Mathematical Programming manuscript No.
(will be inserted by the editor)

David Applegate · Robert Bixby · Vašek Chvátal · William Cook*

Implementing the Dantzig-Fulkerson-Johnson Algorithm for Large Traveling Salesman Problems

Received: date / Revised version: date

Abstract. Dantzig, Fulkerson, and Johnson (1954) introduced the cutting-plane method as a means of attacking the traveling salesman problem; this method has been applied to broad classes of problems in combinatorial optimization and integer programming. In this paper we discuss an implementation of Dantzig et al.'s method that is suitable for TSP instances having 1,000,000 or more cities. Our aim is to use the study of the TSP as a step towards understanding the applicability and limits of the general cutting-plane method in large-scale applications.

1. The Cutting-Plane Method

The symmetric traveling salesman problem, or TSP for short, is this: given a finite number of "cities" along with the cost of travel between each pair of them, find the cheapest way of visiting all of the cities and returning to your starting point. The travel costs are symmetric in the sense that traveling from city X to city Y costs just as much as traveling from Y to X; the "way of visiting all of the cities" is simply the order in which the cities are visited.

The prominence of the TSP in the combinatorial optimization literature is to a large extent due to its success as an engine-of-discovery for techniques that have application far beyond the narrow confines of the TSP itself. Foremost among the TSP-inspired discoveries is Dantzig, Fulkerson, and Johnson's (1954) *cutting-plane method*, which can be used to attack any problem

$$\text{minimize } c^T x \text{ subject to } x \in \mathcal{S} \tag{1}$$

such that $\mathcal{S}$ is a finite subset of some R^m and such that an efficient algorithm to recognize points of $\mathcal{S}$ is available. This method is iterative; each of its

D. Applegate: Algorithms and Optimization Department, AT&T Labs – Research, Florham Park, NJ 07932, USA

R. Bixby: Computational and Applied Mathematics, Rice University, Houston, TX 77005, USA

V. Chvátal: Department of Computer Science, Rutgers University, Piscataway, NJ 08854, USA

W. Cook: Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, GA 30332, USA

* Supported by ONR Grant N00014-03-1-0040

iterations begins with a linear programming (LP) relaxation of (1), meaning a problem

$$\text{minimize } c^T x \text{ subject to } Ax \leq b \tag{2}$$

such that the polyhedron P defined as $\{x : Ax \leq b\}$ contains $\mathcal{S}$ and is bounded. Since P is bounded, we can find an optimal solution x^* of (2) such that x^* is an extreme point of P. If x^* belongs to $\mathcal{S}$, then it constitutes an optimal solution of (1); otherwise some linear inequality is satisfied by all the points in $\mathcal{S}$ and violated by x^*; such an inequality is called a *cutting plane* or simply a *cut*. In the latter case, we find a nonempty family of cuts, add them to the system $Ax \leq b$, and use the resulting tighter relaxation of (1) in the next iteration of the procedure.

Dantzig et al. demonstrated the power of their cutting-plane method by solving a 49-city instance of the TSP, which was an impressive size in 1954. The TSP is a special case of (1) with $m = n(n-1)/2$, where n is the number of the cities, and with $\mathcal{S}$ consisting of the set of the incidence vectors of all the Hamiltonian cycles through the set V of the n cities; in this context, Hamiltonian cycles are commonly called *tours*. In Dantzig et al.'s attack, the initial P consists of all vectors x, with components subscripted by edges of the complete graph on V, that satisfy

$$0 \leq x_e \leq 1 \quad \text{for all edges } e \tag{3}$$

and

$$\sum(x_e : v \in e) = 2 \quad \text{for all cities } v. \tag{4}$$

(Throughout this paper, we treat the edges of a graph as two-point subsets of its vertex-set: $v \in e$ means that vertex v is an endpoint of edge e; $e \cap Q \neq \emptyset$ means that edge e has an endpoint in set Q; $e - Q \neq \emptyset$ means that edge e has an endpoint outside set Q; and so on.) All but two of their cuts have the form $\sum(x_e : e \cap Q \neq \emptyset, e - Q \neq \emptyset) \geq 2$ such that Q is a nonempty proper subset of V. Dantzig et al. called such inequalities "loop constraints"; nowadays, they are commonly referred to as *subtour elimination inequalities*; we are going to call them simply *subtour inequalities*. (As for the two exceptional cuts, Dantzig et al. give ad hoc combinatorial arguments to show that these inequalities are satisfied by incidence vectors of all tours through the 49 cities and, in a footnote, they say "We are indebted to I. Glicksberg of Rand for pointing out relations of this kind to us.")

The original TSP algorithm of Dantzig et al. has been extended and improved by many researchers, led by the fundamental contributions of M. Grötschel and M. Padberg; surveys of this work can be found in Grötschel and Padberg (1985), Padberg and Grötschel (1985), Jünger et al. (1995, 1997), and Naddef (2002). The cutting-plane method is the core of nearly all successful approaches proposed to date for obtaining provably optimal solutions to the TSP, and it remains the only known technique for solving instances having more than several hundred cities. Beyond the TSP, the cutting-plane method has been applied to a host of NP-hard problems (see Jünger et al. (1995)), and is an important component of modern

mixed-integer-programming codes (see Marchand et al. (1999) and Bixby et al. (2000, 2003)).

In this paper we discuss an implementation of the Dantzig et al. algorithm designed for TSP instances having 1,000,000 or more cities; very large TSP instances arise is applications such as genome-sequencing (Agarwala et al. (2000)), but the primary aim of our work is to use the TSP as a means of studying issues that arise in the general application of cutting-plane algorithms for large-scale problems. Instances of this size are well beyond the reach of current (exact) solution techniques, but even in this case the cutting-plane method can be used to provide strong lower bounds on the optimal tour lengths. For example, we use cutting planes to show that the best known tour for a specific 1,000,000-city randomly generated Euclidean instance is no more than 0.05% from optimality. This instance was created by David S. Johnson in 1994, studied by Johnson and McGeoch (1997, 2002) and included in the DIMACS (2001) challenge test set under the name "E1M.0". Its cities are points with integer coordinates drawn uniformly from the 1,000,000 by 1,000,000 grid; the cost of an edge is the Euclidean distance between the corresponding points, rounded to the nearest integer.

The paper is organized as follows. In Section 2 we present separation algorithms for subtour inequalities and in Section 3 we present simple methods for separating a further class of TSP inequalities known as "blossoms"; in these two sections we consider only methods that can be easily applied to large problem instances. In Section 4 we discuss methods for adjusting cutting planes to respond to changes in the optimal LP solution x^*; again, we consider only procedures that perform well on large instances. In Section 5 we discuss a linear-time implementation of the "local cut" technique for generating TSP inequalities by mapping the space of variables to a space of very low dimension. The core LP problem that needs to be solved in each iteration of the cutting-plane algorithm is discussed in Section 6. Data structures for storing cutting planes are treated in Section 7 and methods for handling the $n(n-1)/2$ edges are covered in Section 8. In Section 9 we report on computational results for a variety of test instances.

The techniques developed in this paper are incorporated into the *Concorde* computer code of Applegate et al. (2003); the Concorde code is freely available for use in research studies.

2. Subtour Inequalities

A *separation algorithm* for a class $\mathcal{C}$ of linear inequalities is an algorithm that, given any x^*, returns either an inequality in $\mathcal{C}$ that is violated by x^* or a failure message. Separation algorithms that return a failure message only if all inequalities in $\mathcal{C}$ are satisfied by x^* are called *exact*; separation algorithms that may return a failure message even when some inequality in $\mathcal{C}$ is violated by x^* are called *heuristic*.

We present below several fast heuristics for subtour separation, and discuss briefly the Padberg and Rinaldi (1990a) exact subtour separation procedure.

2.1. The $x(S,T)$ notation

Let V be a finite set of cities, let E be the edge-set of the complete graph on V, and let w be a vector indexed by E. Given disjoint subsets S, T of V, we write $w(S,T)$ to mean

$$\sum(w_e : e \in E, e \cap S \neq \emptyset, e \cap T \neq \emptyset).$$

This notation is adopted from Ford and Fulkerson (1962); using it, the subtour inequality corresponding to S can be written as

$$x(S, V - S) \geq 2.$$

2.2. Parametric connectivity

Let G^* denote the graph with vertices V and whose edges are all e such that $x^*_e > 0$. If G^* is disconnected, then subtour inequalities violated by x^* are readily available: the vertex-set S of any connected component of G^* satisfies $x^*(S, V - S) = 0$.

The power of this separation heuristic is illustrated on a TSP instance generated in the same way as David Johnson's E1M.0 (cities are points with integer coordinates drawn uniformly from the 1,000,000 by 1,000,000 grid; the cost of an edge is the Euclidean distance between the corresponding points, rounded to the nearest integer) except that it has only 100,000 cities. We repeatedly apply the heuristic until G^* is connected; then we compare the final lower bound to the "subtour bound" obtained by optimizing over all subtour inequalities and also compare it to the length of the best tour we found (using the tour-merging heuristic of Cook and Seymour (2003)). The results

Gap to Subtour Bound	Gap to Optimal
0.394%	$\leq$ 1.111%

demonstrate that even this simple idea leads to a respectable bound for this geometric instance.

To improve on this connectivity heuristic, we observe that looking for subtour inequalities violated by x^* simply by listing connected components of G^* means throwing away much information about x^*: all nonzero x^*_e, regardless of their actual values, are treated the same. Such lack of discrimination can have its repercussions. For example, consider an x^* whose G^* is disconnected and let $S_1, \ldots, S_k$ denote the vertex-sets of the connected

components of G^*: slight perturbations of the components of x^* can make G^* connected while maintaining the conditions

$$0 \le x_e \le 1 \quad \text{for all edges } e,$$
$$\sum(x_e : v \in e) = 2 \quad \text{for all cities } v,$$

and $x^*(S_i, V - S_i) < 2$ for all i. In this example, we could have spotted the sets $S_1, S_2, \ldots, S_k$ as the vertex-sets of connected components of the graph with edges e such that $x^*_e > \varepsilon$ for some fixed positive ε. Pursuing this idea further, we make ε a parameter ranging from 1 down to 0 and arrive at Algorithm 2.1.

Algorithm 2.1 Testing connected components in a parametric family.

```
initialize an empty list L of sets;
F = the graph with the vertex-set of G* and with no edges;
for   all edges e of G* in a nonincreasing order of x*_e
do    if      the two endpoints of e
              belong to distinct connected components of F
      then add edge e to F;
              S =   vertex-set of the connected component of F
                    that contains e;
              if  x*(S, V − S) < 2 then add S to L end
              if    F consists of two connected components
              then return L
              end
      end
end
return L;
```

With m standing for the number of edges of G^*, each individual test for $x^*(S, V - S) < 2$ in Algorithm 2.1 may take time in $\Theta(m)$, which puts the total running time of Algorithm 2.1 in $\Theta(mn)$. It is quicker to first collect all the relevant S and then evaluate all the corresponding values of $x^*(S, V - S)$. All the relevant S may be recorded in a *decomposition forest* whose leaves are the n vertices of G^* and whose interior nodes are in a one-to-one correspondence with sets S for which Algorithm 2.1 tests the inequality $x^*(S, V - S) < 2$; each interior node w of the decomposition forest corresponds to the set S_w of all leaves of the decomposition forest that are descendants of w. One way of constructing the decomposition forest is Algorithm 2.2; there, roots are nodes equal to their own parents and ROOT(w) is the root of the tree that contains w.

Except for the evaluations of ROOT, straightforward implementations of Algorithm 2.2 take time $\Theta(n + m \log n)$, with the bottleneck $\Theta(m \log n)$ taken up by sorting these edges.

