Combining this with the inequality (see Exercise 3.66)

$$u(\delta(S)) + u(\delta(Q)) \geq u(\delta(S \cup Q)) + u(\delta(S \cap Q))$$

we can conclude that

$$u(\delta(S \cap Q)) \leq u(\delta(Q)).$$

∎

This allows us to reduce the problem to two problems on smaller graphs. To see this, let G^1 be the graph obtained by shrinking $V \setminus S$ to a single (new) node and let $T^1 = T \setminus S$ and $u^1_e = u_e$ for all $e \in E(G^1)$. Similarly, let G^2 be the graph obtained by shrinking S to a single (new) node and let $T^2 = T \cap S$ and $u^2_e = u_e$ for all $e \in E(G^2)$. Then the minimum T-cut in G can be found by solving the minimum T^1-cut problem in G^1 and the minimum T^2-cut problem in G^2.

We solve the two new problems with the same procedure, splitting them into further subproblems if necessary.

Since we can find S and build G^1 and G^2 in polynomial time, it follows by induction on $|T|$ that the whole procedure runs in polynomial time. (See Exercise 6.37.)

As one would expect, the minimum T-cut algorithm performs poorly in practice for larger test instances. A more efficient alternative (actually, the algorithm proposed by Padberg and Rao), works by computing a Gomory-Hu cut-tree, as described in Exercise 6.39.

Exercises

6.37. Show that the minimum T-cut algorithm runs in time $O(n^5)$. (Hint: Use induction and the fact that $|T| = |T^1| + |T^2|$.)

6.38. Suppose that we are given a vector $\bar{x} \in \mathbf{R}^V$ satisfying the initial valid inequalities (6.23) for the stable set polytope of $G = (V, E)$. Show how to reduce the separation problem for the odd circuit inequalities to the problem of finding a minimum weight odd circuit in a graph having nonnegative edge weights. Show that the latter problem can be solved using shortest path methods. (See Exercise 2.38).

6.39. Let $G = (V, E)$ be a graph, $T \subseteq V$ with $|T|$ even, and $u \in \mathbf{R}^E$ a nonnegative capacity function. Consider a Gomory-Hu cut-tree H with T as the set of terminals. Show that there exists an edge e of H such that the bipartition of V defined by the two components of $H \setminus e$ gives a minimum T-cut.

CHAPTER 7

The Traveling Salesman Problem

7.1 INTRODUCTION

In the general form of the traveling salesman problem, we are given a finite set of points V and a cost c_{uv} of travel between each pair $u, v \in V$. A *tour* is a circuit that passes exactly once through each point in V. The *traveling salesman problem* (TSP) is to find a tour of minimal cost.

The TSP can be modeled as a graph problem by considering a complete graph $G = (V, E)$, and assigning each edge $uv \in E$ the cost c_{uv}. A tour is then a circuit in G that meets every node. In this context, tours are sometimes called *Hamiltonian circuits.*

The TSP is one of the best known problems of combinatorial optimization. A nice collection of papers tracing the history and research on the problem can be found in Lawler, Lenstra, Rinnooy Kan, and Shmoys [1985].

Unlike the cases of matching or network flows, no polynomial-time algorithm is known for solving the TSP in general. Indeed, it belongs to the class of *$\mathcal{NP}$-hard* problems, which we describe in Chapter 9. Consequently, many people believe that no such efficient solution method exists, for such an algorithm would imply that we could solve virtually every problem in combinatorial optimization in polynomial time.

Nevertheless, TSPs do arise in practice, and relatively large ones can now be solved efficiently to optimality. In this chapter we discuss how. We will illustrate the methods on the 1173-node Euclidean problem depicted in Figure 7.1. Although this problem is smaller than the largest solved so far (as of August 1994, the record is 7397 nodes (Applegate, Bixby, Chvátal, Cook [1995])), it is still of a respectable size. The node coordinates for this instance are con-

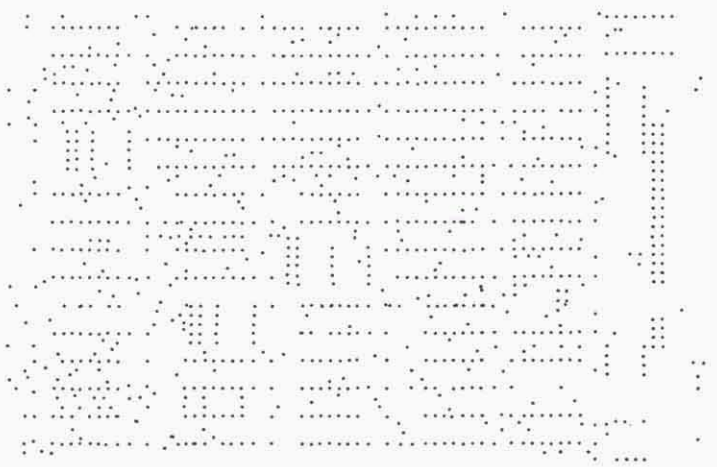

Figure 7.1. Sample TSP

tained in the "TSPLIB" library of test problems described in Reinelt [1991]. We encourage readers to try out some of their own methods.

Whereas in prior chapters, we were often able to describe polynomial-time algorithms that also performed well in practice, in this chapter we discuss algorithms which do work well empirically, but for which only very weak guarantees can be provided.

7.2 HEURISTICS FOR THE TSP

Heuristics are methods which cannot be guaranteed to produce optimal solutions, but which, we hope, produce fairly good solutions at least some of the time. For the TSP, there are two different types of heuristics. The first attempts to construct a "good" tour from scratch. The second tries to improve an existing tour, by means of "local" improvements. In practice it seems very difficult to get a really good tour construction method. It is the second type of method, in particular, an algorithm developed by Lin and Kernighan [1973], which usually results in the best solutions and forms the basis of the most effective computer codes.

Nearest Neighbor Algorithm

In Chapter 1 we described the Nearest Neighbor Algorithm for the TSP: Start at any node; visit the nearest node not yet visited, then return to the start node when all other nodes are visited. Applying it to our test problem, we obtain the tour of cost 67,822 that is exhibited in Figure 1.1. Note that the tour includes some very expensive edges. In practice, this almost always seems to happen when we use the Nearest Neighbor Algorithm.

Johnson, Bentley, McGeoch, and Rothberg [1997] report that on problems in TSPLIB, the average costs of the tours found by the Nearest Neighbor Algorithm are about 1.26 times the costs of the corresponding optimal tours. Thus, for some applications, Nearest Neighbor may be an effective method: It is easy to implement, runs quickly, and usually produces tours of reasonable quality. It should be noted, however, that the "1.26 times optimal" estimate is an empirical observation, not a performance guarantee. Indeed, it is easy to construct problems on only four nodes for which the Nearest Neighbor Algorithm can produce a tour of cost arbitrarily many times that of the optimal tour. (See Exercise 7.2.)

To obtain a guaranteed bound, we need to assume that the edge costs are nonnegative and satisfy the triangle inequality:

$$c_{uv} + c_{vw} \geq c_{uw}, \text{ for all } u, v, w \in V.$$

In this case, Rosenkrantz, Stearns, and Lewis [1977] show that a Nearest Neighbor tour is never more than $\frac{1}{2}\lceil log_2 n \rceil + \frac{1}{2}$ times the optimum, where n is the cardinality of V.

This bound may seem very weak (particularly when compared to the 1.26 observed bound on the TSPLIB problems), but Rosenkrantz, Stearns, and Lewis [1977] proved that we cannot do much better. They showed this by describing a family of problems with nonnegative costs satisfying the triangle inequality and with arbitrarily many nodes, such that the Nearest Neighbor Algorithm can produce a tour of cost $\frac{1}{3}\lceil log_2(n+1) + \frac{4}{9} \rceil$ times the optimum. This result shows that if we wish to give a worst case bound on the performance of this heuristic, the bound we get is so bad that it is not of much practical interest.

The proofs of the two results are not hard, but they are technical and we refer the reader to the reference cited above.

Insertion Methods

Insertion methods provide a different set of tour construction heuristics. They start with a tour joining two of the nodes, then add the remaining nodes one by one, in such a way that the tour cost is increased by a minimum amount. There are several variations, depending on which two nodes are chosen to start, and more importantly, which node is chosen to be inserted at each stage.

In practice, usually the best insertion method is *Farthest Insertion*. In this case, we start with an initial tour passing through two nodes that are the ends of some high-cost edge. For each uninserted node v, we compute the minimum cost between v and any node in the tour constructed thus far. Then we choose as the next node to be inserted the one for which this cost is *maximum*.

At first, this may seem counterintuitive. However, in practice, it often works well. This is probably because a rough shape of the final tour to be

produced is obtained quite early, and in later stages, only relatively slight modifications are made.

Nearest Insertion is a heuristic which, at each stage, chooses as the next node to insert the one for which the cost to any node in the tour is minimum.

Another variant is *Cheapest Insertion.* In this case, the next node for insertion is the one that increases the tour cost the least.

Usually the solutions produced by Nearest Insertion and Cheapest Insertion are inferior to those produced by Farthest Insertion. In Figures 7.2 and 7.3 we show the results of applying Nearest Insertion and Farthest Insertion to our test problem. On the TSPLIB problems, Johnson, Bentley, McGeoch, and Rothberg [1997] report that, on average, Farthest Insertion found tours of length about 1.16 times that of the optimal tours.

