	0.74	0.50
rat783	4:50	0	0.22	0.14
dsj1000	14:25	0	1.54	1.30
pr1002	11:11	0	1.33	1.08
u1060	11:20	0	0.82	0.54
vm1084	8:32	0	1.67	0.91
pcb1173	8:15	0	0.65	0.38
d1291	216:48	0	1.55	—
rl1304	34:01	0	0.90	0.27

Table XI
Guarantee 10% (part 2)

Problem	T	N	Guarantee	Quality
rl1323	56:33	0	1.32	0.60
nrw1379	11:12	0	0.87	0.74
fl1400	17:23	0	6.20	—
u1432	15:59	0	0.38	0.30
fl1577	58:58	0	1.54	—
d1655	85:34	0	0.30	0.09
vm1748	12:37	0	2.29	1.47
u1817	57:38	0	1.64	—
rl1889	65:50	0	1.56	—
d2103	50:38	0	2.87	—
u2152	68:30	0	1.51	—
u2319	20:17	0	0.36	—
pr2392	21:33	0	3.55	2.93
pcb3038	69:38	0	1.09	0.77
fl3795	417:58	0	2.97	—
fnl4461	184:16	0	0.62	—

Table XII
Guarantee 5%

Problem	T	N	Guarantee	Quality
ts225	5:05	14	4.78	1.14
fl1400	19:33	0	4.23	—

for 38 instances and lasted between 15 and 210 seconds for the instances with less than 3000 cities. However, for the three largest problems (pcb3038, fl3795, fnl4461) this iteration needed between 270 and 420 seconds. For $t = 60$ we had to do these repeated pricing operations only for seven problems.

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 surprisingly good solutions. However, the quality and guarantee could only be assessed because optimum solutions or tight lower bounds, respectively, were known.

When optimization problems arise in practice we have to have confidence in the guarantee of the solutions. Giving guarantees becomes possible by reasonably efficient calculations of lower bounds. The best such bounds have always been produced by some variant of the LP relaxation method. The branch and cut approach meets the goals of producing good solutions as well as reasonable guarantees at the same time.

4 Acknowledgements

The research underlying this report would not have been possible without the help of Robert E. Bixby, Martin Grötschel, Denis Naddef, Manfred Padberg and Giovanni Rinaldi, who supported us with their insights on the traveling salesman problem, polyhedral combinatorics and LP-techniques for several years.

Table XIII
Guarantee 1%

Problem	T	N	Guarantee	Quality
kroC100	0:07	0	0.92	0.50
kroE100	0:05	0	0.79	0.00
pr136	0:09	0	0.53	0.03
kroA150	0:28	0	0.94	0.68
kroB150	0:21	0	0.98	0.34
u159	0:09	0	0.24	0.00
rat195	0:45	0	0.99	0.52
ts225	—	–	—	—
pr226	0:30	0	0.01	0.00
gr229	2:04	2	0.43	0.08
pr299	5:53	2	0.81	0.36
rd400	1:03	0	0.72	0.11
att532	7:25	0	0.93	0.44
d657	3:41	0	0.68	0.25
dsj1000	25:54	0	0.75	0.56
pr1002	33:55	0	0.99	0.82
vm1084	46:22	0	0.98	0.45
d1291	—		—	—
rl1323	146:30	0	0.91	0.34
fl1400	—		—	—
fl1577	—		—	—
vm1748	230:50	0	0.96	0.60
u1817	—	–	—	—
rl1889	—	–	—	—
d2103	—	–	—	—
u2152	—	–	—	—
pr2392	—	–	—	—
pcb3038	121:36	0	0.93	0.63
fl3795	—	–	—	—

References

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.

J.L. Bentley (1990), Experiments on geometric traveling salesman heuristics, Computer Science Technical Report No. 151, AT&T Bell Laboratories.

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

S.C. Boyd and W.H. Cunningham (1991), Small traveling salesman polytopes, *Mathematics of Operations Research* 16, 259–271.

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

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.

N. Christofides (1976), Worst case analysis of a new heuristic for the traveling salesman problem, Report 388, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh.