Algorithm 2.2 Constructing a decomposition forest.

```
for all cities w do parent(w) = w end
counter = n;
for all edges e of G* in a nonincreasing order of x*_e
do  u, v = the two endpoints of e;
    u* = ROOT(u), v* = ROOT(v);
    if   u* ≠ v*
    then get a new node w;
         parent(u*)= w, parent(v*)= w, parent(w)= w;
         counter = counter−1;
         if counter = 2 then return array parent end
    end
end
return array parent;
```

To implement the evaluations of ROOT, we may use the following triple of operations that maintain a changing family $\mathcal{F}$ of disjoint sets with each set in $\mathcal{F}$ named by one of its elements:

MAKESET(w),	with w in no set in $\mathcal{F}$, adds $\{w\}$ to $\mathcal{F}$;
FIND(u),	with u in some set in $\mathcal{F}$, returns the name of this set;
LINK(u, v),	with $u \neq v$, deletes the sets named u and v from $\mathcal{F}$ and adds their union to $\mathcal{F}$.

In our application, members of $\mathcal{F}$ are the sets S_w such that `parent`$(w)= w$; if, for each S_w in $\mathcal{F}$, we maintain a pointer `root` from the name of S_w to the root w of S_w, then we can evaluate ROOT(u) simply as `root`(FIND(u)). This policy is used in Algorithm 2.3.

A celebrated result of Tarjan (1975) (see also Tarjan and van Leeuwen (1984) and Chapter 2 of Tarjan (1983)) asserts that a simple and practical implementation of these three operations runs very fast: the time it requires to execute any sequence of k operations is in $O(k\alpha(k))$ with α the very slowly growing function commonly referred to as "the inverse of the Ackermann function." Hence Algorithm 2.3 can be implemented so that the total time spent on calls of MAKESET, FIND, and LINK is in $O(m\alpha(m))$.

The final step in the parametric connectivity procedure is to evaluate $x^*(S_w, V - S_w)$ for all nodes w of the decomposition tree; we describe this in Algorithm 2.4.

Harel and Tarjan (1984) and Schieber and Vishkin (1988) designed implementations of the first **for** loop in Algorithm 2.4 that run in time in $O(m)$; a straightforward recursive implementation of the second **for** loop runs in time $O(n)$; the third **for** loop runs in time $O(m)$.

Algorithm 2.3 An implementation of Algorithm 2.2.

```
for   all cities w
do    parent(w) = w, root(w) = w, MAKESET(w);
end
counter = n;
for   all edges e of G* in a nonincreasing order of x*_e
do    u, v = the two endpoints of e;
      u* = root(FIND(u)), v* = root(FIND(v));
      if    u* ≠ v*
      then  get a new node w;
            parent(u*)= w, parent(v*)= w, parent(w)= w;
            LINK(u*, v*), root(FIND(u)) = w;
            counter = counter−1;
            if counter = 2 then return array parent end
      end
end
return array parent;
```

Algorithm 2.4 Computing all the values of $x^*(S_w, V - S_w)$.

```
for   all edges e of G*
do    w(e) = the lowest common ancestor of two endpoints of e;
end
for   all nodes w of the decomposition forest
do    x*(S_w, V − S_w) = 2|S_w|;
end
for   all edges e of G*
do    x*(S_w(e), V − S_w(e)) = x*(S_w(e), V − S_w(e)) − 2x*_e
end
```

To illustrate its power on the 100,000-city instance described earlier in this section, we repeatedly apply Algorithm 2.1 until it returns without finding any cuts. The results

Gap to Subtour Bound	Gap to Optimal
0.029%	$\leq$ 0.746%

show a nice improvement over the bounds obtained by working only with the connected components of G^*.

2.3. Shrinking heuristic

Crowder and Padberg (1980) and Land (1979) developed a heuristic for subtour inequalities that is based on the intuitive notion of *shrinking* a subset of cities. Formally, shrinking a subset S of V means replacing V with $\bar{V}$ defined as $(V - S) \cup \{\sigma\}$ for some new vertex σ (representing the shrunk S) and replacing x with $\bar{x}$ defined on the edges of the complete graph with vertex set $\bar{V}$ by

$$\bar{x}_{\{\sigma,t\}} = x(S, \{t\}) \text{ for all } \ t \in V - S$$

and

$$\bar{x}_{\{u,v\}} = x_{\{u,v\}} \text{ for all } \ u, v \in V - S.$$

The heuristic proceeds by examining the components of the solution vector x^* and shrinking the ends $\{u, v\}$ of any edge satisfying $x^*_{\{u,v\}} = 1$. If this process creates an edge e satisfying $\bar{x}^*_e > 1$, then the set of original vertices S corresponding to the ends of e gives a violated subtour inequality; we record S and continue by shrinking the ends of e. We repeat this procedure until all edges e in the remaining graph satisfy $\bar{x}^*_e < 1$.

The shrinking heuristic is a very effective technique for finding violated subtour inequalities. Combining the shrinking cuts with the parametric connectivity heuristic, we obtain the results

Gap to Subtour Bound	Gap to Optimal
0.0009%	$\leq$ 0.7174%

for our 100,000-city instance. The lower bound produced in this way is very close to the optimal value over all subtour inequalities.

2.4. Subtour cuts from tour intervals

In this subsection we present another fast heuristic separation algorithm for subtour inequalities, allowing us to take advantage of any approximation to an optimal tour that we might have obtained by running a tour-finding heuristic. Our motivation for the design of this algorithm comes from the following argument:

> Since the optimal solution x^* of the current LP relaxation of our TSP instance approximates an optimal tour and since our best heuristically generated tour $\hat{x}$ approximates an optimal tour, the two vectors x^* and $\hat{x}$ are likely to approximate each other at least in the sense that $x^*(S, V - S) \approx \hat{x}(S, V - S)$ for most subsets S of V. In particular, sets S that minimize $x^*(S, V - S)$ subject to $S \subset V, S \neq V, S \neq \emptyset$ are likely to be found among sets S that minimize $\hat{x}(S, V - S)$ subject to the same constraints.

This argument may be not entirely convincing, but its conclusion was confirmed by our experience: in examples we have experimented with, many of the sets S such that $x^*(S, V-S) < 2$ and $S \neq V, S \neq \emptyset$ satisfied $\hat{x}(S, V-S) = 2$.

Sets S with $\hat{x}(S, V-S) = 2$ are characterized trivially: if $v_0 v_1 \dots v_{n-1} v_0$ is the cyclic order on V defined by the tour $\hat{x}$, then $\hat{x}(S, V-S) = 2$ if and only if S or $V-S$ is one of the *intervals* I_{it} $(1 \leq i \leq t \leq n-1)$ defined by

$$I_{it} = \{v_k : i \leq k \leq t\}.$$

Since $x^*(V-S, S) = x^*(S, V-S)$ for all subsets S of V, we are led to search for intervals I such that $x^*(I, V-I) < 2$. We might set our goal at finding just one such interval or we might set it at finding all of them. The objective accomplished by our computer code comes between these these two extremes: for each $i = 1, 2, \dots, n-2$, we

$$\text{find a } t \text{ that minimizes } x^*(I_{it}, V - I_{it}) \text{ subject to } i \leq t \leq n-1 \qquad (5)$$

and, in case $x^*(I_{it}, V - I_{it}) < 2$, we record the subtour inequality violated by x^*.

We describe an algorithm that solves the sequence of problems (5) in time that, with m standing again for the number of positive components of x^*, is in $\Theta(m \log n)$.

We reduce each of the problems (5) to a *minimum prefix-sum problem,*

> given a sequence $s_1, s_2, \dots, s_N$ of numbers,
> find a t that minimizes $\sum_{k=1}^{t} s_k$ subject to $1 \leq t \leq N$.

To elaborate, let us write

$$s(i,k) = \begin{cases} 0 & \text{if } 1 \leq k \leq i \leq n-1, \\ 1 - \sum_{i \leq j < k} x^*(\{v_j, v_k\}) & \text{if } 1 \leq i < k \leq n-1. \end{cases}$$

If $1 \leq t \leq i$, then $\sum_{k=1}^{t} s(i,k) = 0$; if $i \leq t \leq n-1$, then

$$\sum_{k=1}^{t} s(i,k) = \sum_{k=i+1}^{t} s(i,k) = (t-i) - \sum_{i \leq j < k \leq t} x^*(\{v_j, v_k\});$$

since

$$x^*(I_{it}, V - I_{it}) = 2|I_{it}| - 2 \sum_{i \leq j < k \leq t} x^*(\{v_j, v_k\}),$$

it follows that

$$\sum_{k=1}^{t} s(i,k) = \begin{cases} 0 & \text{if } t \leq i, \\ (x^*(I_{it}, V - I_{it})/2) - 1 & \text{if } t \geq i. \end{cases}$$

Hence problem (5) reduces to the problem

$$\text{find a } t \text{ that minimizes } \sum_{k=1}^{t} s(i,k) \text{ subject to } 1 \le t \le n-1. \qquad (6)$$

We solve the sequence of minimum prefix-sum problems (6) for $i = n-2$, $n-3, \ldots, 1$ in this order; after each decrement of i, we use the formula

$$s(i,k) = \begin{cases} s(i+1,k) & \text{if } k \le i, \\ 1 - x^*(\{v_i, v_k\}) & \text{if } k = i+1, \\ s(i+1,k) - x^*(\{v_i, v_k\}) & \text{if } k > i+1 \end{cases}$$

to update the input of (6). The resulting scheme is Algorithm 2.5.

Algorithm 2.5 Finding intervals I such that $x^*(I, V-I) < 2$.

initialize an empty list $\mathcal{L}$ of intervals;
for $k = 1, 2, \ldots, n-1$ **do** $s_k = 0$ **end**
for $i = n-2, n-3 \ldots, 1$
do $s_{i+1} = 1$;
  for all edges $\{v_i, v_k\}$ such that $x^*(\{v_i, v_k\}) > 0$ and $i < k$
  do $s_k = s_k - x^*(\{v_i, v_k\})$;
  end
  $t =$ a subscript that minimizes $\sum_{k=1}^{t} s_k$ subject to $1 \le t \le n-1$;
  if $\sum_{k=1}^{t} s_k < 0$ **then** add I_{it} to $\mathcal{L}$ **end**
end
return $\mathcal{L}$;

Each of the minimum prefix-sum problems

$$t = \text{a subscript that minimizes } \sum_{k=1}^{t} s_k \text{ subject to } 1 \le t \le n-1$$

in Algorithm 2.5 can be solved trivially in time that is in $\Theta(n)$; the total running time of the resulting implementation of Algorithm 2.5 is in $\Theta(n^2)$. Our implementation reduces this total to $\Theta(m \log n)$ by making use of the fact that each of the minimum prefix-sum problems that has to be solved is related to the minimum prefix-sum problem solved in the previous iteration.

Let us set this implementation in the more general framework of the following three operations:

INITIALIZE(N)	sets $s_1 = s_2 = \ldots = s_N = 0$,
RESET(k, `value`)	sets $s_k =$ `value`,
MIN-PREFIX	returns a t that minimizes $\sum_{k=1}^{t} s_k$ subject to $1 \le t \le N$.

We are going to describe a data structure that supports these three operations in such a way that

each INITIALIZE	takes time in $\Theta(N)$,
each RESET	takes time in $\Theta(\log N)$,
each MIN-PREFIX	takes time in $\Theta(\log N)$.

This data structure is a full binary tree T (meaning, as usual, any binary tree in which each node other than a leaf has both a left child and a right child) with leaves $u_1, u_2, \ldots, u_N$ in the left-to-right order and such that each node u of T holds a pair of numbers $s(u), p(u)$ defined recursively by

- $s(u_k) = p(u_k) = s_k$
  whenever u_k is a leaf,
- $s(u) = s(v) + s(w)$, $p(u) = \min\{p(v), s(v) + p(w)\}$
  whenever u is a node with left child v and right child w.