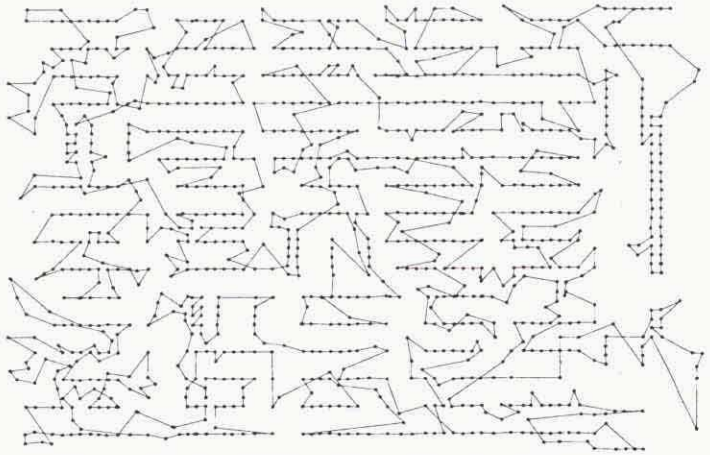

Figure 7.2. Sample TSP and Nearest Insertion solution: 72337

An extensive worst-case analysis of various insertion heuristics is provided in Rosenkrantz, Stearns, and Lewis [1977]. They prove that any insertion heuristic produces a solution whose value is at most $\lceil log_2 n \rceil + 1$ times the optimum, for a problem with n nodes (for which the edge costs are nonnegative and satisfy the triangle inequality). They show, further, that Cheapest Insertion and Nearest Insertion always produce solutions whose costs are at most twice the optimum. They also give an example which shows that this bound is essentially tight. See the above reference for details.

Interestingly, no examples are known which force any insertion method to construct a tour of cost more than four times that of the optimum. Also, in spite of the fact that Farthest Insertion usually produces the best solutions of any insertion method, no better worst case bound has been established for it than for insertion methods in general.

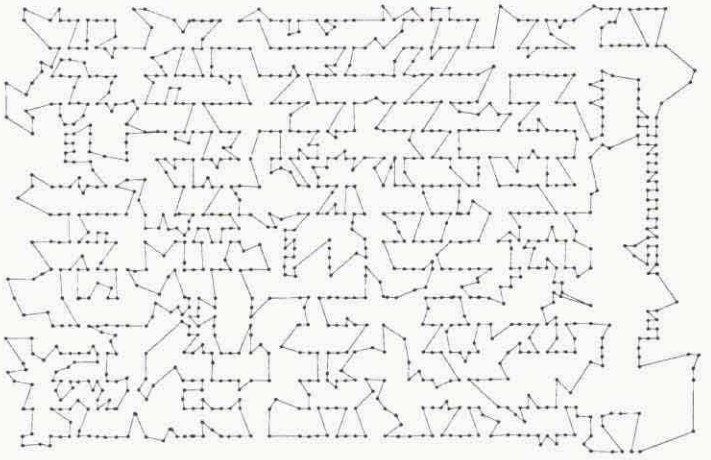

Figure 7.3. Sample TSP and Farthest Insertion solution: 65980

Christofides' Heuristic

We describe one more tour construction method, due to Christofides [1976]. It has the best worst case bound of any known method: It always produces a solution of cost at most $\frac{3}{2}$ times the optimum (assuming that the graph is complete and the costs are nonnegative and satisfy the triangle inequality).

The algorithm begins by finding a minimum-cost spanning tree, T, of G using, for example, Kruskal's Algorithm. The edges in T will be used in a search for a good tour.

Let W be the set of nodes which have odd degree in T, and find a perfect matching M of $G[W]$, the subgraph of G induced by W, which is of minimum cost with respect to c.

Now let J consist of $E(T) \cup M$, where, if some edge is in both T and M, we take two copies of the edge. Then J is the edge-set of a connected graph with node-set V for which each node has even degree. If all nodes have degree 2, then J is the edge-set of a tour and we terminate with it. If not, let v be any node of degree at least 4 in (V, J). Then there are edges uv and vw such that if we delete these edges from J and add the edge uw to J, then the subgraph remains connected. Moreover, the new subgraph has even degree at each node. (This is because the subgraph induced by J has an Euler tour; we choose uv and vw to be consecutive edges of the tour.) Make this "shortcut" and repeat this process until all nodes are incident with two edges of J.

Theorem 7.1 *Suppose we have a TSP with nonnegative costs satisfying the triangle inequality. Then any tour constructed by Christofides' Heuristic has cost at most $\frac{3}{2}$ times the cost of an optimal tour.*

Proof: Let H^* be an optimal tour. Removing any edge from H^* yields a spanning tree, so the cost $c(T)$ of a minimum-cost spanning tree T is at most $c(H^*)$. We can define a circuit C on the set W of odd nodes of T by joining these nodes in the order they appear in H^*. Note that $|W|$ is even and the edge-set of C partitions into two perfect matchings of $G[W]$. Since c satisfies the triangle inequality, each edge of these matchings has cost no greater than the corresponding subpath of H^*. Therefore one of these matchings has cost at most $c(H^*)/2$. This implies that the cost of the minimum-cost perfect matching M of $G[W]$ is at most $c(H^*)/2$. Thus $c(J) \leq \frac{3}{2} \cdot c(H^*)$. Since shortcutting can only improve $c(J)$, the final tour produced also has cost at most $\frac{3}{2} \cdot c(H^*)$, as required. ∎

In the Johnson, Bentley, McGeoch, and Rothberg [1997] tests on the TSPLIB problems, Christofides' Heuristic produced tours that were about 1.14 times the optimum. They also made the interesting discovery that if at each shortcut step the best shortcut for the given node is chosen, then the performance of the algorithm improves to 1.09 times the optimum.

Tour Improvement Methods: 2-opt and 3-opt

There are several standard methods for attempting to improve an existing tour T. The simplest is called *2-opt*. It proceeds by considering each nonadjacent pair of edges of T in turn. If these edges are deleted, then T breaks up into two paths T_1 and T_2. There is a unique way that these two paths can be recombined to form a new tour T'. If $c(T') < c(T)$, then we replace T with T' and repeat. This process is called a *2-interchange*. See Figure 7.4. If $c(T') \geq c(T)$ for every choice of pairs of nonadjacent edges, then T is *2-optimal* and we terminate.

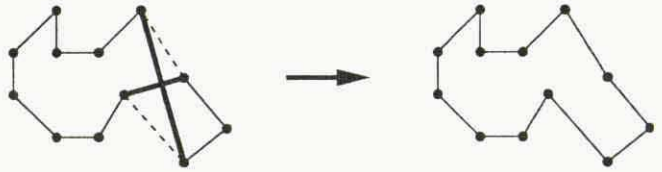

Figure 7.4. 2-interchange

For a general cost function, in order to check whether a tour is 2-optimal we check $O(|V|^2)$ pairs of edges. For each pair, the work required to see if the switch decreases the tour cost can be performed in constant time. Thus the amount of time required to check a tour for 2-optimality is $O(|V|^2)$. But this does not mean that we can transform a tour into a 2-optimal tour in polynomial time. Indeed, Papadimitriou and Steiglitz [1977] show that if

we make unfortunate choices, we may in some cases perform an exponential number of interchanges, before a 2-optimal tour is found.

The 2-opt algorithm can be generalized naturally to a k-opt algorithm, wherein we consider all subsets of the edge-set of a tour of size k, or size at most k, remove each subset in turn, then see if the resulting paths can be recombined to form a tour of lesser cost. The problem is that the number of subsets grows exponentially with k, and we soon reach a point of diminishing return. For this reason, k-opt for $k > 3$ is seldom used.

Johnson, Bentley, McGeoch, and Rothberg [1997] report that on the TSPLIB problems, 2-opt produces tours about 1.06 times the optimum and 3-opt about 1.04 times the optimum.

Tour Improvement Methods: Lin-Kernighan

Lin and Kernighan [1973] developed a heuristic which works extremely well in practice. It is basically a k-opt method with two novel features. First, the value of k is allowed to vary. Second, when an improvement is found, it is not necessarily used immediately. Rather the search continues in hopes of finding an even greater improvement. In order to describe it, we require several definitions.

A *δ-path* in a graph G on n nodes is a path containing n edges and $n+1$ nodes all of which are distinct except for the last one, which will appear somewhere earlier in the path. See Figure 7.5. (The name comes from the shape of the path.)

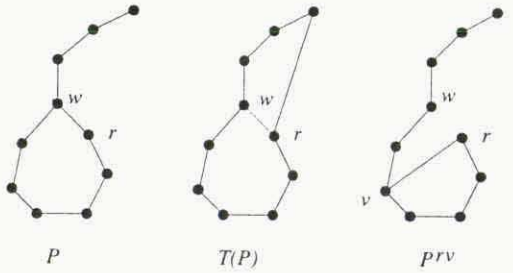

Figure 7.5. δ-path

Note that a tour is a δ-path for which the last node is the same as the first node. If P is a δ-path which is not a tour, then we can obtain a tour $T(P)$ as follows. Let w be the last node of the path (which also appears earlier in the path). Let wr be the first edge of the subpath of P between the two occurrences of w. By removing the edge wr and adding the edge joining r to the first node of the path, we obtain the edge set of a tour. See Figure 7.5.

Suppose that P is a δ-path which is not a tour. Again, let w be the last node and let wr be the first edge of the subpath of P between the two occurrences

of w. If we remove wr we obtain the edge-set of a path ending at r. If we then add one more edge rv and node v, we obtain a new δ-path P^{rv} ending at v. We call this operation an *rv-switch.* Note that $c(P^{rv}) = c(P) + c_{rv} - c_{wr}$. Again, see Figure 7.5.