N. Christofides (1979), The traveling salesman problem, in: N. Christofides, A. Mingozzi, P. Toth and C. Sandi, eds., *Combinatorial Optimization*, John Wiley & Sons, Chichester, pp. 131–149.

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

V. Chvátal (1983), *Linear programming*, W.H. Freeman and Company, New York.

Table XIV
Prespecified time (part 1)

Problem	g 7.5	q 7.5	g 15	q 15	g 30	q 30	g 60	q 60
kroA100	0.00	0.00						
kroB100	0.00	0.00						
kroC100	0.00	0.00						
kroD100	0.00	0.00						
kroE100	0.00	0.00						
rd100	0.00	0.00						
eil101	0.00	0.00						
lin105	0.00	0.00						
pr107	0.00	0.00						
gr120	0.00	0.00						
pr124	0.00	0.00						
bier127	0.00	0.00						
pr136	0.00	0.00						
gr137	0.00	0.00						
pr144	0.00	0.00						
kroA150	0.00	0.00						
kroB150	0.00	0.00						
pr152	0.00	0.00						
u159	0.00	0.00						
rat195	0.00	0.00						
d198	0.00	0.00						
kroA200	0.00	0.00						
kroB200	0.00	0.00						
gr202	0.00	0.00						
ts225	4.72	1.14	3.95	0.50	3.79	0.50	3.35	0.25
pr226	0.00	0.00						
gr229	0.15	0.04	0.08	0.03	0.00	0.00		
gil262	0.00	0.00						
pr264	0.00	0.00						
pr299	0.81	0.36	0.40	0.10	0.17	0.00	0.09	0.00
lin318	0.00	0.00						
rd400	0.25	0.04	0.18	0.03	0.09	0.00	0.04	0.00
fl417	0.14	0.00	0.12	0.00	0.11	0.00	0.09	0.00
gr431	0.48	0.18	0.39	0.17	0.27	0.10	0.13	0.03
pr439	0.57	0.20	0.36	0.07	0.24	0.07	0.13	0.00
pcb442	0.12	0.06	0.12	0.06	0.03	0.00	0.01	0.00
d493	0.14	0.02	0.11	0.02	0.06	0.01	0.02	0.00
att532	0.83	0.44	0.35	0.16	0.30	0.16	0.14	0.03
ali535	0.81	0.63	0.28	0.19	0.00	0.00		
u574	0.16	0.08	0.09	0.04	0.04	0.00	0.00	0.00
rat575	0.70	0.48	0.47	0.29	0.15	0.04	0.12	0.04
p654	0.00	0.00						
d657	0.56	0.25	0.36	0.05	0.29	0.05	0.24	0.05
gr666	0.42	0.17	0.32	0.17	0.30	0.16	0.14	0.08
u724	0.45	0.25	0.42	0.25	0.40	0.25	0.27	0.15
rat783	0.19	0.14	0.05	0.02	0.00	0.00		
dsj1000	1.93	1.62	1.54	1.30	0.71	0.53	0.64	0.46
pr1002	1.60	1.28	1.25	0.99	1.19	0.99	0.76	0.63
u1060	0.83	0.53	0.80	0.53	0.52	0.31	0.41	0.24
vm1084	1.69	0.91	1.18	0.50	1.12	0.50	0.91	0.45
pcb1173	0.65	0.38	0.56	0.35	0.28	0.12	0.23	0.09
d1291	2.11	—	1.96	—	1.81	—	1.76	—
rl1304	2.81	1.43	1.61	0.76	0.90	0.27	0.81	0.22

Table XIV
Prespecified time (part 2)