For each node u of T, there are subscripts $a(u)$ and $b(u)$ such that a leaf u_k is a descendant of u if and only if $a(u) \le k \le b(u)$; it is easy to see that

$$s(u) = \sum_{k=a(u)}^{b(u)} s_k \quad \text{and} \quad p(u) = \min\left\{ \sum_{k=a(u)}^{t} s_k : a(u) \le t \le b(u) \right\};$$

in particular, $p(\text{root}) = \min\{\sum_{k=1}^{t} s_k : 1 \le t \le N\}$. These observations suggest the implementations of INITIALIZE, RESET, and MIN-PREFIX that are spelled out in Algorithm 2.6.

To keep the running time of RESET and MIN-PREFIX in $\Theta(\log N)$, it is imperative to choose a T in INITIALIZE so that the depth of T is in $\Theta(\log N)$. Our choice is the *heap structure* with nodes $1, 2, \ldots, 2N-1$. There, every node i with $i < N$ has left child $2i$ and right child $2i+1$; nodes $N, N+1, \ldots, 2N-1$ are leaves in the left-to right order; the depth of this tree is $\lfloor \lg(2N-1) \rfloor$.

The resulting algorithm is presented as Algorithm 2.7.

We illustrated the power of Algorithm 2.5 on our 100,000-city instance. Taking the algorithm as the only source of cutting planes, the result (3.175% gap to the subtour bound) is worse than that obtained using just the connected components of G. This is not too surprising, given the restricted form of subtour cuts that are produced by Algorithm 2.5 (the argument that $x^*(S, V-S) \approx \hat{x}(S, V-S)$ does not hold well for the x^* vectors that appear after the addition of only subtour inequalities). If, however, we combine this algorithm with the heuristics presented earlier in this section, the results

Gap to Subtour Bound	Gap to Optimal
0.0008%	$\le$ 0.7173%

are a slight improvement over our previous lower bounds. Although this improvement is rather small, the cuts generated by this procedure are particularly useful in the Concorde code, where we store inequalities based on their representation as the union of intervals from the heuristic tour $\hat{x}$ (see Section 7).

Algorithm 2.6 Three operations for solving a sequence of minimum prefix-sum problems.

INITIALIZE(N):
T = a full binary tree of depth in $\Theta(\log N)$
    and with leaves $u_1, u_2, \ldots, u_N$ in the left-to right order;
for each node u of T **do** $s(u) = 0$, $p(u) = 0$ **end**

RESET(k, `value`):
$s(u_k) =$ `value`, $p(u_k) =$ `value`;
$u = u_k$;
while u is not the root
do u = parent of u;
    v = left child of u, w = right child of u;
    $s(u) = s(v) + s(w)$, $p(u) = \min\{p(v), s(v) + p(w)\}$;
end

MIN-PREFIX:
u = the root;
while u is not a leaf
do v = left child of u, w = right child of u;
    if $p(u) = p(v)$ **then** $u = v$ **else** $u = w$ **end**
end
return the subscript t for which $u = u_t$;

Algorithm 2.7 An efficient implementation of Algorithm 2.5.

initialize an empty list $\mathcal{L}$ of intervals;
INITIALIZE($n-1$);
for $i = n-2, n-3, \ldots, 1$
do RESET($i+1, 1$);
    for all edges $\{v_i, v_k\}$ such that $x^*(\{v_i, v_k\}) > 0$ and $i < k$
    do RESET($k, s(u_k) - x^*(\{v_i, v_k\})$);
    end
    if $p(\text{root}) < 0$
    then $t =$MIN-PREFIX;
        add I_{it} to $\mathcal{L}$;
    end
end
return $\mathcal{L}$;

2.5. Padberg-Rinaldi exact-separation procedure

Given a vector w of nonnegative edge weights, the global minimum-cut problem is to find a proper subset of vertices $S \subseteq V$ such that $w(S, V - S)$ is minimized. To solve the exact separation problem for subtour inequalities, one can let $w_e = x^*_e$ for each edge e, find a global minimum cut S, and check if $x^*(S, V - S) < 2$. This approach was adopted as early as Hong (1972), and it is a common ingredient in implementations of the Dantzig et al. algorithm.

Hong (1972) found the global minimum cut in his study by solving a series of $n - 1$ max-flow/min-cut problems (choose some vertex s and for each other vertex t find an (s, t)-minimum cut S_{st}, that is, a set $S_{st} \subseteq V$ with $s \in S_{st}$ and $t \notin S_{st}$, minimizing $w(S_{st}, V - S_{st})$). Padberg and Rinaldi (1990a) combined this approach with shrinking techniques (generalizing the procedure described above in Subsection 2.3) to obtain a method suitable for large TSP instances (in Padberg and Rinaldi (1991), their approach is used on examples having up to 2,392 cities).

We adopt the Padberg-Rinaldi approach in our code, using an implementation of Goldberg's (1985) algorithm to solve the (s, t)-minimum cut problems that arise. The effectiveness of the Padberg-Rinaldi shrinking rules together with the good practical performance of Goldberg's algorithm allows us to apply the separation algorithm to very large instances (we have carried out tests on up to 3,000,000 cities). On our 100,000-city instance, the exact subtour separation algorithm produced the result

Gap to Optimal
$\leq 0.7166\%$.

It is important to note that a single run of the Padberg-Rinaldi algorithm can produce a large collection of violated subtour inequalities, rather than just the single inequality determined by the global minimum cut. This is crucial for large-scale instances where subtour heuristics usually fail before the subtour bound is reached. This point is discussed by Levine (1999) in his study combining Concorde with Karger and Stein's (1996) random-contraction algorithm for global minimum cuts.

Further computational studies of global minimum cut algorithms can be found in Chekuri et al. (1997) and in Jünger et al. (2000). A conclusion of these studies is that the Padberg-Rinaldi shrinking method is an important pre-processing tool, even if the full procedure is not adopted.

We remark that Fleischer (1999) describes a fast algorithm for building a cactus representation of all minimum cuts and a practical implementation of her method is described in Wenger (2002), together with computational results for instances with up to 18,512 cities. We have not pursued this method in our implementation.

3. Fast blossoms

Let $S_0, S_1, \ldots, S_k$ be subsets of V such that k is odd, $S_1, \ldots, S_k$ are pairwise disjoint, and for each $i = 1, \ldots, k$ we have $S_i \cap S_0 \neq \emptyset$ and $S_i - S_0 \neq \emptyset$. Every incidence vector x of a tour satisfies

$$\sum (x(S_i, V - S_i) : i = 0, \ldots, k) \geq 3k + 1. \tag{7}$$

Inequalities (7) are known as *comb inequalities*. The name comes from Chvátal (1973), who introduced a variant of (7) with $S_1, \ldots, S_k$ not required to be pairwise disjoint but for each $i = 1, \ldots, k$ the subset S_i is required to S_0 in exactly one city. The present version is due to Grötschel and Padberg (1979a, 1979b), who have shown that it properly subsumes the original theme; we follow them in referring to S_0 as the *handle* of the comb and referring to $S_1, \ldots, S_k$ as its *teeth*.

After subtour inequalities, combs are the most common class of inequalities that have been used as cuts in TSP computations. Unlike subtours, however, no polynomial-time exact separation algorithm is known for this class; establishing the complexity of comb separation is an important open problem in the TSP (it is not known to be NP-hard). Recent progress on comb separation has been made by Letchford and Lodi (2002), giving a polynomial-time separation algorithm for the class of combs satisfying, for each $i = 1, \ldots k$, either $|S_i \cap S_0| = 1$ or $|S_i - S_0| = 1$. Their result generalizes the Padberg and Rao (1980) exact separation algorithm for *blossom inequalities*, that is, the case where $|S_i| = 2$ for each $i = 1, \ldots, k$. (Blossoms were defined by Edmonds (1965) in connection with two-matchings. See also Pulleyblank (1973).)

Computer codes for the TSP have relied on heuristics for comb separation, often combined with the Padberg-Rao exact algorithm for blossoms. Comb heuristics based on shrinking subsets of cities, followed by application of the Padberg-Rao algorithm, are described in Padberg and Grötschel (1985) and in Grötschel and Holland (1991; heuristics based on the structure of the graph $G_{1/2}$ having vertex-set V and edge-set $\{e : 0 < x^*_e < 1\}$ are described in Padberg and Hong (1980), in Padberg and Rinaldi (1990b), and in Naddef and Thienel (2002a). (Related separation algorithms can be found in Applegate et al (1995), Fleischer and Tardos (1999), and Letchford (2000).)

In our cutting-plane implementation for large-scale TSP instances, we use fast and simple heuristics for blossom inequalities, relying on the techniques described in the next section to extend the blossoms to more general comb inequalities.

Padberg and Hong (1980) propose a blossom-separation algorithm that builds the graph $G_{1/2}$ (as described above), and examines the vertex-sets $V_1, \ldots, V_q$ of the connected components of $G_{1/2}$. If for some i in $\{1, \ldots, q\}$ the set of edges

$$T = \{e : e \cap V_i \neq \emptyset, e - V_i \neq \emptyset, x^*_e = 1\}$$

has odd cardinality, then the blossom inequality with handle V_i and with teeth consisting of the sets of endpoints of T is violated by x^*. We refer to this technique for finding blossoms as the *odd-component heuristic*. (Variants of this method can be found in Hong (1972) and in Land (1979).)

Combining the odd-component heuristic with the subtour separation routines described in Section 2 produces the result

Gap to Optimal
$\leq 0.3387\%$

on our 100,000-city instance. To obtain this result, we ran the separation algorithms until they returned without any violated cuts. The addition of the blossom inequalities to the mix of cuts closed over half of the gap between the subtour bound and the length of the best tour we know.

The odd-component heuristic for blossoms suffers from the same problem as the connected-component heuristic for subtours we discussed in Section 2.2, namely, small perturbations in x^* can hide the odd components that make up the handles of the blossoms. We do not have an analogue of the parametric connectivity procedure in this case, but Grötschel and Holland (1987) proposed a method for handling a fixed perturbation ε in the heuristic. Their idea is to consider as possible handles the vertex-sets of the components of the graph G_ε having vertices V and edges $\{e : \varepsilon \leq x_e^* \leq 1-\varepsilon\}$. Let V_i denote the vertex-set of such a component, and let $e_1, \ldots, e_t$ be the edges in the set

$$\{e : e \cap V_i \neq \emptyset, e - V_i \neq \emptyset, x_e^* > 1 - \varepsilon\}$$

in a nonincreasing order of x_e^*; if t is even, then e_{t+1} is the edge in

$$\{e : e \cap V_i \neq \emptyset, e - V_i \neq \emptyset, x_e^* < \varepsilon\}$$

with the greatest x_e^* and t is incremented by one; now t is odd. For each odd integer k from 1 up to t such that

$$x^*(V_i, V - V_i) + \sum_{j=1}^{k} x^*(e_j, V - e_j) < 3k + 1,$$

Grötschel and Holland find a subtour inequality or a blossom inequality violated by x^*. If two of the edges e_j intersect inside V_i, then these two edges are removed from the collection and their intersection is deleted from V_i; if two of the edges e_j intersect outside V_i, then these two edges are removed from the collection and their intersection is added from V_i. Eventually, the collection consists of an odd number of disjoint edges; if there are at least three, then they form the teeth of a violated blossom inequality; if there is just one, then the handle alone yields a violated subtour inequality.