The Lin-Kernighan heuristic starts with a tour and then constructs a sequence of non-tour δ-paths, each obtained from the preceding one by an rv-switch. For each δ-path P so produced, it computes the cost of $T(P)$. If this is better than the best tour known, then it is "remembered." When the scan is complete, it replaces the starting tour with the best tour found in the scan. A full description of the *Core Lin-Kernighan Heuristic* is given below.

Step 1 [Outer loop: node/edge pairs]. For each node v of G, for each of the two edges uv of T incident with v in turn, perform Steps 2 through 5 in an attempt to obtain an improvement. This process is called an *edge scan.*

Step 2 [Initialize edge scan]. Initially, the best tour found is T. Let $u_0 = u$. Remove edge u_0v and add an edge u_0w_0, for some $w_0 \neq v$, provided that such a w_0 can be found for which $c_{w_0u_0} \leq c_{u_0v}$. If no such w_0 can be found, then this scan is complete and we go on to the next node/edge pair.

We now have a δ-path P^0 (with last edge u_0w_0) and $c(P^0) \leq c(T)$. Set $i = 0$ and proceed to Step 3.

Step 3 [Test tour]. Construct the tour $T(P^i)$. If $c(T(P^i))$ is less than the cost of the best tour found so far, then store this as the new best tour found so far. In either case, proceed to the next step.

Step 4 [Build next δ-path]. Let u_{i+1} be the neighbor of w_i in P^i which belongs to the subpath joining w_i to u_i. If the edge w_iu_{i+1} was an edge added to a δ-path in this iteration, then go to Step 5 and stop this scan. Otherwise, try to find a node w_{i+1} such that $u_{i+1}w_{i+1}$ is not in T and when we perform the $u_{i+1}w_{i+1}$-switch, the new δ-path P^{i+1}, with last edge $u_{i+1}w_{i+1}$, we obtain has cost no greater than that of T. Again, if no such w_{i+1} can be found, we go to Step 5 and stop this scan. But if we are successful, then we set $i = i + 1$ and go back to Step 3. (See Figure 7.6.)

Step 5 [End of node/edge scan]. If we have found a tour whose cost is less than that of T, replace T with the minimum cost such tour found. If there remain untested node/edge combinations, then return to Step 2 and try the next.

Now let us make a few comments. First, note that in the process of a single node/edge scan, for any given edge, we can either add it to a δ-path or remove it from a δ-path but not both. Thus it makes sense to speak of *added* and *removed* edges. Each successive δ-path generated will have cost no greater than that of T, the initial tour. This is equivalent to saying that the sum of the costs of removed edges minus the sum of the cost of added edges is kept nonnegative. This difference is sometimes called the *gain sum.*

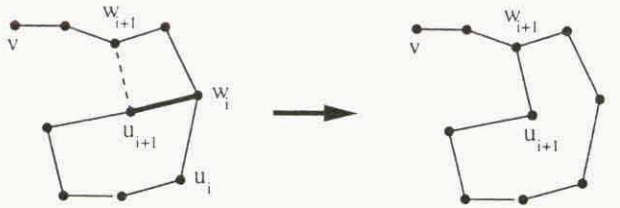

Figure 7.6. Construction of next δ-path

Notice that when a better tour $T(P^i)$ is found, we do not immediately abandon the process. Rather we continue looking for an even better completion.

In Steps 2 and 4 we chose a node w_{i+1}, subject to certain conditions. In general there will be many possible choices for these nodes. It may be too time consuming to try all possibilities at each stage, so Lin and Kernighan suggest the following compromise, intended to limit the amount of back-tracking. For each candidate w, compute $l(w) = c_{wu_{i+2}} - c_{u_{i+1}w}$, where u_{i+2} is the node which would be selected the next time through this step. Note that u_{i+2} is completely determined. When choosing each of w_0 and w_1, we consider in turn each of the five candidates w for which $l(w)$ is maximum. For all subsequent iterations, we consider only the best candidate. Thus in the process of scanning associated with a single node/edge pair, we will in fact consider as many as 25 choices for the first two edges to be switched in. For each we completely follow its chain of switches. If a better tour is found, then T is replaced and we start over. If not, we go on to the next.

They recommend one additional modification to the above core. The first time Step 4 is executed, when we choose u_1, we consider a second alternative. This is the neighbor of w_0 in the path back to v. Now removing the edge u_1w_0 yields a circuit and a path joining v and u_1. By joining u_1 to a node w_1 in the circuit, we obtain once again a δ-path starting from v. (We may orient the δ-path in either direction around the circuit.) We choose the best such w_1, according to the above criterion. See Figure 7.7(a). Going one step further in this situation, they also allow w_1 to be a node in the path joining v and u_1 (rather than in the circuit). In this case, we let u_2 be the first node on the subpath from w_1 to u_1, delete the edge w_1u_2, and form a δ-path by letting w_2 be a node in the circuit. See Figure 7.7(b).

This completes the description of the core algorithm. There is a wide range of possible modifications to the core that can be considered. Some interesting variants are described in Johnson and McGeoch [1997], Mak and Morton [1993], and Reinelt [1994].

To produce a very good quality tour, we need to embed the core algorithm into a larger search procedure. Lin and Kernighan propose running the core repeatedly, starting from many different tours. They also propose several

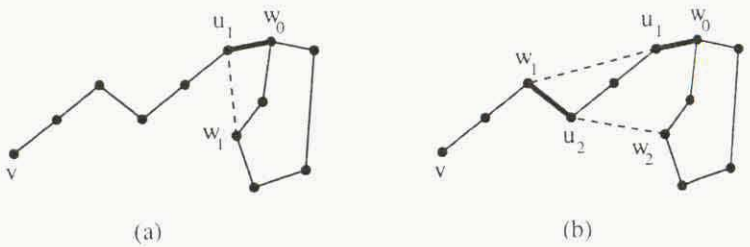

Figure 7.7. Extra first step

different methods for reducing the total amount of work required by these many runs of the core routine.

An alternative has been proposed by Martin, Otto, and Felten [1992] which seems to work very well in practice. Each time we complete a run of the core routine and hence have a "locally optimum" tour, T, we apply to it a "kick" that will perturb the tour so that it is likely to no longer be locally optimal. We then rerun the core routine from this new tour. If the core routine produces a new tour T' that is cheaper than T, then we replace T by T', and repeat the kicking process with this new tour. Otherwise, we go back and repeat the process with our best tour T.

One kick that they propose is a 4-interchange that the core routine is incapable of performing. It consists of randomly choosing four nonadjacent edges $u_0v_0, u_1v_1, u_2v_2, u_3v_3$ of the tour where we assume that the nodes appear in the above order on the tour. We remove these edges and add the edges $u_0v_2, u_1v_3, u_2v_0, u_3v_1$. See Figure 7.8. This becomes our new starting tour. Its cost will probably be much worse than that of the old local optimum, but it does provide a new starting point to rerun the core routine.

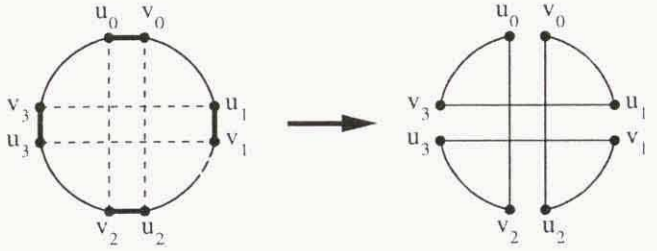

Figure 7.8. 4-interchange to restart core

This procedure is called *Chained Lin-Kernighan*. Martin and Otto [1996] describe it in a general context for search procedures for combinatorial optimization. One point that they make is that it may be helpful to replace T by T' even when T' is slightly more expensive than T. This added flexibility may allow the procedure to break away from a locally optimal tour that does not seem to permit good kicks. The rule they suggest is to replace T by T' with a certain probability that depends on the difference in the costs of the two tours and on the number of iterations of the procedure that have already been carried out.

Notice that Chained Lin-Kernighan is not a finite algorithm, since we have not provided any stopping rules. We would normally let it run until we see a long period with no improvement. Sometimes, however, we have available a good lower bound on the optimum solution cost, which will permit us to stop sooner. Obtaining such bounds is the subject of the next two sections.

The result of applying this method to our test problem is shown in Figure 7.9. On the TSPLIB problems, Johnson, Bentley, McGeoch, and Rothberg [1997] report that Lin-Kernighan produces tours about 1.02 times the optimum, and Chained Lin-Kernighan under 1.01 times the optimum.

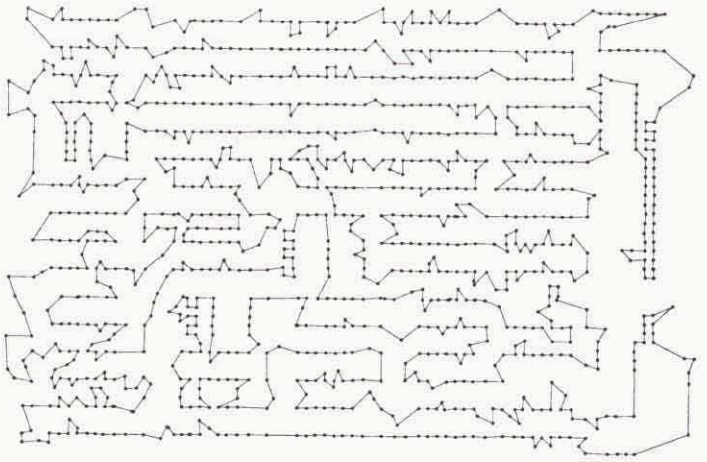

Figure 7.9. Sample TSP and Chained Lin Kernighan solution: 56892

Running Times

The methods described above can all be implemented to run efficiently, even for quite large problems. We refer the reader to the extensive treatment of this subject by Johnson, Bentley, McGeoch, and Rothberg [1997]. They report, for example, that on a randomly generated 10,000-node Euclidean problem, running times on a fast workstation (that is, fast in 1994), are 0.3 seconds for Nearest Neighbor, 7.0 seconds for Farthest Insertion, 41.9 seconds for Christofides' Heuristic, 3.8 seconds for 2-opt, 4.8 seconds for 3-opt, and 9.7 seconds for Lin-Kernighan.