Problem	g 7.5	q 7.5	g 15	q 15	g 30	q 30	g 60	q 60
`rl1323`	2.72	1.51	2.47	1.51	1.87	1.03	1.31	0.60
`nrw1379`	0.91	0.73	0.86	0.73	0.83	0.73	0.52	0.43
`fl1400`	6.73	—	6.20	—	4.10	—	4.10	—
`u1432`	1.28	1.10	0.38	0.30	0.37	0.30	0.36	0.30
`fl1577`	7.63	—	3.38	—	2.19	—	1.54	—
`d1655`	3.37	2.65	1.03	0.61	0.62	0.27	0.42	0.19
`vm1748`	2.47	1.48	2.17	1.47	2.06	1.47	2.01	1.47
`u1817`	3.27	—	2.38	—	1.83	—	1.64	—
`rl1889`	4.60	—	3.88	—	1.67	—	1.57	—
`d2103`	3.81	—	3.35	—	2.89	—	2.86	—
`u2152`	3.72	—	1.66	—	1.54	—	1.51	—
`u2319`	0.44	—	0.36	—	0.30	—	0.24	—
`pr2392`	3.93	2.85	3.58	2.85	3.51	2.85	2.91	2.37
`pcb3038`	4.63	3.85	3.81	3.16	1.70	1.28	1.09	0.77
`fl3795`	25.27	—	11.81	—	7.59	—	6.38	—
`fnl4461`	12.66	—	9.84	—	1.74	—	1.47	—

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.

T.H. Cormen, C.E. Leiserson and R.L. Rivest (1990), *Introduction to algorithms*, MIT Press, Cambridge.

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

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

U. Derigs and A. Metz (1991), Solving (large-scale) matching problems combinatorically, *Mathematical Programming* 50, 113–121.

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.

R.E. Gomory (1958), Outline of an algorithm for integer solutions to linear programs, *Bulletin of the American Mathematical Society* 64, 275–278.

R.E. Gomory (1960), Solving linear programming problems in integers, *Proceedings of the Symposium on Applied Mathematics* 10, 211–215.

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.

M. Grötschel (1977), Polyedrische Charakterisierungen kombinatorischer Optimierungsprobleme, Hain, Meisenheim am Glan.

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

M. Grötschel and O. Holland (1991), Solution of large-scale symmetric traveling salesman problems, *Mathematical Programming* 51, 141–202.

M. Grötschel, M. Jünger and G. Reinelt (1984), A cutting plane algorithm for the linear ordering problem, *Operations Research* 32, 1195–1220.

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

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

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

M. Held and R.M. Karp (1971), The traveling salesman problem and minimum spanning trees: Part II, *Mathematical Programming* 1, 6–25.

D.S. Johnson (1990), Local Optimization and the traveling salesman problem, Proceedings of the 17th colloquium on automata, languages and programming, Springer Verlag, 446–461.

M. Jünger and P. Mutzel (1993), Solving the maximum weight planar subgraph problem by branch & cut, *Proceedings of the third IPCO conference*, 479–492.

M. Jünger, G. Reinelt and G. Rinaldi (1992), The traveling salesman problem, Report No. 92.113, Angewandte Mathematik und Informatik, Universität zu Köln, to apear in M. Ball, T. Magnanti, C.L. Monma and G. Nemhauser (eds.), *Handbook on Operations Research and Management Sciences: Networks*, North Holland.

D.E. Knuth (1973), *The art of computer programming, volume 3, sorting and searching*, Addison-Wesley, Reading, Massachusetts.

S. Lin and B.W. Kernighan (1973), An effective heuristic algorithm for the traveling salesman problem, *Operations Research* 21, 498–516.

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

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 the 14th International Symposium on Mathematical Programming, Amsterdam.

G.L. Nemhauser and L.A. Wolsey (1988), *Integer and combinatorial optimization*, John Wiley & Sons, Chichester.

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

M.W. Padberg and M.R. Rao (1982), Odd minimum cut sets and b-matchings, *Mathematics of Operations Research* 7, 67–80.

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.

M.W. Padberg and G. Rinaldi (1990), Facet identification for the symmetric traveling salesman polytope, *Mathematical Programming* 47, 219–257.

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.

G. Reinelt (1991a), TSPLIB – A traveling salesman problem library, *ORSA Journal on Computing* 3, 376–384.

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

G. Reinelt (1992), Fast heuristics for large geometric traveling salesman problems, *ORSA Journal on Computing* 4, 206–217.

T. Volgenant and R. Jonker (1982), A branch and bound algorithm for the symmetric traveling salesman problem based on the 1-tree relaxation, *European Journal of Operational Research* 9, 83–89.