We have implemented a variation of the Grötschel-Holland heuristic, where we consider only $k = t$ or (if $x^*(e_{t-1}, V - e_{t-1}) + x^*(e_t, V - e_t) < 6$)

$k = t - 2$; in choosing the value of ε, we follow the recommendation of Grötschel and Holland and set $\varepsilon = 0.3$. Using this algorithm allows us to produce the result

Gap to Optimal
$\leq 0.3109\%$

for our 100,000-city instance. Here, we combined the Grötschel-Holland heuristic, the odd-component heuristic, and the subtour separation heuristics, running the cutting-plane procedures until no further cuts were produced.

4. Tightening and teething

Watching our implementation of the cutting-plane method run, we have observed that optimal solutions x^* of the successive LP relaxations often react to each new cut we add by shifting the defect prohibited by the cut to an area just beyond the cut's control. An obvious remedy is to respond to each slight adjustment of x^* with slight adjustments of our cuts. In this section we describe two methods for making these adjustments

To describe our cut-alteration procedures, it will be convenient to introduce standard notation for describing the types of cuts we consider in our computer code.

A *hypergraph* is an ordered pair $(V, \mathcal{F})$ such that $\mathcal{F}$ is a family of (not necessarily distinct) subsets of V; the elements of $\mathcal{F}$ are called the *edges* of the hypergraph. Given a hypergraph $(V, \mathcal{F})$ denoted $\mathcal{H}$, we write

$$\mathcal{H} \circ x = \sum (x(S, V - S) : \ S \in \mathcal{F})$$

and we let $\mu(\mathcal{H})$ stand for the minimum of $\mathcal{H} \circ x$ taken over the incidence vectors of tours through V. Every linear inequality satisfied by all the incidence vectors of tours through V is the sum of a linear combination of equations (4) and a *hypergraph inequality*,

$$\mathcal{H} \circ x \geq t$$

with $t \leq \mu(\mathcal{H})$.

We express all cutting planes used in our computer code as hypergraph inequalities. For example, if $\mathcal{H} = (V, \mathcal{F})$ is a comb with subsets $\mathcal{F} = \{S_0, S_1, \ldots, S_k\}$, then $\mu(\mathcal{H}) = 3k + 1$ and $\mathcal{H} \circ x \geq 3k + 1$ is the corresponding comb inequality.

4.1. Tightening an inequality

Let $\mathcal{H}$ be a hypergraph and let $E_1, E_2, \ldots E_m$ be the edges of $\mathcal{H}$. For each subset I of $\{1, 2, \ldots, m\}$, we set

$$\alpha(I, \mathcal{H}) = \bigcap_{i \in I} E_i - \bigcup_{i \notin I} E_i \, ;$$

we refer to each nonempty $\alpha(I,\mathcal{H})$ as an *atom* of $\mathcal{H}$. We write $\mathcal{H} \sqsubseteq \mathcal{H}'$ to signify that $\mathcal{H}$ and $\mathcal{H}'$ are hypergraphs with the same set of vertices and the same number of edges such that $\alpha(I,\mathcal{H}') \neq \emptyset$ whenever $\alpha(I,\mathcal{H}) \neq \emptyset$; it is not difficult to see that

$$\mathcal{H} \sqsubseteq \mathcal{H}' \text{ implies } \mu(\mathcal{H}') \geq \mu(\mathcal{H}).$$

By *tightening* a hypergraph $\mathcal{H}_0$, we mean attempting to modify $\mathcal{H}_0$ in such a way that the resulting hypergraph, $\mathcal{H}$, satisfies

$$\mathcal{H}_0 \sqsubseteq \mathcal{H} \quad \text{and} \quad \mathcal{H} \circ x^* < \mathcal{H}_0 \circ x^*. \tag{8}$$

Here, "attempting" and "modify" are the operative words: by tightening, we do *not* mean finding a solution $\mathcal{H}$ of (8). Rather, we mean a swift and not necessarily exhaustive search for a solution $\mathcal{H}$ of (8) such that each edge of $\mathcal{H}$ is either identical with the corresponding edge of $\mathcal{H}_0$, or it differs from it in just a few elements.

When the edges of $\mathcal{H}$ are $E_1, \ldots E_m$ and the edges of $\mathcal{H}'$ are $E'_1, \ldots E'_m$, we write $\mathcal{H}' \approx \mathcal{H}$ to signify that there is a subscript j such that E_i, E'_i are identical whenever $i \neq j$ and differ in precisely one element when $i = j$. Our starting point for tightening a prescribed hypergraph $\mathcal{H}_0$ is the *greedy search* specified in Algorithm 4.1.

Algorithm 4.1 Greedy search.

$\mathcal{H} = \mathcal{H}_0$;
repeat $\mathcal{H}'$ = hypergraph that minimizes $\mathcal{H}' \circ x^*$
subject to $\mathcal{H}_0 \sqsubseteq \mathcal{H}'$ and $\mathcal{H}' \approx \mathcal{H}$;
if $\mathcal{H}' \circ x^* < \mathcal{H} \circ x^*$ **then** $\mathcal{H} = \mathcal{H}'$ **else** return $\mathcal{H}$ **end**
end

One trouble with greedy search is that it terminates as soon as it reaches a local minimum, even though a better solution may be just around the corner. One remedy is to continue the search even when $\mathcal{H}' \circ x^* \geq \mathcal{H} \circ x^*$; now the best $\mathcal{H}$ found so far is not necessarily the current $\mathcal{H}$, and so it has to be stored and updated; furthermore, measures that make the search terminate have to be imposed. In our variation on this theme, the search terminates as soon as $\mathcal{H}' \circ x^* > \mathcal{H} \circ x^*$ (we prefer missing a solution of (8) to spending an inordinate amount of time in the search and letting $\mathcal{H}$ stray far away from $\mathcal{H}_0$), but it may go on when $\mathcal{H}' \circ x^* = \mathcal{H} \circ x^*$. We don't have to worry about storing and updating the best $\mathcal{H}$ found so far (this $\mathcal{H}$ is our current $\mathcal{H}$), but we still have to ensure that the modified search terminates.

Consider a vertex v of $\mathcal{H}$ and an edge E_i of $\mathcal{H}$. The *move* (v,i) is the replacement of E_i by $E_i \cup \{v\}$ in case $v \notin E_i$ and by $E_i - \{v\}$ in case $v \in E_i$.

In this terminology, $\mathcal{H}' \approx \mathcal{H}$ if and only if $\mathcal{H}'$ can be obtained from $\mathcal{H}$ by a single move; with $\mathcal{H} \oplus (v,i)$ standing for the hypergraph obtained from $\mathcal{H}$ by move (v,i) and with

$$\Delta(v,i) = \begin{cases} x^*(\{v\}, E_i) - 1 & \text{if } v \notin E_i, \\ 1 - x^*(\{v\}, E_i) & \text{if } v \in E_i, \end{cases}$$

we have

$$(\mathcal{H} \oplus (v,i)) \circ x^* = \mathcal{H} \circ x^* - 2\Delta(v,i).$$

In each iteration of our modified greedy search, we find a move (v,i) that maximizes $\Delta(v,i)$ subject to $\mathcal{H}_0 \sqsubseteq \mathcal{H} \oplus (v,i)$. If $\Delta(v,i) > 0$, then we substitute $\mathcal{H} \oplus (v,i)$ for $\mathcal{H}$; if $\Delta(v,i) < 0$, then we return $\mathcal{H}$; if $\Delta(v,i) = 0$, then we either substitute some $\mathcal{H} \oplus (w,j)$ with $\mathcal{H}_0 \sqsubseteq \mathcal{H} \oplus (w,j)$ and $\Delta(w,j) = 0$ for $\mathcal{H}$ or return $\mathcal{H}$. More precisely, we classify all the moves (v,i) with $\Delta(v,i) = 0$ into three categories by assigning to each of them a number $\chi(v,i)$ in $\{0,1,2\}$. If

$$\max\{\Delta(v,i) : \mathcal{H}_0 \sqsubseteq \mathcal{H} \oplus (v,i)\} = 0,$$

then we find a move (w,j) that maximizes $\chi(w,j)$ subject to $\mathcal{H}_0 \sqsubseteq \mathcal{H} \oplus (w,j)$ and $\Delta(w,j) = 0$; if $\chi(w,j) > 0$, then we substitute $\mathcal{H} \oplus (w,j)$ for $\mathcal{H}$; if $\chi(w,j) = 0$, then we return $\mathcal{H}$.

To describe this policy in different terms, let us write

$$(\Delta_1, \chi_1) \prec (\Delta_2, \chi_2) \text{ to mean that } \Delta_1 < \Delta_2 \text{ or else } \Delta_1 = \Delta_2 = 0,\ \chi_1 < \chi_2.$$

(Note that this $\prec$ is a partial order similar to, but not identical with, the lexicographic order on all the pairs (Δ, χ): in $\prec$, the second component of (Δ, χ) is used to break ties only if the first component equals zero.) In each iteration of our modified greedy search, we find a move (v,i) that

$$\text{maximizes } (\Delta(v,i), \chi(v,i)) \text{ subject to } \mathcal{H}_0 \sqsubseteq \mathcal{H} \oplus (v,i)$$

in the sense that no move (w,j) with $\mathcal{H}_0 \sqsubseteq \mathcal{H} \oplus (w,j)$ has $(\Delta(v,i), \chi(v,i)) \prec (\Delta(w,j), \chi(w,j))$. If $(0,0) \prec (\Delta(v,i), \chi(v,i))$, then we substitute $\mathcal{H} \oplus (v,i)$ for $\mathcal{H}$; if $(\Delta(v,i), \chi(v,i)) \preceq (0,0)$, then we return $\mathcal{H}$.

The values of $\chi(v,i)$ are defined in Algorithm 4.2.

This definition of $\chi(v,i)$ is motivated by three objectives:

(i) to make the search terminate,
(ii) to steer the search towards
returning a hypergraph with relatively small edges,
(iii) to improve the chances of
finding a hypergraph $\mathcal{H}$ with $\mathcal{H} \circ x^* < \mathcal{H}_0 \circ x^*$.

To discuss these three items, let S denote the sequence of moves made by the search. Each move (v,i) in S is either an *add*, $E_i \mapsto E_i \cup \{v\}$, or a *drop*, $E_i \mapsto E_i - \{v\}$; each move (v,i) in S is either *improving*, with $\Delta(v,i) > 0$, or *nonimproving*, with $\Delta(v,i) = 0$.

Algorithm 4.2 Tightening $\mathcal{H}_0$.

```
for  all vertices v and all i = 1, 2, ..., m
do   if v ∈ E_i then χ(v, i) = 1 else χ(v, i) = 2 end
end
H = H_0;
repeat (v, i) = move that
        maximizes (Δ(v, i), χ(v, i)) subject to H_0 ⊑ H ⊕ (v, i);
        if    (Δ(v, i), χ(v, i)) ≻ (0, 0)
        then if    χ(v, i) = 1
             then if    Δ(v, i) = 0
                  then  χ(v, i) = 0;
                  else  χ(v, i) = 2;
                  end
             else  χ(v, i) = 1;
             end
             H = H ⊕ (v, i);
        else  return H;
        end
end
```

For an arbitrary but fixed choice of v and i, let S^* denote the subsequence of S that consists of all the terms of S that equal (v, i). Trivially, adds and drops alternate in S^*; hence the definition of $\chi(v, i)$ guarantees that S^* does not contain three consecutive nonimproving terms; since S^* contains only finitely many improving terms, it follows that S^* is finite; in turn, as v and i were chosen arbitrarily, it follows that S is finite.