Exercises

7.1. Let $G = (V, E)$ be a graph and $c \in \mathbf{R}^E$. Show that the problem of finding a minimum-cost Hamiltonian circuit in G can be formulated as a TSP.

7.2. Show that if we have a TSP on n nodes for which the triangle inequality does not hold, then the ratio of the cost of the solution produced by the Nearest Neighbor Algorithm to the cost of an optimal tour can be arbitrarily large.

7.3. A variant on the TSP permits a node to be visited more than once if it results in a better solution. Show that if the cost function satisfies the triangle inequality, then there is always an optimum solution which visits each node exactly once. Show that if the triangle inequality does not hold, this problem can be solved by solving a TSP for which the triangle inequality does hold.

7.4. Show that if we have a Euclidean TSP, then a tour is 2-optimal only if it never crosses itself, but the converse is false.

7.5. Use the following example to show that the factor 3/2 in Theorem 7.1 cannot be decreased. Let $V = \{v_1, v_2, \ldots, v_n\}$ and define $c_{v_i v_j}$ to be $\lfloor (|i - j| + 1)/2 \rfloor$ for $i \neq j$.

7.6. (Van Leeuwen and Schoone) Let V be any set of n nodes in the Euclidean plane and let T be any tour on these points. Suppose we attempt to obtain a noncrossing tour by choosing any pair of edges which cross and then performing a 2-interchange to uncross them. Show that the total number of crossings can be increased by such an operation. Show that after at most $|V|^3$ uncrossings, we necessarily obtain a noncrossing tour. (Hint: For each edge, viewed as a line segment, count the number of [infinite] lines that can be drawn through two cities so as to intersect that segment. Show that the removal of a crossing always reduces the total count, for all edges, by at least one.)

7.7. Show that if we consider all choices for w_{i+1} in Step 4 of the Core Lin-Kernighan Heuristic, then the resulting solution will always be 3-optimal but need not be 4-optimal. Show that the version described here (core only) need not be 3-optimal.

7.3 LOWER BOUNDS

We have emphasized the use of min-max relations in procedures for computing optimal solutions to combinatorial problems. The lower bound provided by the "max" side of the relation gives a proof of the optimality of the solution given in the "min" side. Unfortunately, for the TSP and many other problems known to be just as difficult as the TSP (see Chapter 9), no such min-max relation is known. Nonetheless it is important in some practical situations to give lower bounds as a measure of the quality of a proposed solution.

Held and Karp

A classic approach to lower bounds for the TSP involves the computation of minimum-cost spanning trees. The general technique is standard: To obtain a lower bound on a difficult problem we relax its constraints until we arrive at a problem that we know how to solve efficiently. In this case, the idea is that if we remove from a given tour the two edges incident with a particular node, then we are left with a path running through the remaining nodes. Although we do not in general know how to compute such a "spanning path" of minimum cost (it is just as hard as the TSP), we do know how to compute a minimum-cost spanning tree, and this will give us a lower bound on the cost of the path.

More precisely, suppose we have a graph $G = (V, E)$ with edge costs $(c_e : e \in E)$ and a tour $T \subseteq E$. Let $v_1 \in V$, let e and f be the two edges in T that are incident with v_1, and let P be the edge-set of the path obtained by removing e and f from T. The cost of T can be written as $c_e + c_f + c(P)$. So if we have numbers A and B such that $A \leq c_e + c_f$ and $B \leq c(P)$, then $A + B$ will be a lower bound on the cost of the tour T. Our goal is to define A and B so that they are valid for all choices of T. In this way, we will obtain a lower bound for all tours.

So, what can we say about the edges e and f? Since all we know is that they both are incident with v_1, we can do no better than to set A equal to the sum of the costs of the two cheapest edges in E incident with that node.

The interesting part is the bound on $c(P)$. Notice that P is a spanning tree (albeit of a special form) for the graph $G \setminus v_1$ we get by deleting v_1 and the incident edges from G. Thus, if we let B be the minimum cost of a spanning tree in $G \setminus v_1$ (which we can compute with the methods described in Chapter 2), then we know that P must have cost at least B. So we have our bound, $A + B$. This is commonly called the *1-tree bound* and a set of edges consisting of two edges incident with node v_1 plus a spanning tree of $G \setminus v_1$ is called a *1-tree*. The name comes from the usual practice of denoting the node that we delete as node v_1 or node "1." We can summarize this discussion as follows.

1-tree Bound

Let $G = (V, E)$ with edge costs $(c_e : e \in E)$ and let $v_1 \in V$. Now let $A = \min\{c_e + c_f : e, f \in \delta(v_1), e \neq f\}$ and let B be the cost of a minimum spanning tree in $G \setminus v_1$. Then $A + B$ is a lower bound for the TSP on G.

A minimum-cost 1-tree for the 1173-node problem is illustrated in Figure 7.10. (The node v_1 is in the lower right-hand corner of the figure, and is drawn larger than the other nodes.) Its value is approximately 90.5% of the cost of the best tour reported above. Although this is quite respectable, it

leaves a rather large gap between the upper and lower bounds. To improve

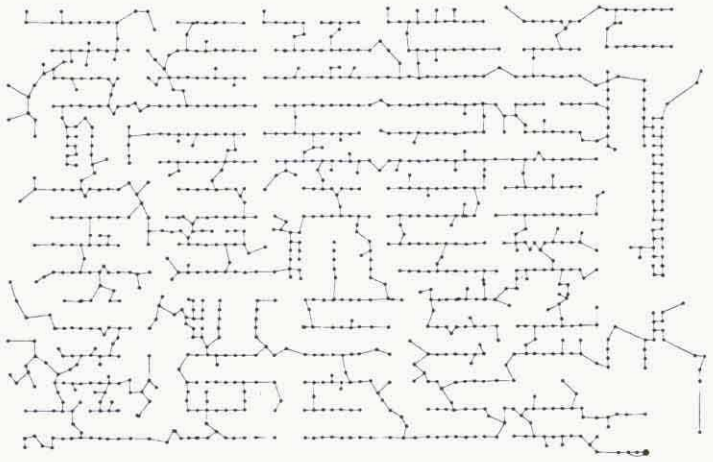

Figure 7.10. Sample TSP and 1-tree bound: 51488

this we might try several different nodes as "node v_1," but for larger problems, like our test problem, this will not result in a significant gain. The key to obtaining a real improvement can be found by examining the structure of the 1-tree in Figure 7.10. What catches your eye is that the optimal 1-tree does not at all resemble a tour: Many nodes do not have degree 2. We can use this to our advantage.

To see clearly what is happening, consider the small example given in Figure 7.11. With v_1 chosen as indicated, the 1-tree bound is 0. As you can

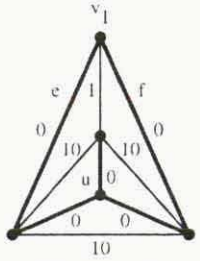

Figure 7.11. An optimal 1-tree

see, however, the cheapest tour has cost 10. What went wrong is that the spanning tree in G' can use all three of the 0 cost edges incident with node u, whereas a tour can only make use of two of them. There is a way around this problem, although at first it may seem like a sleight of hand.

What would happen if we added 10 to the cost of each of the edges incident with node u? Every tour uses exactly two of these edges, so the cost of every tour increases by precisely 20. So, as far as the TSP goes, we have not really changed anything: The old tour is still optimal, its cost now being 30. But what has happened to the 1-tree bound? A simple computation shows that it also has value 30. So we have a simple proof that no tour in the altered graph can have cost less than 30. This means that no tour in the original graph can have cost less than 10! By this simple transformation we have therefore increased the lower bound from 0 to 10. The point is that although the transformation does not alter the TSP, it does fundamentally alter the minimum spanning tree computation.

In the above example, we say that we "assigned u the node number -10." That is, we refer to the process of subtracting k from the cost of each edge incident with a given node v as assigning v the *node number* k. Notice that we could assign several node numbers at once. The change in the cost of any tour will just be twice the sum of the assigned numbers. With this fact, we can formally state a lower bounding technique, introduced by Held and Karp [1970].

Held-Karp Bound

Let $G = (V, E)$ be a graph with edge costs $(c_e : e \in E)$, let $v_1 \in V$, and for each node $v \in V$ let y_v be a real number. Now for each edge $e = uv \in E$ let $\bar{c}_e = c_e - y_u - y_v$ and let C be the 1-tree bound for G with respect to the edge costs $(\bar{c}_e : e \in E)$. Then $2\sum(y_v : v \in V)+C$ is a lower bound for the TSP on G (with respect to the original edge costs $(c_e : e \in E)$).

With a good set of node numbers, the difference between the Held-Karp lower bound and the 1-tree lower bound can be dramatic, as we indicate in Figure 7.12 for our 1173-node test problem. The 56349 bound we obtained in this way implies that the 56892-cost tour we found in the previous section is no more than 1% above the cost of an optimal tour.

For the important computational issue of how to find a good set of numbers, Held and Karp [1971] proposed a simple iterative scheme. The main step is the following. Suppose we have computed an optimum 1-tree T with respect to the altered edge costs $(c_{uv} - y_u - y_v : v \in V)$ for some set of node numbers $(y_v : v \in V)$. For each node $v \in V$, let $d_T(v)$ denote the number of edges of T that are incident with v. Based on our above discussion, if $d_T(v)$ is greater than 2 then we should decrease y_v and if $d_T(v)$ is equal to 1 (it cannot be less than 1) we should increase y_v. This is just what Held and Karp tell us to do: For each node v, replace y_v by

$$y_v + t(2 - d_T(v))$$

for some positive real number t (the *step size*).

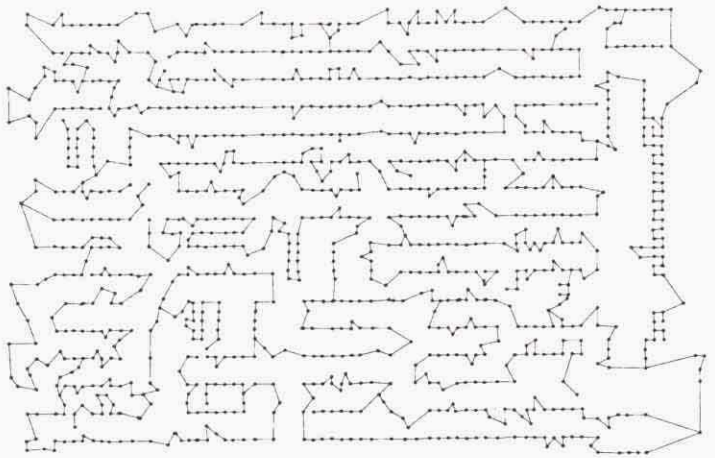

Figure 7.12. Sample TSP and Held-Karp bound: 56349

By iterating this step, we obtain a sequence of Held-Karp lower bounds. Although it is not true that the bound improves at each iteration, under certain natural conditions on the choice of the step sizes it can be shown that the bound will converge to the optimum Held-Karp bound (that is, the maximum Held-Karp bound over all choices of node numbers). (See Held, Wolfe, and Crowder [1974].) Unfortunately, no conditions are known that guarantee the convergence will occur in polynomial time. But all is not lost. As reported in Grötschel and Holland [1988], Holland [1987], Smith and Thompson [1977], and elsewhere, several ways of selecting the step sizes have shown good performance in practice. We describe one such method.

Motivated by geometrical considerations, at the kth iteration Held and Karp [1971] suggest the step size

$$t^{(k)} = \alpha^{(k)}(U - H)/\sum((2 - d_T(v))^2 : v \in V)$$

where U is some target value (an upper bound on the cost of a minimum tour), H is the current Held-Karp bound, and $\alpha^{(k)}$ is a real number satisfying $0 < \alpha^{(k)} \leq 2$. Following Held, Wolfe, and Crowder [1974], we start with $\alpha^{(0)} = 2$ and decrease $\alpha^{(k)}$ by some fixed factor after every block of iterations, where the size of a block depends on the number of nodes in the TSP we are solving and the amount of computation time we are willing to spend on obtaining the lower bound.

The entire method is summarized in the box below.

Held-Karp Iterative Method

Input

Graph: $G = (V, E)$ with edge costs $(c_e : e \in E)$ and $v_1 \in V$
Real number: U (a target value)
Positive real number: ITERATIONFACTOR (for example, 0.015)
Positive integer: MAXCHANGES (for example, 100)

Initialization

$y_v = 0\ \forall v \in V$, $H^* = -\infty$, TSMALL = 0.001, $\alpha = 2$, $\beta = 0.5$
NUMITERATIONS = ITERATIONFACTOR $\times |V|$

Algorithm

For $i = 1$ to MAXCHANGES
  For $k = 1$ to NUMITERATIONS
    Let T be the optimum 1-tree with respect to the edge costs $(c_{uv} - y_u - y_v : uv \in E)$ and let H be the corresponding Held-Karp bound;
    If $H > H^*$, then set $H^* = H$ (improvement);
    If T is a tour, then STOP.
    Let $t^{(k)} = \alpha(U - H)/\sum((2 - d_T(v))^2 : v \in V)$;
    If $t^{(k)} < TSMALL$, then STOP.
    Replace y_v by $y_v + t^{(k)}(2 - d_T(v))$ for all $v \in V$;
  Replace α by $\beta\alpha$;

If you carry out experiments with this method, you will notice that the bound you obtain for a given graph and edge costs will depend on the choices of the input parameters (particularly the target value U). Only experimenting with the settings will allow you to optimize the method for a particular class of problems. Also, you should keep in mind that many other choices for the step sizes are possible, and experiments may suggest an alternative scheme that performs better on your problems.

Linear Programming

Dantzig, Fulkerson, and Johnson [1954] proposed to attack the TSP with linear-programming methods. The approach they outlined is still the most effective method known for computing good lower bounds for the TSP. Their work also plays a very important role in the history of combinatorial optimization since it was the first time cutting-plane methods were used to solve a combinatorial problem. We will discuss their method further in the next section, but here we present their linear-programming relaxation to the TSP.

Let x be the characteristic vector of a tour. Then x satisfies

$$x(\delta(v)) = 2, \text{ for all } v \in V$$

Box 7.1: *Lagrangean Relaxation*
The Held-Karp lower bounding technique is an instance of a general integer programming method known as *Lagrangean relaxation* (Held and Karp [1971]). The method is appropriate for integer programming problems max $\{w^T x : Ax \leq b, x \text{ integral }\}$ for which the constraints $Ax \leq b$ can be split into two parts, $A_1 x \leq b_1$ and $A_2 x \leq b_2$, in such a way that "relaxed" problems of the form max $\{c^T x : A_2 x \leq b_2, x$ integral $\}$ can be solved efficiently. The method is based on the observation that for any vector $y \geq 0$ (where the number of components of y matches the number of inequalities in $A_1 x \leq b_1$) the value

$$L(y) = y^T b_1 + \max \{(w - y^T A_1)^T x : A_2 x \leq b_2, x \text{ integral }\}$$

is an upper bound on the original integer programming problem (since $y^T b_1 \geq y^T A_1 x$). By assumption, for any given vector y we can easily compute $L(y)$, so what we need is a way to find a good y (that is, one that gives a strong upper bound $L(y)$). This can be accomplished with a general iterative technique called *subgradient optimization* (Polyak [1967], Held and Karp [1971], Held, Wolfe, and Crowder [1974]). The kth step of the subgradient method is the following. Having the vector $y^{(k)}$, we compute an optimal solution $x^{(k)}$ to the problem

$$\max\{(w - y^{(k)^T}) A_1^T x : A_2 x \leq b_2, x \text{ integral }\}.$$

Now, for a specified step size $t^{(k)}$, we let

$$y^{(k+1)} = y^{(k)} - t^{(k)}(b_1 - A_1 x^{(k)})$$

and go on to the $(k+1)$st step. Polyak [1967] has shown that if the step sizes $t^{(0)}, t^{(1)}, \ldots$ are chosen so that they converge to 0 but not too fast (namely $\lim_{k\to\infty} t^{(k)} = 0$ and $\sum_{k=0}^{\infty} t^{(k)} = \infty$), then the sequence of upper bounds produced by the subgradient method converges to the optimal $L(y)$ bound.

$$0 \leq x_e \leq 1, \text{ for all } e \in E.$$

It is not true, however, that tours are the only integer solutions to this system, since every 2-factor (that is, disjoint union of circuits meeting all of the nodes) will appear in the solution set. To forbid these non-tours, we can add the inequalities

$$x(\delta(S)) \geq 2, \text{ for all } \emptyset \neq S \neq V$$

since any tour must both enter and leave such a set S, and thus will contain at least two edges from $\delta(S)$. These inequalities are known as *subtour constraints* since they forbid small circuits (or "subtours").

The Dantzig, Fulkerson, and Johnson relaxation of the TSP is

$$\begin{aligned} &\text{Minimize } \textstyle\sum(c_e x_e : e \in E) \\ &\text{subject to} \\ &x(\delta(v)) = 2, \text{ for all } v \in V \\ &x(\delta(S)) \geq 2, \text{ for all } \emptyset \neq S \neq V \\ &0 \leq x_e \leq 1, \text{ for all } e \in E. \end{aligned} \tag{7.1}$$

Note that any integral solution to (7.1) is a tour, so (7.1) can be used to create an integer-linear-programming formulation of the TSP. What is important for us, however, is that the optimal value of (7.1) is a lower bound on the cost of any tour. We call this the *subtour bound* for the TSP. We show below that the subtour bound is equal to the optimal Held-Karp bound!

To start off, choose some node $v_1 \in V$ and notice that we can restrict the set of subtour constraints to those sets S that do not contain v_1, since the constraints for S and $V \setminus S$ are identical. Furthermore, making use of the equations $x(\delta(v)) = 2$, we can write the subtour constraints in the "inside" form

$$x(\gamma(S)) \leq |S| - 1$$

(see Exercise 7.11). Also, the equation

$$x(\gamma(V \setminus \{v_1\})) = |V| - 2$$

is implied by the equations $x(\delta(v)) = 2$, and thus can be added as a redundant constraint to (7.1).