In S^*, a nonimproving drop cannot be followed by a nonimproving add (which has $\Delta = 0, \chi = 0$), but a nonimproving add can be followed by a nonimproving drop (which has $\Delta = 0, \chi = 1$). This asymmetry pertains to objective (ii): our search returns a hypergraph $\mathcal{H}$ such that, for all choices of v and i with $v \in E_i$, we have $\Delta(v, i) < 0$ or else $\mathcal{H}_0 \not\sqsubseteq \mathcal{H} \oplus (v, i)$.

The algorithm prefers nonimproving adds with $\chi > 0$ (these have $\chi = 2$) to nonimproving drops with $\chi > 0$ (these have $\chi = 1$). This asymmetry pertains to objective (iii): its role is to offset the asymmetry introduced by objective (ii). When no improving move is immediately available, we let edges of $\mathcal{H}$ shift by nonimproving moves in the hope of eventually discovering an improving move. Nonimproving adds that lead nowhere can get undone later by nonimproving drops, after which additional nonimproving drops may lead to a discovery of an improving move. However, nonimproving drops that lead nowhere cannot get undone later by nonimproving adds, and so they may forbid a sequence of nonimproving adds leading to a discovery of an improving move.

The importance of objective (ii) comes from the facts that (a) the LP solver works faster when its constraint matrix gets sparser and (b) cuts defined by hypergraphs with relatively small edges tend to have relatively small numbers of nonzero coefficients. (Part (b) of this argument could be criticized on the grounds that a hypergraph cut is invariant under complementing an edge and so, in problems with n cities, hypergraph edges of size k are just as good as edges of size $n-k$. However, this criticism is just a quibble: in hypergraphs that we use to supply cuts, edges tend to have sizes far smaller than $n/2$.)

To implement the instruction

(v,i) = move that
maximizes $(\Delta(v,i),\chi(v,i))$ subject to $\mathcal{H}_0 \sqsubseteq \mathcal{H} \oplus (v,i)$,

we maintain

- a priority queue $\mathcal{Q}$ of all pairs (w,j) such that $(\Delta(w,j),\chi(w,j)) \succ (0,0)$ and $\mathcal{H}_0 \sqsubseteq \mathcal{H} \oplus (w,j)$;

to speed up the update $\mathcal{Q}$ after each iteration, we maintain a number of auxiliary objects. Call vertices v and w *neighbors* if $x^*_{vw} > 0$ and write

$$V^*(\mathcal{H}) = \{w : w \text{ belongs to or has a neighbor in an edge of } \mathcal{H}\}.$$

We keep track of

- a superset V^* of $V^*(\mathcal{H})$

initialized as $V^*(\mathcal{H}_0)$ and updated by $V^* = V^* \cup V^*(\mathcal{H})$ after each iteration; for each w in V^*, we maintain

- the values of $(\Delta(w,j),\chi(w,j))$ in an array of length m.

(Note that each w outside V^* has $(\Delta(w,j),\chi(w,j)) = (-1,2)$ for all j.) In addition, we store

- the family $\mathcal{A}$ of all sets I such that $\alpha(I,\mathcal{H}_0)$ is an atom of $\mathcal{H}_0$;

for each I in $\mathcal{A}$, we maintain

- a doubly linked list $A(I)$ that holds the elements of $\alpha(I,\mathcal{H}) \cap V^*$

(the only I in $\mathcal{A}$ that may have $\alpha(I,\mathcal{H}) \not\subseteq V^*$ is the empty set, which labels the exterior atom of $\mathcal{H}$); for each w in V^*, we define $I(w) = \{j : w \in E_j\}$ and maintain

- a pointer $a(w)$ to $A(I(w))$, with $a(w) =$ `NULL` if $I(w) \notin \mathcal{A}$.

Initializing all this information and updating it after each iteration is a routine task; we will discuss its details only for the sake of clarity.

Inserting a new vertex w into V^* — with $\Delta(w,j)$ and $\chi(w,j)$ set as if w remained outside V^* — is a simple operation; for convenience, we set it apart as function INSERT(w) defined in Algorithm 4.3. The membership of a

Algorithm 4.3 INSERT(w).

$V^* = V^* \cup \{w\}$;
for $j = 1, 2, \ldots m$ **do** $\Delta(w, j) = -1$, $\chi(w, j) = 2$ **end**

move (w, j) in $\mathcal{Q}$ may change after each change of $\mathcal{H}$. Algorithm 4.4 defines a function MEMBERSHIP(w, j), that, given a move (w, j) such that $w \in V^*$, inserts (w, j) into $\mathcal{Q}$ or deletes it from $\mathcal{Q}$ as necessary. The initialization defined in Algorithm 4.5 replaces the first four lines of Algorithm 4.2; the update defined in Algorithm 4.6 replaces the line

$$\mathcal{H} = \mathcal{H} \oplus (v, i);$$

of Algorithm 4.2; the current $\mathcal{H}$ is represented by χ since

$$E_j = \{w \in V^* : \chi(w, j) = 1\}.$$

Algorithm 4.4 MEMBERSHIP(w, j).

if $a(w)$ points to a list that includes only one item
then if $(w, j) \in \mathcal{Q}$ **then** delete (w, j) from $\mathcal{Q}$ **end**
else if $(\Delta(w, j), \chi(w, j)) \succ (0, 0)$
then if $(w, j) \notin \mathcal{Q}$ **then** insert (w, j) into $\mathcal{Q}$ **end**
else if $(w, j) \in \mathcal{Q}$ **then** delete (w, j) from $\mathcal{Q}$ **end**
end
end

We apply the tightening procedure in our code in two ways. Firstly, we scan the cuts that we currently have in our LP and try tightening each of them in turn. Secondly, if a scan of the list of inequalities we maintain in a pool (inequalities that have at one time been added to the LP, but may no longer be present; see Section 6), does not produce sufficiently many cuts then we try tightening each one that is within some fixed tolerance ε of being violated by the current x^* (we use $\varepsilon = 0.1$).

Using these separation routines in combination with the algorithms presented in Section 2 and in Section 3, we obtained the result

Gap to Optimal
$\leq 0.1722\%$

for our 100,000-city instance.

Algorithm 4.5 Initialization.

```
V* = ∅;
for   i = 1, 2, ..., m
do    for   all v in E_i
      do    if v ∉ V* then INSERT(v) end
            χ(v, i) = 1;
            for   all neighbors w of v
            do    if w ∉ V* then INSERT(w) end
            end
      end
end

A = ∅;
for   all v in V*
do    I = {i : χ(v, i) = 1};
      if    I ∉ A
      then add I to A and initialize an empty list A(I);
      end
      insert v into A(I) and make a(v) point to A(I);
end

Q = ∅;
for   all v in V*
do    for   all neighbors w of v
      do    if    w ∈ V*
            then for   i = 1, 2, ... m
                 do    if    χ(w, i) ≠ χ(v, i)
                       then Δ(v, i) = Δ(v, i) + x*_vw;
                       end
                 end
            end
      end
      for i = 1, ..., m do MEMBERSHIP(v, i) end
end
```

Algorithm 4.6 Update.

if $a(v) \neq$ `NULL`
then $A =$ the list that $a(v)$ points to;
delete v from A;
if $|A| = 1$
then $w =$ the unique vertex in A;
for $j = 1, \ldots, m$ **do** MEMBERSHIP(w, j) **end**
end
end

$I = \{j : \chi(v, j) = 1\}$;
if $I \in \mathcal{A}$
then insert v into $A(I)$ and make $a(v)$ point to $A(I)$;
if $|A(I)| = 2$
then $w =$ the unique vertex in $A(I)$ other than v;
for $j = 1, \ldots, m$ **do** MEMBERSHIP(w, j) **end**
end
else $a(v) =$ `NULL`;
end

$\Delta(v, i) = -\Delta(v, i)$;
MEMBERSHIP(v, i);

for all neighbors w of v
do **if** $w \notin V^*$
then INSERT(w);
insert w into $A(\emptyset)$ and make $a(w)$ point to $A(\emptyset)$;
end
if $\chi(w, i) \equiv \chi(v, i) \bmod 2$
then $\Delta(w, i) = \Delta(w, i) - x^*_{vw}$;
else $\Delta(w, i) = \Delta(w, i) + x^*_{vw}$;
end
MEMBERSHIP(w, i);
end

4.2. Teething a comb inequality

The Grötschel-Holland heuristic mentioned in Section 3 builds a blossom inequality with a prescribed handle by attaching to this handle a set of two-node teeth selected in a greedy way. Its generalization would replace the set of two-node teeth of any comb inequality by an optimal set of two-node teeth. Unfortunately, such a procedure would have to take care to avoid teeth with nonempty intersection. Fortunately, if a two-node tooth intersects another tooth in a single node, then an even stronger inequality can be obtained by adjusting the hypergraph (or discovering a violated subtour inequality); if a two-node tooth intersects another tooth in more than a single node, then it must be contained in the other tooth. Concorde exploits this relationship in an algorithm which we refer to as *teething*.

More precisely, we say that a tooth is *big* if its size is least three, and *small* if its size is two; given a comb $\mathcal{H}_0$, we set

$$\begin{aligned}\Delta(\mathcal{H}_0) = \min\{\mathcal{H} \circ x^* - \mu(\mathcal{H}) : \mathcal{H} &\text{ is a comb such that} \\ &\mathcal{H} \text{ and } \mathcal{H}_0 \text{ have the same handle and} \\ &\text{all big teeth of } \mathcal{H} \text{ are teeth of } \mathcal{H}_0\};\end{aligned}$$

teething a comb $\mathcal{H}_0$ means finding either a comb $\mathcal{H}$ such that all big teeth of $\mathcal{H}$ are teeth of $\mathcal{H}_0$ and

$$\text{if } \Delta(\mathcal{H}_0) \le 0 \text{ then } \mathcal{H} \circ x^* - \mu(\mathcal{H}) \le \Delta(\mathcal{H}_0)$$

or else a subtour inequality violated by x^*.

As a preliminary step in teething, we test the input $\mathcal{H}_0$ for the property

$$x(S, V - S) \ge 2 \text{ whenever } S \text{ is an edge of } \mathcal{H}_0; \tag{9}$$

if (9) fails, then we have found a subtour inequality violated by x^*. The remainder of the algorithm consists of three parts.

The first part involves the notion of a *pseudocomb*, which is just like a comb except that its small teeth are allowed to intersect — but not to be contained in — other teeth. More rigorously, a pseudocomb is a hypergraph with edge-set $\{H\} \cup \mathcal{T}$ such that

- if $T \in \mathcal{T}$, then $T \cap H \neq \emptyset$ and $T - H \neq \emptyset$,
- if $T_1, T_2 \in \mathcal{T}$, $T_1 \neq T_2$, and $|T_1| \ge 3, |T_2| \ge 3$, then $T_1 \cap T_2 = \emptyset$,
- if $T_1, T_2 \in \mathcal{T}$ and $|T_1| = 2, |T_2| \ge 3$, then $T_1 \not\subset T_2$,
- $|\mathcal{T}|$ is odd.

Given an arbitrary hypergraph $\mathcal{H}$ with edge-set $\mathcal{E}$, we write

$$\nu(\mathcal{H}) = \begin{cases} 3|\mathcal{E}| - 2 & \text{if } |\mathcal{E}| \text{ is even,} \\ 3|\mathcal{E}| - 3 & \text{if } |\mathcal{E}| \text{ is odd;} \end{cases}$$

note that $\nu(\mathcal{H}) = \mu(\mathcal{H})$ whenever $\mathcal{H}$ is a comb. In the first part, we

(i) find a pseudocomb $\mathcal{H}_1$ that minimizes $\mathcal{H}_1 \circ x^* - \nu(\mathcal{H}_1)$
subject to the constraints that $\mathcal{H}_1$ and $\mathcal{H}_0$ have the same handle
and that all big teeth of $\mathcal{H}_1$ are teeth of $\mathcal{H}_0$.