With these modifications, we can write (7.1) as

$$\text{Minimize } \textstyle\sum(c_e x_e : e \in E) \tag{7.2}$$

$$\text{subject to}$$

$$x(\delta(v)) = 2, \text{ for all } v \in V \tag{7.3}$$

$$x(\gamma(S)) \leq |S| - 1, \text{ for all } S \subseteq V,\ v_1 \notin S \tag{7.4}$$

$$x(\gamma(V \setminus \{v_1\})) = |V| - 2 \tag{7.5}$$

$$0 \leq x_e \leq 1, \text{ for all } e \in E. \tag{7.6}$$

These constraints look similar to the constraints of a linear-programming problem we saw in Chapter 2. Indeed, if we remove the equations (7.3) and the variables corresponding to the edges in $\delta(v_1)$, then we are left with the defining system for the convex hull of the spanning trees in the graph $G[V \setminus \{v_1\}]$. This is the connection with the Held-Karp bound. More directly, we can conclude that the cost of a minimum 1-tree in G is equal to

$$\min \{c^T x : \ x \text{ satisfies (7.4), (7.5), (7.6), and } x(\delta(v_1)) = 2\}. \tag{7.7}$$

To see how the node numbers come in, consider the form of the dual linear-programming problem of (7.2). We have a dual variable y_v for all $v \in V \setminus \{v_1\}$, together with a dual variable for each constraint in the 1-tree formulation (7.7). Now suppose we have an optimal solution to this dual linear-programming problem, and let $(y_v^* : v \in V \setminus \{v_1\})$ be the values of the variables $(y_v : v \in V \setminus \{v_1\})$. If we fix these variables at their values y_v^*, then the remaining variables constitute an optimal solution to the dual linear-programming problem of

$$\begin{gathered} \text{Minimize } \sum((c_{uv} - y_u^* - y_v^*)x_{uv} : uv \in E) \\ \text{subject to} \\ x \text{ satisfies (7.4), (7.5), (7.6)} \\ x(\delta(v_1)) = 2. \end{gathered}$$

But this is a 1-tree problem. So the optimal value of (7.2) is equal to the Held-Karp bound obtained using the node numbers $(y_v^* : v \in V \setminus \{v_1\})$ (setting the node number on v_1 to 0).

Conversely, the arguments show that for any set of node numbers we can construct a feasible solution to the dual of (7.2) with objective value equal to the corresponding Held-Karp bound. Thus, we have shown the following result.

Theorem 7.2 *The subtour bound is equal to the optimal Held-Karp bound.* ∎

The Held and Karp procedure can therefore be viewed as a heuristic for approximating the subtour bound. Direct methods for computing this bound will be discussed in the next section.

Exercises

7.8. Modify the edge costs in the graph given in Figure 7.11 so that they satisfy the triangle inequality, keeping the fact that the 1-tree bound is not equal to the optimal value of the TSP, but the best Held-Karp bound is equal to the optimum TSP value.

7.9. Give a graph and edge costs such that the best Held-Karp bound is not equal to the optimum value of the TSP.

7.10. Let $G = (V, E)$ be a graph with edge costs $(c_e : e \in E)$, and let T be a minimum spanning tree of G. Show that if v is a leaf of T, then an optimum 1-tree with v as node v_1 can be obtained be adding to the edge-set of T the edge joining v to its second nearest neighbor. Give an example of G, c, and T, where the best choice of v_1 (that is, the one giving the greatest 1-tree bound) is not a leaf of T.

7.11. Let $G = (V, E)$ be a graph with edge costs $c \in \mathbf{R}^E$. Show that the linear-programming problem (7.1) is equivalent to the linear-programming problem (7.2).

7.4 CUTTING PLANES

In the last section, we described an intuitively motivated procedure for constructing what is usually a good lower bound on the cost of an optimal solution to a TSP. We showed that this procedure was in fact a heuristic for obtaining a good feasible solution to the "subtour" linear-programming problem (7.1), which we restate here:

$$\begin{gathered} \text{Minimize } \sum(c_e x_e : e \in E) \qquad (7.8) \\ \text{subject to} \\ x(\delta(v)) = 2, \text{ for all } v \in V \qquad (7.9) \\ x(\delta(S)) \geq 2, \text{ for all } S \subseteq V, \ S \neq V, S \neq \emptyset \qquad (7.10) \\ 0 \leq x_e \leq 1, \text{ for all } e \in E. \qquad (7.11) \end{gathered}$$

Suppose we tried to solve this linear-programming problem directly. What problems would we encounter? One big obstacle is that the number of inequalities (7.10) is about the same as the number of distinct subsets of cities, or about $2^{|V|}$. Even if we notice that we do not need inequalities for both S and $V \setminus S$, and hence can limit ourselves to sets S satisfying $|S| \leq |V|/2$, we still need about $2^{|V|-1}$ of these inequalities.

Dantzig, Fulkerson, and Johnson overcame this obstacle by solving the linear-programming problem using the cutting-plane approach described in Section 6.7. We describe their approach in this section.

We begin by solving the linear-programming problem (7.8), (7.9), and (7.11). If the optimal solution happens to be the characteristic vector of a tour, then we can stop since this must be the solution to the TSP. If not, we will try to find some subtour constraints (7.10) violated by the optimal solution. We add these inequalities to our starting set and solve the resulting linear program.

We perform this process over and over. If we ever obtain a solution which does not violate any subtour constraints, then we have solved the original

linear-programming problem. If not, we add some violated subtour constraints to obtain the next problem.

If this iterative process is to work, there are two problems to solve. First, we must have an efficient method of checking an optimum solution to a relaxed problem to see whether it violates any subtour constraints from (7.10). Second, we must have an efficient way of solving the linear-programming problems that arise.

The first problem can be solved using results from Chapter 3, as we describe below.

For small TSPs the second problem is easily handled by any commercial-quality simplex-based linear-programming code. For larger TSPs, however, we will run into the difficulty of having to deal with linear-programming problems with a large number of variables. For example, the problems for our 1173-node sample TSP will have 687368 variables. In such a case, it is probably not a good idea to solve the problem directly. Instead, we handle the variables in a manner similar to the way we handle cutting planes: Start out with a linear-programming problem that contains only a subset of the variables and add in the remaining ones as they are needed. We need to explain what we mean by "as they are needed."

Suppose we select a set $E' \subseteq E$, such that the linear-programming problem

$$\begin{aligned} &\text{Minimize } \sum(c_e x_e : e \in E') && (7.12)\\ &\text{subject to}\\ &x(\delta(v)) = 2, \text{ for all } v \in V\\ &x(\delta(S)) \geq 2, \text{ for all } \emptyset \neq S \neq V\\ &0 \leq x_e \leq 1, \text{ for all } e \in E'. \end{aligned}$$

has a feasible solution. (A common choice is to take the union of a small number [say 10] of tours produced by the Core Lin-Kernighan Heuristic.) An optimal solution, x', to (7.12) can be extended to a feasible solution, x^*, to (7.8) by setting $x^*_e = 0$ for all $e \in E \setminus E'$. The trouble is that x^* may not be an optimal solution to (7.8).

To check optimality, let y', Y' be an optimal solution to the dual linear-programming problem of (7.12):

$$\begin{aligned} &\text{Maximize } \sum(2y_v : v \in V) + \sum(2Y_S : S \subseteq V, S \neq V, S \neq \emptyset) && (7.13)\\ &\text{subject to}\\ &y_u + y_v + \sum(Y_S : uv \in \delta(S), S \subseteq V, S \neq V, S \neq \emptyset) \leq c_{uv}, && (7.14)\\ &\qquad \text{for all } uv \in E'\\ &Y_S \geq 0, \text{ for all } S \subseteq V, S \neq V, S \neq \emptyset. && (7.15) \end{aligned}$$

If y', Y' is also feasible for the dual linear-programming problem of (7.8) (that is, the problem we obtain by replacing E' by E in (7.14)) then we know by

linear-programming duality that x^* is indeed optimal for the original linear-programming problem (7.8). Otherwise, we can add those edges $e \in E \setminus E'$ for which the corresponding constraint (7.14) is violated to our set E', resolve the linear-programming problem (7.12), and repeat the process.

This is another example of column generation. It is similar to the methods we described in Chapter 3 for multicommodity flows and in Chapter 5 for solving minimum-weight perfect matching problems on dense graphs. Combining column generation with the cutting-plane approach allows us to solve linear-programming problems that are both "long" and "wide."

Using this combined method, suppose that we eventually solve the linear-programming problem (7.8). Generally this solution will not be the characteristic vector of a tour. What then? We could stop with a lower bound which is usually pretty good. We could go on to branch-and-bound, as discussed in the next section. Or we could try to find some other class of cutting planes to add which would permit this process to continue.

We now discuss the cutting-plane generation in some detail.

Handling Subtour Constraints

Suppose x^* is a feasible solution to the (initial) linear-programming problem

$$\begin{aligned} &\text{Minimize } \sum(c_e x_e : e \in E) && (7.16)\\ &\text{subject to}\\ &x(\delta(v)) = 2, \text{ for all } v \in V && (7.17)\\ &0 \leq x_e \leq 1, \text{ for all } e \in E. && (7.18) \end{aligned}$$

We wish to determine whether all subtour constraints (7.10) are satisfied, and if not, find one or more that are violated.

If the solution falls apart into several components (that is, the graph with node-set V and edge-set $\{e \in E : x^*_e > 0\}$ is disconnected), then the node-set S of each component violates (7.10). This situation is easy to detect.

After several waves of cutting-plane addition, we will in general not have a disconnected solution. In this case, we need a more sophisticated separation algorithm.

For each edge e of G, define its *capacity* u_e to be x^*_e. Then the value $x^*(\delta(S))$ for any set S of nodes is precisely the same as the capacity of the cut in $\delta(S)$ in G. So we can apply the minimum cut methods we discussed in Section 3.5: There exists a set S of nodes that violates (7.10) if and only if some cut in G has capacity less than two.