Trivially, $\mathcal{H}_1 \circ x^* - \nu(\mathcal{H}_1) \leq \Delta(\mathcal{H}_0)$. If $\mathcal{H}_1 \circ x^* - \nu(\mathcal{H}_1) \geq 0$, then we give up; else we proceed to the second part. This part involves the notion of a *generalized comb*, which is just a comb without some of its teeth. More rigorously, a generalized comb is a hypergraph with edge-set $\{H\} \cup \mathcal{T}$ such that

- $H \neq \emptyset$ and $H \neq V$,
- if $T \in \mathcal{T}$, then $T \cap H \neq \emptyset$ and $T - H \neq \emptyset$,
- if $T_1, T_2 \in \mathcal{T}$ and $T_1 \neq T_2$, then $T_1 \cap T_2 = \emptyset$.

In the second part,we

(ii) find either a generalized comb $\mathcal{H}_2$ such that
all teeth of $\mathcal{H}_2$ are teeth of $\mathcal{H}_1$ and
$\mathcal{H}_2 \circ x^* - \nu(\mathcal{H}_2) \leq \mathcal{H}_1 \circ x^* - \nu(\mathcal{H}_1)$
or else a subtour inequality violated by x^*;

in the third part, we

(iii) we find either a comb $\mathcal{H}$ such that
all teeth of $\mathcal{H}$ are teeth of $\mathcal{H}_2$ and
$\mathcal{H} \circ x^* - \mu(\mathcal{H}) \leq \mathcal{H}_2 \circ x^* - \nu(\mathcal{H}_2)$
or else a subtour inequality violated by x^*.

To discuss implementations of part (i), let H denote the handle of $\mathcal{H}_0$, set

$$\mathcal{S} = \{T : |T \cap H| = 1,\ |T - H| = 1\},$$

and let $\mathcal{B}$ denote the set of big teeth of $\mathcal{H}_0$. In this notation, we may state the problem of finding $\mathcal{H}_1$ as

$$\begin{array}{ll} \text{minimize} & \sum_{T \in \mathcal{T}} (x^*(T, V - T) - 3) \\ \text{subject to} & \mathcal{T} \subseteq \mathcal{S} \cup \mathcal{B}, \\ & T_1 \not\subset T_2 \text{ whenever } T_1, T_2 \in \mathcal{T}, \\ & |\mathcal{T}| \equiv 1 \bmod 2. \end{array} \tag{10}$$

Now let us write

$$\mathcal{S}^* = \{e \in \mathcal{S} : x^*_e > 0\}.$$

We claim that (10) has a solution $\mathcal{T}$ such that $\mathcal{T} \subseteq \mathcal{S}^* \cup \mathcal{B}$, and so the problem of finding $\mathcal{H}_1$ can be stated as

$$\begin{array}{ll} \text{minimize} & \sum_{T \in \mathcal{T}} (x^*(T, V - T) - 3) \\ \text{subject to} & \mathcal{T} \subseteq \mathcal{S}^* \cup \mathcal{B}, \\ & T_1 \not\subset T_2 \text{ whenever } T_1, T_2 \in \mathcal{T}, \\ & |\mathcal{T}| \equiv 1 \bmod 2. \end{array} \tag{11}$$

To justify this claim, note that property (9) with $S = H$ guarantees $\mathcal{S}^* \neq \emptyset$ and consider an arbitrary solution $\mathcal{T}$ of (10). If some T_1 in $\mathcal{S} - \mathcal{S}^*$ belongs

to $\mathcal{T}$, then $x(T_1, V - T_1) = 4 > x(T, V - T)$ for all T in $\mathcal{S}^*$; hence $\mathcal{T}$ must include a set T_2 other than T_1; since property (9) with $S = T_2$ guarantees $x(T_2, V - T_2) \geq 2$, the removal of T_1 and T_2 from $\mathcal{T}$ yields another solution of (10). Iterating this process, we arrive at the desired conclusion.

Concorde's way of solving problem (11) is specified in Algorithm 4.7. The initial part of this algorithm computes sets $R_0(i), R_1(i)$ with $i = 0, 1, 2, \ldots, k$ such that each $R_t(i)$ with $1 \leq i \leq k$

$$\begin{aligned} &\text{minimizes } \textstyle\sum_{T \in \mathcal{T}} (x^*(T, V - T) - 3) \\ &\text{subject to } \mathcal{T} \subseteq \mathcal{S}^*, \\ &\qquad T \subset T_i \text{ whenever } T \in \mathcal{T}, \\ &\qquad |\mathcal{T}| \equiv t \bmod 2 \end{aligned}$$

and $R_t(0)$

$$\begin{aligned} &\text{minimizes } \textstyle\sum_{T \in \mathcal{T}} (x^*(T, V - T) - 3) \\ &\text{subject to } \mathcal{T} \subseteq \mathcal{S}^*, \\ &\qquad T \not\subset T_i \text{ whenever } T \in \mathcal{T} \text{ and } 1 \leq i \leq k, \\ &\qquad |\mathcal{T}| \equiv t \bmod 2. \end{aligned}$$

The i-th iteration of the last **for** loop computes sets $R_0(0), R_1(0)$ such that $R_t(0)$

$$\begin{aligned} &\text{minimizes } \textstyle\sum_{T \in \mathcal{T}} (x^*(T, V - T) - 3) \\ &\text{subject to } \mathcal{T} \subseteq \mathcal{S}^* \cup \mathcal{B}, \\ &\qquad T_1 \not\subset T_2 \text{ whenever } T_1, T_2 \in \mathcal{T}, \\ &\qquad T \not\subset T_j \text{ whenever } T \in \mathcal{T} \text{ and } i < j \leq k, \\ &\qquad |\mathcal{T}| \equiv t \bmod 2. \end{aligned}$$

Lemma 1. *Let A, B and $T_1, \ldots T_s$ be distinct sets such that*

$$T_j \subseteq (A - B) \cup (B - A) \text{ and } |T_j \cap (A - B)| = |T_j \cap (B - A)| = 1$$

whenever $1 \leq j \leq s$. Then

$$\begin{aligned} x(A \cup B, V - (A \cup B)) + x(A \cap B, V - (A \cap B)) \leq \\ x(A, V - A) + x(B, V - B) + \sum_{j=1}^{s} x(T_j, V - T_j) - 4s. \end{aligned}$$

Proof. Observe that

$$\begin{aligned} \textstyle\sum (x_e^* : e \subseteq A) + \sum (x_e^* : e \subseteq B) + \sum_{j=1}^{s} \sum (x_e^* : e \subseteq T_j) \leq \\ \textstyle\sum (x_e^* : e \subseteq A \cup B) + \sum (x_e^* : e \subseteq A \cap B). \end{aligned}$$

Substituting $|S| - \frac{1}{2} x^*(S, V - S)$ for each $\sum (x_e^* : e \subseteq S)$ in this inequality yields the desired result. □

Algorithm 4.7 First part of a teething iteration.

H = the handle of $\mathcal{H}_0$;
$T_1, T_2, \ldots, T_k$ = the big teeth of $\mathcal{H}_0$;
for $i = 0, 1, \ldots k$ **do** $\rho_0(i) = 0$, $\rho_1(i) = +\infty$, $R_0(i) = R_1(i) = \emptyset$ **end**
for all vertices u in H
do **for** all vertices v such that $x^*_{uv} > 0$ and $v \notin H$
do **if** $u, v \in T_j$ for some j **then** $i = j$ **else** $i = 0$ **end**
$P_0 = R_0(i)$, $P_1 = R_1(i)$, $\nu_0 = \rho_0(i)$, $\nu_1 = \rho_1(i)$;
if $\nu_1 + (1 - 2x^*_{uv}) < \nu_0$
then $\rho_0(i) = \nu_1 + (1 - 2x^*_{uv})$, $R_0(i) = P_1 \cup \{\{u, v\}\}$;
end
if $\nu_0 + (1 - 2x^*_{uv}) < \nu_1$
then $\rho_1(i) = \nu_0 + (1 - 2x^*_{uv})$, $R_1(i) = P_0 \cup \{\{u, v\}\}$;
end
end
end

for $i = 1, 2, \ldots, k$
do **if** $x(T_i, V - T_i) - 3 < \rho_1(i)$
then $R_1(i) = \{T_i\}$, $\rho_1(i) = x(T_i, V - T_i) - 3$
end
$P_0 = R_0(0)$, $P_1 = R_1(0)$, $\nu_0 = \rho_0(0)$, $\nu_1 = \rho_1(0)$;
if $\nu_1 + \rho_1(i) < \nu_0 + \rho_0(i)$
then $\rho_0(0) = \nu_1 + \rho_1(i)$, $R_0(0) = P_1 \cup R_1(i)$;
else $\rho_0(0) = \nu_0 + \rho_0(i)$, $R_0(0) = P_0 \cup R_0(i)$;
end
if $\nu_0 + \rho_1(i) < \nu_1 + \rho_0(i)$
then $\rho_0(0) = \nu_0 + \rho_1(i)$, $R_1(0) = P_0 \cup R_1(i)$;
else $\rho_0(0) = \nu_1 + \rho_0(i)$, $R_1(0) = P_1 \cup R_0(i)$;
end
end
$\mathcal{H}_1$ = hypergraph with edge-set $\{H\} \cup R_1(0)$;

Let us use Lemma 1 to show that Algorithm 4.8 maintains the following invariant:

$$x^*(H, V - H) + \sum_{T \in \mathcal{T}} x^*(T, V - T) \leq \tag{12}$$
$$\begin{cases} \mathcal{H}_1 \circ x^* - \nu(\mathcal{H}_1) + 3|\mathcal{T}| + 1 & \text{if } |\mathcal{T}| \text{ is odd,} \\ \mathcal{H}_1 \circ x^* - \nu(\mathcal{H}_1) + 3|\mathcal{T}| & \text{if } |\mathcal{T}| \text{ is even.} \end{cases}$$

For this purpose, consider first the change of H and $\mathcal{T}$ made by an iteration of the first **for** loop. Lemma 1 with $A = H$, $B = T'$, and $T_1, \ldots T_s$ the sets

Algorithm 4.8 Second part of a teething iteration.

H = the handle of $\mathcal{H}_1$;
$\mathcal{T}$ = the set of all teeth of $\mathcal{H}_1$;
for all vertices v outside H
do **if** v belongs to at least two sets in $\mathcal{T}$
then T' = largest set in $\mathcal{T}$ such that $v \in T'$;
if $x(T' \cap H, V - (T' \cap H)) \geq 2$
then $H = H \cup T'$;
$\mathcal{T} = \{T \in \mathcal{T} : T \not\subseteq H\}$;
else return $T' \cap H$;
end
end
end
for all vertices v in H
do **if** v belongs to at least two sets in $\mathcal{T}$
then T' = largest set in $\mathcal{T}$ such that $v \in T'$;
if $x(T' - H, V - (T' - H)) \geq 2$
then $H = H - T'$;
$\mathcal{T} = \{T \in \mathcal{T} : T \cap H \neq \emptyset\}$;
else return $T' - H$;
end
end
end
$\mathcal{H}_2$ = the hypergraph with edge-set $\{H\} \cup \mathcal{T}$;

in $\mathcal{T}$ distinct from T' and contained in $H \cup T'$ guarantees that the left-hand side of (12) drops by at least $4s + 2$. If s is odd, then the right-hand side of (12) drops by $3(s + 1)$; if s is even, then the right-hand side of (12) drops by at most $3(s + 1) + 1$; in either case, the right-hand side of (12) drops by at most $4s + 2$. Hence all iterations of the first **for** loop maintain invariant (12); the same argument with $V - H$ in place of H shows that all iterations of the second **for** loop maintain invariant (12).