Now we are in a good position. We solve the initial linear-programming relaxation. Then we can add violated subtour constraints as long as the solution is not sufficiently connected. Then we can use a minimum-cut algorithm to ensure that there are no violated subtour constraints at all. Each time we

add more subtour constraints, we can use the simplex algorithm to obtain a new optimal solution.

For our 1173-node sample TSP, the optimal value of (7.8) is 56361. As we would expect, this is slightly better than the bound we found using the Held-Karp method.

Suppose we terminate with a solution such as in Figure 7.13. It is an optimal solution to the linear-programming problem (7.8) if all costs are Euclidean, but it is not a tour. Below, we describe a class of cutting planes that is very useful in improving the lower bound in situations like this.

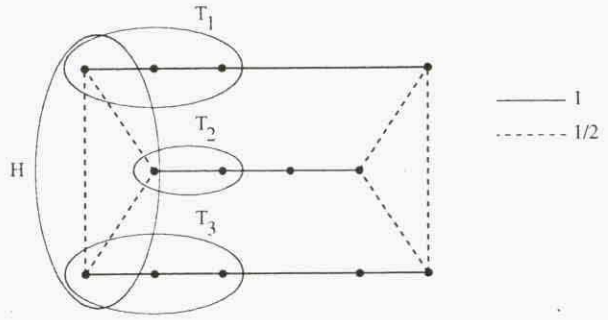

Figure 7.13. A fractional solution and violated comb

Comb Inequalities

A *comb* is defined by giving several subsets of nodes of the graph: We need one nonempty *handle* $H \subseteq V, H \neq V$ and $2k+1$ pairwise disjoint, nonempty *teeth* $T_1, T_2, ..., T_{2k+1} \subseteq V$, for k at least 1. (So the number of teeth is odd and at least 3.) We also require each tooth to have at least one node in common with the handle and at least one node that is not in the handle. See Figure 7.13.

Chvátal [1973a] and Grötschel and Padberg [1979] proved the following result.

Theorem 7.3 *Let C be a comb with handle H and teeth $T_1, T_2, ..., T_{2k+1}$ for $k \geq 1$. Then the characteristic vector x of any tour satisfies*

$$x(\gamma(H)) + \sum_{i=1}^{2k+1} x(\gamma(T_i)) \leq |H| + \sum_{i=1}^{2k+1} (|T_i| - 1) - (k+1).$$

Proof: Let x be a tour. Then x satisfies all the constraints (7.9)–(7.11). Add up the following constraints.

Equations (7.9) for the nodes in H.

Constraints (7.10) for the teeth T_i in the "inside form" $x(\gamma(T_i)) \leq |T_i| - 1$, (See Exercise 7.11).

Constraints $x_j \geq 0$ for the edges in $\delta(H)$ but not belonging to any tooth, but in the form $-x_j \leq 0$.

Constraints (7.10) for the sets $T_i \setminus H$ in the "inside form" $x(\gamma(T_i \setminus H)) \leq |T_i \setminus H| - 1$.

Constraints (7.10) for the sets $T_i \cap H$ (for those i such that T_i intersects H in more than one node) again in the "inside form" $x(\gamma(T_i \cap H)) \leq |T_i \cap H| - 1$.

We obtain

$$2x(\gamma(H)) + 2\sum_{i=1}^{2k+1} x(\gamma(T_i)) \leq 2|H| + 2\sum_{i=1}^{2k+1} (|T_i| - 1) - (2k+1).$$

Now, dividing through by 2, we obtain

$$x(\gamma(H)) + \sum_{i=1}^{2k+1} x(\gamma(T_i)) \leq |H| + \sum_{i=1}^{2k+1} (|T_i| - 1) - \frac{2k+1}{2}.$$

Since the left-hand side is integer-valued, we can round down the right-hand side, and get the desired result. ∎

Another way of stating this theorem is that

$$x(\gamma(H)) + \sum_{i=1}^{2k+1} x(\gamma(T_i)) \leq |H| + \sum_{i=1}^{2k+1} (|T_i| - 1) - (k+1) \qquad (7.19)$$

is a valid cutting plane. These are called *comb inequalities.*

Often when we see a fractional solution x in which there exists an odd circuit all of whose edges have the value 1/2, the node-set of the circuit forms the handle of a comb giving rise to a cutting plane that is violated by x. Again, see Figure 7.13. At the current time, there is no polynomial-time algorithm known for deciding in general whether a given (nonnegative) vector x violates some comb inequality.

When each tooth of a comb has exactly two nodes, we call the corresponding inequality a *blossom inequality.* This name comes from a connection to the 2-factor problem. (See Exercise 7.15.) For this special class of comb inequalities, Padberg and Rao [1982] showed that there is a polynomial-time separation routine. We describe their method below.

Let $G = (V, E)$ be a graph. A blossom inequality can be specified by a handle $H \subseteq V$ and a set of edges $A \subseteq \delta(H)$, with $|A|$ odd and at least 3. So the ends of the edges in A are the teeth of the corresponding comb.

When we are solving a TSP, we assume that the nonnegative vector x satisfies the equations (7.9). So we may write the blossom inequality

$$x(\gamma(H)) + x(A) \leq |H| + \frac{|A| - 1}{2} \tag{7.20}$$

in the form

$$x(\delta(H) \setminus A) - x(A) \geq 1 - |A|. \tag{7.21}$$

(See Exercise 7.16.) If we rewrite this inequality as

$$x(\delta(H) \setminus A) + (|A| - x(A)) \geq 1,$$

then the left-hand side looks similar to the capacity of the cut $\delta(H)$, except that the edges in $e \in A$ contribute $1 - x_e$ instead of the usual x_e. It is perhaps not surprising that our separation routine will make use of the minimum T-cut algorithm (see Section 6.8), as we now describe.

To speed up our computations, we begin by deleting from E all those edges e such that $x_e = 0$.

Now define a new graph G' by subdividing each edge $e \in E$ with two new nodes v'_e and v''_e, that is, if e has ends v and w then we replace e by the three edges vv'_e, $v'_e v''_e$, and $v''_e w$. Let T be the set of all new nodes v'_e, v''_e for all $e \in E$, and define edge weights $u \in \mathbf{R}^{E(G')}$ by setting $u_{vv'_e} = x_e$, $u_{v'_e v''_e} = 1 - x_e$, and $u_{v''_e w} = x_e$ for all edges $e = vw \in E$. See Figure 7.14.

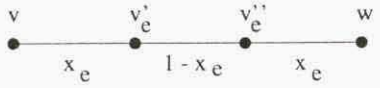

Figure 7.14. Subdivided edge

Suppose that the blossom inequality corresponding to $H \subseteq V$ and $A \subseteq \delta(A)$ is violated by x. Let $S \subset V(G')$ consist of H, together with the two new nodes v'_e, v''_e for all edges $e \in \gamma_G(H)$, and for each edge $e \in A$ the new node v'_e or v''_e that is joined by an edge in G' to a node in H. Then $S \cap T$ is odd and the capacity of $\delta_{G'}(S)$ is precisely

$$x(\delta(H) \setminus A) + (|A| - x(A)). \tag{7.22}$$

So $\delta_{G'}(S)$ is a T-cut in G' with capacity less than 1.

Conversely, let $\delta_{G'}(S)$ be a minimum T-cut in G' and suppose that its capacity is less than 1. Let $H = S \cap V$, and let A consist of those edges $e \in \delta_G(H)$ such that exactly one of the two new nodes v'_e or v''_e is in S and that node is adjacent to a node in H.

Using the fact that for any new node $t \in V(G') \setminus V$, the sum of the capacities of the two edges in $E(G')$ that are incident with t is exactly 1, it is easy to check that $|A|$ is odd and that the capacity of $\delta_{G'}(S)$ is at least (7.22). So the blossom inequality corresponding to H and A is violated by x.

The blossom separation problem can thus be solved by finding a minimum T-cut in G' using the algorithm described in Section 6.8. If the T-cut has capacity less than 1 then we can extract a violated blossom inequality. Otherwise, we conclude that no such inequality exists.

Using this method to optimize over the linear-programming problem we obtain by adding all blossom inequalities to (7.8), we obtain a lower bound of 56785 for our 1173-node sample TSP. This is a very good bound. It implies that the tour we found with Chained Lin-Kernighan in Section 7.3 has cost no more than 0.2% above that of an optimal tour.

A number of heuristics have been proposed for finding violated comb inequalities. One of the common ideas is to shrink certain subsets of nodes and look for a violated blossom inequality in the shrunk graph that corresponds to a violated comb in the original graph. (See Exercise 7.19.)

There are many more classes of valid cutting planes known for the TSP. We refer the reader to Grötschel and Padberg [1985], Jünger, Reinelt, and Rinaldi [1995], and Naddef [1990] for further discussions.

Exercises

7.12. Show how to solve linear program (7.16)–(7.18) as a minimum-cost flow problem. Use your construction to prove that there exists an optimal solution to (7.16)–(7.18) for which all variables have value $0, 1/2$, or 1.

7.13. Let x be a feasible solution to the linear program (7.16)–(7.18). Let F be the set of all $e \in E$ for which $x_e \neq 0$ or 1.

(a) Show that if F contains the edge-set of an even circuit then x can be expressed as a convex combination of two other feasible solutions, and so is not a vertex of the polyhedron defined by (7.17),(7.18).

(b) Show that if any connected component of F contains two or more odd circuits, then x is not a vertex of (7.17),(7.18).

(c) Show that if every component of F consists of a single odd circuit, then there exists a set of inequalities (7.18) that can be set to equations such that x is the unique vector satisfying these equations plus (7.17).

(d) State a necessary and sufficient condition, based on (a)–(c), for a vector x to be a vertex of (7.17),(7.18).

7.14. Prove that if x is a vertex of (7.17),(7.18), then at most $|V|$ components of x can have nonintegral values. Construct an example that shows that this bound can be attained.

7.15. Show that the blossom inequalities are satisfied by the characteristic vectors of the 2-factors of a graph G.

7.16. Show that in the presence of the equations (7.9), the blossom inequality (7.20) can be written as (7.21). What is the connection with the system (5.40)?

7.17. Show that in the presence of the equations (7.9), the comb inequality (7.19) can be written as

$$(x(\delta(H)) - 2) + \sum_{i=1}^{2k+1} (x(\delta(T_i) - 2) \geq 2k. \tag{7.23}$$

7.18. (Karger) Let k be a fixed positive integer and let x be a vector that satisfies (7.9), (7.10), and (7.11). Use Exercise 7.17 and the result of Exercise 3.65 (on page 85) to give a polynomial-time algorithm that with probability at least $1 - \frac{1}{n}$ will find some violated comb inequality having at most $2k + 1$ teeth if such an inequality exists.

7.19. Let $G = (V, E)$ be a graph and $x \in \mathbf{R}^E$ a nonnegative vector. Suppose that $S \subset V$ has the property that $x(\gamma(S)) = |S| - 1$ and consider the graph, G', we obtain by shrinking S to a single node, v, and replacing the parallel edges $e_1, \ldots, e_k$ between v and any other node by a single edge e having $x_e = x_{e_1} + \cdots + x_{e_k}$. Show that any violated comb inequality in G' corresponds to a violated comb inequality in G.

7.5 BRANCH AND BOUND

Cutting-plane methods can provide a very good lower bound on a TSP. Combining this with a tour produced by Chained Lin-Kernighan will typically leave only a small gap between the cost of the tour and the value of the bound. But suppose the gap is too large for a given application. How can we proceed further? The *branch and bound* method we present below is a common approach for doing just this. We will describe it in terms of the TSP, but the same principles apply to virtually any combinatorial optimization problem. Our description follows the TSP algorithm of Padberg and Rinaldi [1991].

Suppose we have a graph $G = (V, E)$ with edge costs $(c_e : e \in E)$ and let $\mathcal{T}$ denote the set of all tours of G. A lower bound on the TSP is a number B such that $c(T) \geq B$ for all $T \in \mathcal{T}$. A lower bounding technique is a method for producing such a number B. Now suppose we split $\mathcal{T}$ into two sets $\mathcal{T}_0$ and $\mathcal{T}_1$ such that $\mathcal{T}_0 \cup \mathcal{T}_1 = \mathcal{T}$. If we can produce numbers B_0 and B_1 such that $c(T) \geq B_0$ for all $T \in \mathcal{T}_0$ and $c(T) \geq B_1$ for all $T \in \mathcal{T}_1$, then the minimum of B_0 and B_1 is a lower bound on the TSP. The point of splitting $\mathcal{T}$ is that the extra structure in $\mathcal{T}_0$ and $\mathcal{T}_1$ may allow our lower bounding technique to perform better than it did on the entire set $\mathcal{T}$. This is the basis of branch and bound methods: We successively split the solution set and apply our lower bounding algorithm to each part. To see how this works, we describe how to use the cutting-plane lower bound in a branch and bound framework.

In this context, a natural way to partition the set of tours is to select an edge e and let $\mathcal{T}_0$ be those tours that do not contain e and let $\mathcal{T}_1$ be those that do contain e. So if we let P denote the original TSP, then we can work with this partition by considering a new problem P_0 obtained by setting $x_e = 0$ and a new problem P_1 obtained by setting $x_e = 1$.

Suppose that we have applied our cutting-plane methods to obtain a linear-programming relaxation, LP, of the original TSP. Then we can immediately write linear programs LP_0 and LP_1 (corresponding to the new problems) by adding the equations $x_e = 0$ and $x_e = 1$ to LP. If e is chosen carefully, we may obtain an immediate improvement in the lower bound by simply solving LP_0 and LP_1. Moreover, we can apply our cutting-plane generation routines to strengthen each of these linear programs, obtaining the relaxations LP_0' and LP_1'. Our lower bound will then be the minimum of the optimal values of LP_0' and LP_1'.

If we have not already established the optimality of our best tour, we can repeat the above process by taking one of the problems, say P_1, and some edge f, and creating the problems P_{10} and P_{11} by setting $x_f = 0$ and $x_f = 1$. Again, we can apply the cut generation routines to each of the new problems, obtaining the linear programs LP_{10}' and LP_{11}'. A bound on our TSP is then the minimum of the optimal values of LP_0', LP_{10}', and LP_{11}'. And we can go further, creating two new problems from either P_0, P_{10}, or P_{11}, and so on.

A general stage of the process can be described by a tree, where the nodes represent problems. (See Figure 7.15.) Each node Q that is not a leaf of the tree has two children, corresponding to the problems Q_0 and Q_1 we created from Q . (See Figure 7.15.) At any point, a lower bound for the original

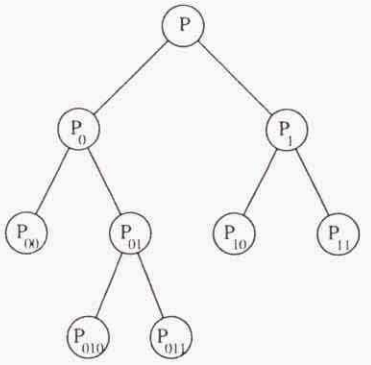

Figure 7.15. A branch and bound tree

TSP can be obtained by taking the minimum of the lower bounds we have computed for the problems corresponding to the leaves of the tree. We stop the procedure whenever this bound is greater than or equal to the cost of our

best tour (in which case we have proven that our tour is optimal) or if we have established a bound that is strong enough for our given application.

Notice that while working on some problem Q, we might well discover a tour that has cost less than the cost of our current best tour. In such a case, we should let this new tour be our best tour and continue the process.

The procedure we have described is the branch and bound method. The "branching" is the process of choosing a problem Q (from the leaves of the tree) to split into Q_0 and Q_1, and the edge e that determines the split. We still need to specify how these choices are made.

Since our goal is to improve the lower bound, at any point we could choose to process a problem Q whose linear-programming value is equal to the minimum over all leaves of the branch and bound tree. This will lead to a direct improvement in the lower bound. Other strategies may be adopted (for example, a depth first search of the branch and bound tree, where we always process one of the most recently created problems), but this choice is simple and has proved to work well in practice.

Box 7.2: *Branch and Bound for General Integer Programming*
The most successful methods that have been developed for solving general integer programming problems $\max\{wx : Ax \leq b, x \text{ integer}\}$ are based on branch and bound techniques. Branch and bound is a general scheme that requires two main decisions: how to branch and how to bound. The standard bounding method for integer programming is to solve the linear-programming (LP) relaxation of the current subproblem. This is used almost uniformly in commercially available integer programming codes. In some cases the LP relaxations are strengthened by the addition of cutting-planes derived from the structure of the given matrix A. On the branching side, many different schemes have been proposed. A common one is to choose some variable x_i that takes on a fractional value x_i^* in the optimal solution to the current LP relaxation, and create one new subproblem with the additional constraint $x_i \leq \lfloor x_i^* \rfloor$ and a second new subproblem with the additional constraint $x_i \geq \lceil x_i^* \rceil$. The rule for selecting the variable x_i often depends on user-specified priorities, the simplest being to choose the first variable x_i that takes on a fractional value. (So the user would input the problem with the variables in the order of their "importance" to the model.) For a detailed discussion of general integer programming methods see Nemhauser and Wolsey [1988].

Once we have selected a problem Q, what is a good choice for a "branching edge" e? If x^* is an optimal solution to the linear-programming relaxation for Q, then an obvious choice for e is some edge such that x_e^* is close to .5, since

then both $x_e = 0$ and $x_e = 1$ will hopefully force the linear program to move far away from the current optimal solution (and cause the optimal value to increase). Along the same lines, since we want to increase the objective function, we prefer more expensive edges e over cheaper edges. So, one proposal for a branching choice is to examine all edges e such that x_e^* is in some fixed interval surrounding .5, and select that edge having the greatest cost c_e.

We now have a rudimentary branch and bound scheme for the TSP. Of the many enhancements that can be made, we would like to mention one that seems particularly useful in practice. This enhancement concerns the generation of cutting-planes. Since we are using the inequalities we described in Section 4, all of the cuts we find while processing problem Q are actually valid inequalities for all problems in the branch and bound tree. So we can save the cuts in a pool, and search the pool for violated inequalities during any of our cut generation steps. The use of a pool is especially important when our generation routines are not exact separation methods, but rather heuristics for finding cuts in a particular class. In this case, the pool not only speeds up the search, it actually gives us a chance to find cutting planes that our heuristic would miss.

Using this type of branch and bound scheme, Applegate, Bixby, Chvátal, and Cook [1995] showed that the tour for the 1173-node problem we reported in Section 2 is in fact optimal. Their branch and bound tree contained 25 nodes. Moreover, they have solved a 7397-node problem to optimality with this approach. We have, of course, skimmed over all of the implementation details, and we refer the reader to the papers of Padberg and Rinaldi [1991] and Jünger, Reinelt, and Thienel [1994] for discussions of their realizations of these techniques.

Exercises

7.20. A collection of TSPs (many coming from industrial applications) can be found in Reinelt [1991]. Develop a computer implementation of some of the techniques described in this chapter and apply your code to one of these "TSPLIB" problems.