In particular, if Algorithm 4.8 gets around to constructing $\mathcal{H}_2$, then

$$\mathcal{H}_2 \circ x^* - \nu(\mathcal{H}_2) \leq \mathcal{H}_1 \circ x^* - \nu(\mathcal{H}_1).$$

Note that (12) and the assumption $\mathcal{H}_1 \circ x^* - \nu(\mathcal{H}_1) < 0$ guarantee that Algorithm 4.8 maintains the invariant

- $\mathcal{T} \neq \emptyset$;

trivially, it maintains the invariant

- if $T \in \mathcal{T}$, then $T \cap H \neq \emptyset$ and $T - H \neq \emptyset$.

The first **for** loop changes H and $\mathcal{T}$ so that

if $T_1, T_2 \in \mathcal{T}$ and $T_1 \neq T_2$, then $(T_1 \cap T_2) - H = \emptyset$;
the second **for** loop changes H and $\mathcal{T}$ so that

- if $T_1, T_2 \in \mathcal{T}$ and $T_1 \neq T_2$, then $T_1 \cap T_2 = \emptyset$.

To summarize, if the algorithm gets around to constructing $\mathcal{H}_2$, then $\mathcal{H}_2$ is a generalized comb with at least one tooth.

A practical variation on Algorithm 4.8 takes the conditions

$$x^*(T' \cap H, V - (T' \cap H)) \geq 2 \quad \text{and} \quad x^*(T' - H, V - (T' - H)) \geq 2$$

for granted: skipping the persistent tests speeds up the computations. If the resulting hypergraph $\mathcal{H}_2$ satisfies

$$\mathcal{H}_2 \circ x^* - \nu(\mathcal{H}_2) \leq \mathcal{H}_1 \circ x^* - \nu(\mathcal{H}_1),$$

then all is well and we proceed to part (iii); else we simply give up. In the latter case, we know that some big tooth T of $\mathcal{H}_1$ satisfies

$$x^*(T \cap H, V - (T \cap H)) < 2 \quad \text{or} \quad x^*(T - H, V - (T - H)) < 2;$$

the option of finding this T now remains open, even though its appeal may be marred by the fact that violated subtour inequalities can be spotted reasonably fast from scratch.

Part (iii) of teething is trivial. If $\mathcal{H}_2$ has at most two teeth, then

$$\mathcal{H}_2 \circ x^* < 2(1 + |\mathcal{T}|),$$

and so at least one edge S of $\mathcal{H}_2$ has $x^*(S, V - S) < 2$. If the number of teeth of $\mathcal{H}_2$ is at least three and odd, then we may set $\mathcal{H} = \mathcal{H}_2$. If the number of teeth of $\mathcal{H}_2$ is at least four and even, then we may let $\mathcal{H}$ be $\mathcal{H}_2$ with an arbitrary tooth deleted: we have $\mu(\mathcal{H}) = \nu(\mathcal{H}_2) - 2$ and (9) guarantees that $\mathcal{H} \circ x^* \leq \mathcal{H}_2 \circ x^* - 2$; in this case, the number of distinct cuts we obtain is the number of teeth of $\mathcal{H}_2$.

In our code, as a heuristic separation algorithm, we apply teething to each comb inequality in the current LP. Adding this routine to our mix of cuts improves the bound on our 100,000-city instance to

$$\begin{array}{|c|}\hline \text{Gap to Optimal} \\ \hline \leq 0.1671\%. \\ \hline \end{array}$$

5. Local cuts

The development of the cut-finding techniques of Section 2 and Section 3 conforms to the following paradigm:

1. describe a class $\mathcal{C}$ of linear inequalities that are satisfied by the set $\mathcal{S}$ of incidence vectors of all the tours through the n cities and then
2. design an efficient (exact or heuristic) separation algorithm for $\mathcal{C}$.

The cut-finding technique of the present section deviates from this paradigm: the kinds of cuts it finds are unspecified and unpredictable.

The idea is to first map $\mathcal{S}$ and x^* to a space of very low dimension by some suitable linear mapping ϕ and then, using standard general-purpose methods, to look for linear inequalities

$$a^T \overline{x} \leq b \tag{13}$$

that are satisfied by all points of $\phi(\mathcal{S})$ and violated by $\phi(x^*)$: every such inequality yields a cut,

$$a^T \phi(x) \leq b, \tag{14}$$

separating $\mathcal{S}$ from x^*. (Boyd's (1993, 1994) variation on a theme by Crowder, Johnson, and Padberg (1983) can also be outlined in these terms, with ϕ a projection onto a small set of coordinates, but his general-purpose method for finding cuts in the low-dimensional space is radically different from ours.) In our implementation of this idea, ϕ is defined by a partition of V into pairwise disjoint nonempty sets $V_0, V_1, \ldots, V_k$ and can be thought of as shrinking each set V_i into a single node: formally,

$$\phi : \mathbf{R}^{n(n-1)/2} \to \mathbf{R}^{(k+1)k/2}$$

is defined by $\phi(x) = \overline{x}$ with

$$\overline{x}_{ij} = x(V_i, V_j) \quad \text{whenever} \quad 0 \leq i < j \leq k.$$

Let us write

$$\overline{V} = \{0, 1, \ldots, k\}.$$

With ϕ defined by shrinking each set V_i into the single node i, the change of variable from $\overline{x}$ in (13) to x in (14) is particularly easy to implement when (13) is a hypergraph inequality: substitution from the definitions of $\overline{x}_{ij}$ converts each linear function $\overline{x}(\overline{Q}, \overline{V} - \overline{Q})$ to the linear function $x(Q, V - Q)$ where Q is the set of all cities that are mapped into $\overline{Q}$ by the function that shrinks V onto $\overline{V}$.

Shrinking V onto $\overline{V}$ reduces each tour through V to a spanning closed walk through $\overline{V}$; it reduces the incidence vector x of the tour to a vector $\overline{x}$ such that

- each $\overline{x}_e$ is a nonnegative integer,
- the graph with vertex-set $\overline{V}$ and edge-set $\{e : \overline{x}_e > 0\}$ is connected,
- $\sum(\overline{x}_e : v \in e)$ is even whenever $v \in \overline{V}$;

we will refer to the set of all the vectors $\overline{x}$ with these three properties as *tangled tours* through $\overline{V}$. This notion, but not the term, was introduced by Cornuéjols, Fonlupt, and Naddef (1985); they refer to the convex hull of the set of all tangled tours through a prescribed set as the *graphical traveling salesman polytope*.

For a particular choice of ϕ, finding a hypergraph inequality that is satisfied by all tangled tours through $\overline{V}$ and violated by $\overline{x}^*$ — if such an

Algorithm 5.1 A scheme for collecting TSP cuts.

initialize an empty list $\mathcal{L}$ of cuts;
for selected small integers k and
    partitions of V into nonempty sets $V_0, V_1, \ldots, V_k$
do $\overline{x}^* =$ the vector obtained from x^* by shrinking each V_i into singleton i;
    $\overline{V} = \{0, 1, \ldots, k\}$;
    if $\overline{x}^*$ lies outside the graphical traveling salesman polytope on $\overline{V}$
    then find a hypergraph inequality that is
            satisfied by all tangled tours through $\overline{V}$ and violated by $\overline{x}^*$;
        change its variable from $\overline{x}$ to x, and
        add the resulting hypergraph inequality to $\mathcal{L}$;
    end
end
return $\mathcal{L}$;

inequality exists at all — is relatively easy; we try many different choices of ϕ; the resulting scheme is summarized in Algorithm 5.1.

Our computer code deviates from the scheme of Algorithm 5.1 in minor ways. When it comes to including a new cut in $\mathcal{L}$, we are more selective than Algorithm 5.1 suggests. We accept only cuts that have integer coefficients and integer right-hand sides; to produce such cuts, our variation on the theme of Algorithm 5.1 uses integer arithmetic whenever necessary. In addition, cuts that are violated only slightly by $\overline{x}^*$ are of little use to a cutting-plane algorithm; instead of adding such a weak cut to $\mathcal{L}$, we move on to the next choice of $V_0, V_1, \ldots, V_k$ as if $\overline{x}^*$ belonged to the graphical traveling salesman polytope on $\overline{V}$.

In certain additional cases, we may also fail to return a cut separating $\overline{x}^*$ from all tangled tours through $\overline{V}$, even though $\overline{x}^*$ lies well outside the graphical traveling salesman polytope on $\overline{V}$. This happens whenever computations using integer arithmetic are about to create overflow and whenever the number of iterations or recursive calls of some procedure has exceeded a prescribed threshold. In such circumstances, we once again move on to the next choice of $V_0, V_1, \ldots, V_k$ just as if $\overline{x}^*$ belonged to the graphical traveling salesman polytope on $\overline{V}$.

5.1. Making choices of $V_0, V_1, \ldots, V_k$

Concorde's choices of $V_0, V_1, \ldots, V_k$ in Algorithm 5.1 are guided by x^* in a way similar to that used by Christof and Reinelt (1996) in their algorithm for finding cuts that match templates from a prescribed large catalog. First, it constructs once and for all an equivalence relation on V in such a way that each equivalence class $V^\star$ of this relation satisfies

$$x^*(V^\star, V - V^\star) = 2;$$

then it makes many different choices of $V_0, V_1, \ldots, V_k$ in such a way that each of $V_1, \ldots, V_k$ is one of these equivalence classes and $V_0 = V - (V_1 \cup \ldots \cup V_k)$.

With W standing for the set of the equivalence classes on V, the first stage amounts to preshrinking V onto W; making each of the many different choices of $V_0, V_1, \ldots, V_k$ in the second stage means choosing a small subset of W and shrinking the entire remainder of W onto a single point. In terms of the preshrunk set W, each choice of $V_0, V_1, \ldots, V_k$ in the second stage zooms in onto a relatively small part of the problem—typically k is at most thirty or so and $|W|$ may run to hundreds or thousands—and effectively discards the rest. For this reason, we developed the habit of referring to the cuts produced by Algorithm 5.1 as *local cuts*. In terms of the original V, each of the sets $V_1, \ldots, V_k$ could be quite large, which makes the qualifier "local" a misnomer. Still, a crisp label for the cuts produced by Algorithm 5.1 is convenient to have and "local cuts" seems to be as good a name as any other that we managed to think up.

The equivalence relation is constructed by iteratively shrinking two-point sets into a single point. At each stage of this process, we have a set W and a mapping $\pi : W \to 2^V$ that defines a partition of V into pairwise disjoint subsets $\pi(w)$ with $w \in W$. Initially, $W = V$ and each $\pi(w)$ is the singleton $\{w\}$; as long as there are distinct elements u, v, w of W such that

$$x^*(\pi(u), \pi(v)) = 1 \quad \text{and} \quad x^*(\pi(u), \pi(w)) + x^*(\pi(v), \pi(w)) = 1, \qquad (15)$$

we keep replacing $\pi(u)$ by $\pi(u) \cup \pi(v)$ and removing v from W; when there are no u, v, w with property (15), we stop. (During this process, we may discover pairs u, v with $x^*(\pi(u), \pi(v)) > 1$, in which case x^* violates the subtour inequality $x(Q, V - Q) \geq 2$ with $Q = \pi(u) \cup \pi(v)$.)

To make the many different choices of $V_1, \ldots, V_k$, we first set the value of a parameter t that nearly determines the value of k in the sense that $t - 3 \leq k \leq t$. For large-scale instances we simply set $t = t_{\max}$, where $t_{\max}$ is a prescribed integer (at least 8). For smaller instances, we let the value of t range between 8 and $t_{\max}$. More precisely, the search always begins with $t = 8$. Whenever a value of t is set, Concorde adds all the resulting cuts produced by Algorithm 5.1 to the LP relaxation of our problem and it solves the tightened LP relaxation; if the increase in the value of the relaxation is not satisfactory and $t < t_{\max}$, then the next iteration takes place with t incremented by one.

For each w in W, we choose a subset C of W so that $w \in C$ and $t - 3 \leq |C| \leq t$; the corresponding $V_1, \ldots, V_k$ are the $\pi(v)$ with $v \in C$. The choice of C is guided by the graph with vertex-set W, where u and v are adjacent if and only if $x^*(\pi(u), \pi(v)) > \varepsilon$ for some prescribed zero tolerance ε: starting at w, we carry out a breadth-first search through this graph, until we collect a set C of $t - 3$ vertices. If there are any vertices u outside this C such that $x^*(\pi(u), \pi(v)) = 1$ for some v in C, then we keep adding these vertices u to C as long as $|C| \leq t$.

It seems plausible that such a crude way of choosing C can be improved. However, we found its performance satisfactory; none of the alternatives that we tried appeared to work better.

5.2. Testing the **if** *condition in Algorithm 5.1*

Each choice of $V_0, V_1, \ldots, V_k$ yields an $\overline{x}^*$, the vector obtained from x^* by shrinking each V_i into singleton i; this $\overline{x}^*$ defines a set of tangled tours through $\overline{V} = \{0, 1, \ldots, k\}$, which we call *strongly constrained*; specifically, a tangled tour is strongly constrained if, and only if,

$$\begin{aligned} \overline{x}_e &= 0 \quad \text{for all } e \text{ such that } \overline{x}^*_e = 0, \\ \overline{x}_e &= 1 \quad \text{for all } e \text{ such that } \overline{x}^*_e = 1 \text{ and } e \subset \{1, 2, \ldots, k\}, \\ \textstyle\sum(\overline{x}_e : u \in e) &= 2 \quad \text{for all } u \text{ in } \{1, 2, \ldots, k\}. \end{aligned}$$

Since every tangled tour $\overline{x}$ satisfies the inequalities

$$\begin{aligned} \overline{x}_e &\geq 0 \quad \text{for all } e, \\ \textstyle\sum(\overline{x}_e : u \in e) &\geq 2 \quad \text{for all } u, \\ \overline{x}(e, \overline{V} - e) &\geq 2 \quad \text{for all } e, \end{aligned}$$

and since

$$\textstyle\sum(\overline{x}^*_e : u \in e) = 2 \text{ for all } u \text{ in } \{1, 2, \ldots, k\},$$

$\overline{x}^*$ belongs to the graphical traveling salesman polytope on $\overline{V}$ (defined as the convex hull of the set of all tangled tours through $\overline{V}$) if and only if it belongs to the convex hull of the set of all strongly constrained tangled tours through $\overline{V}$. Algorithm 5.2, given $\overline{x}^*$, returns either

- a vector a and a scalar b such that the inequality $a^T x \leq b$ is satisfied by all strongly constrained tangled tours through $\overline{V}$ and violated by $\overline{x}^*$

or

- a failure message indicating that $\overline{x}^*$ belongs to the convex hull of the set of all strongly constrained tangled tours through $\overline{V}$.

To sketch our implementation of Algorithm 5.2, let

> $\overline{E}_{1/2}$ denote the set of all the edges e such that $e \subset \{1, 2, \ldots, k\}$, $\overline{x}^*_e \neq 0$, $\overline{x}^*_e \neq 1$.

The significance of $\overline{E}_{1/2}$ comes from the fact that every strongly constrained tangled tour $\overline{x}$ satisfies

$$\begin{aligned} \overline{x}_{0u} &= 2 - \textstyle\sum(\overline{x}_e : e \subset \{1, 2, \ldots, k\}, u \in e) \quad \text{for all } u \text{ in } \{1, 2, \ldots, k\}, \\ \overline{x}_e &= 0 \quad \text{for all } e \text{ such that } e \subset \{1, 2, \ldots, k\} \text{ and } \overline{x}^*_e = 0, \\ \overline{x}_e &= 1 \quad \text{for all } e \text{ such that } e \subset \{1, 2, \ldots, k\} \text{ and } \overline{x}^*_e = 1, \end{aligned}$$

and so the condition

Algorithm 5.2 Testing the **if** condition in Algorithm 5.1:

```
if   there is a strongly constrained tangled tour x̄ through V̄
then make x̄ the only specimen in a collection of
     strongly constrained tangled tours through V̄;
     repeat if   some linear inequality a^T x̄ ≤ b is
                 satisfied by all x̄ in the collection and violated by x̄*
             then find a strongly constrained tangled tour x̄ through V̄
                  that maximizes a^T x̄;
                  if   a^T x̄ ≤ b
                  then return a and b;
                  else add x̄ to the collection;
                  end
             else return a failure message;
             end
     end
else return [0, 0, ..., 0]^T and −1;
end
```

some linear inequality $a^T\overline{x} \le b$ is
satisfied by all $\overline{x}$ in the collection and violated by $\overline{x}^*$

in Algorithm 5.2 is equivalent to the condition

some linear inequality $a^T\overline{x} \le b$ with $a_e = 0$ whenever $e \notin \overline{E}_{1/2}$ is
satisfied by all $\overline{x}$ in the collection and violated by $\overline{x}^*$.

To test this condition, Concorde solves a linear programming problem. With

$\psi(\overline{x})$ standing for the restriction of $\overline{x}$
on its components indexed by elements of $\overline{E}_{1/2}$,

with A the matrix whose columns $\psi(\overline{x})$ come from specimens $\overline{x}$ in the collection, and with e standing—as usual—for the vector $[1, 1, \ldots, 1]^T$ whose number of components is determined by the context, this problem—in variables s, λ, w—reads

$$\begin{array}{lrcl} \text{maximize} & s & & \\ \text{subject to} & s\psi(\overline{x}^*) - A\lambda + w &= & 0, \\ & -s \qquad + e^T\lambda \qquad &= & 0, \\ & w &\le & e, \\ & -w &\le & e, \\ & \lambda &\ge & 0. \end{array} \tag{16}$$

Since its constraints can be satisfied by setting $s = 0, \lambda = 0, w = 0$, problem (16) either has an optimal solution or else it is unbounded. In the former

case, the simplex method applied to (16) finds also an optimal solution of its dual,

$$\begin{array}{ll} \text{minimize} & e^T(u+v) \\ \text{subject to} & a^T\psi(\overline{x}^*) - b = 1, \\ & -a^T A + be^T \geq 0, \\ & a + u - v = 0, \\ & u \geq 0, \quad v \geq 0; \end{array} \tag{17}$$

this optimal solution provides a vector a and a scalar b such that the linear inequality $a^T\psi(\overline{x}) \leq b$ is satisfied by all $\overline{x}$ in the collection and violated by $\overline{x}^*$; in fact, a and b

$$\text{maximize} \quad \frac{a^T\psi(\overline{x}^*) - b}{\|a\|_1}$$

subject to the constraint that $a^T\psi(\overline{x}) \leq b$ for all $\overline{x}$ in the collection. In the latter case, (17) is infeasible, and so no linear inequality is satisfied by all $\overline{x}$ in the collection and violated by $\overline{x}^*$.

To find specimens $\overline{x}$ for the collection, we use a function ORACLE that, given an integer vector c, returns either a strongly constrained tangled tour $\overline{x}$ through $\overline{V}$ that maximizes $c^T\overline{x}$ or the message "infeasible" indicating that no tangled tour through $\overline{V}$ is strongly constrained. Concorde implements ORACLE as two algorithms in tandem: if a primitive branch-and-bound algorithm fails to solve the instance within a prescribed number of recursive calls of itself, then we switch to a more sophisticated branch-and-cut algorithm. To reconcile

- the floating-point arithmetic of the simplex method,
  which finds a and b,

with

- the integer arithmetic of ORACLE,
  which uses a and b,

Concorde approximates the floating-point numbers by rationals with a small common denominator; for this purpose, it uses the continued fraction method (see, for instance, Schrijver (1986)).

5.3. Separating $\overline{x}^*$ from all tangled tours

If $\overline{x}^*$ lies outside the graphical traveling salesman polytope, then Algorithm 5.2 returns a linear inequality which is satisfied by all strongly constrained tangled tours through $\overline{V}$ and violated by $\overline{x}^*$. Just converting this inequality into a hypergraph inequality which is satisfied by all tangled tours through $\overline{V}$ and violated by $\overline{x}^*$would be easy; we make the task more difficult by requiring this hypergraph inequality to induce a facet of the graphical traveling salesman polytope on $\overline{V}$. Concorde does it in three phases:

– in PHASE 1, we find a linear inequality that
  separates $\overline{x}^*$ from all *moderately constrained* tangled tours and

induces a facet of their convex hull.

– in PHASE 2, we find a linear inequality that
separates $\overline{x}^*$ from all *weakly constrained* tangled tours and
induces a facet of their convex hull.

– in PHASE 3, we find a linear inequality that
separates $\overline{x}^*$ from all tangled tours and
induces a facet of their convex hull.

Weakly constrained tangled tours are defined as tangled tours that satisfy

$\sum(\overline{x}_e : u \in e) = 2$ for all u in $\{1, 2, \ldots, k\}$;

moderately constrained tangled tours are defined as weakly constrained tangled tours that satisfy

$\overline{x}_e = 0$ for all e such that $e \subset \{1, 2, \ldots, k\}$ and $\overline{x}^*_e = 0$,
$\overline{x}_e = 1$ for all e such that $e \subset \{1, 2, \ldots, k\}$ and $\overline{x}^*_e = 1$;

note that strongly constrained tangled tours are precisely the moderately constrained tangled tours that satisfy

$\overline{x}_{0u} = 0$ for all u in $\{1, 2, \ldots, k\}$ such that $\overline{x}^*_{0u} = 0$.

In PHASE 1 and PHASE 2, we use a function ORACLE that,

given integer vectors c, ℓ, u and a threshold t (an integer or $-\infty$),
returns either
a weakly constrained tangled tour $\overline{x}$ that maximizes $c^T\overline{x}$
subject to $\ell \le \overline{x} \le u$, $c^T\overline{x} > t$
or
the message "infeasible" indicating that
no weakly constrained tangled tour $\overline{x}$ satisfies
$\ell \le \overline{x} \le u$, $c^T\overline{x} > t$.

This is the same function that is used, with a fixed ℓ, a fixed u, and $t = -\infty$, to find items $\overline{x}$ for the collection in Algorithm 5.2.

5.3.1. PHASE 1. Moderately constrained tangled tours are like strongly constrained tangled tours in that every such tangled tour $\overline{x}$ is determined by its restriction $\psi(\overline{x})$ on $\overline{E}_{1/2}$; they are unlike strongly constrained tangled tours in that (unless $\overline{x}^*$ violates a readily available subtour inequality) the set

$$\{\psi(\overline{x}) : \overline{x} \text{ is a moderately constrained tangled tour}\}$$

includes the $1 + |\overline{E}_{1/2}|$ vertices of the unit simplex, whereas the set

$$\{\psi(\overline{x}) : \overline{x} \text{ is a strongly constrained tangled tour}\}$$

does not necessarily have full dimension (and may even be empty). This is why we choose moderately constrained tangled tours as an intermediate stop on the way from strongly constrained tangled tours to all tangled tours.

Algorithm 5.2 has produced a linear inequality $a^T\psi(x) \leq b$ which is satisfied by all strongly constrained tangled tours and violated by $\overline{x}^*$; since $b < a^T\psi(\overline{x}^*) \leq ||a||_1$, the inequality

$$a^T\psi(\overline{x}) - (||a||_1 - b)\sum(\overline{x}_{0u} : \overline{x}^*_{0u} = 0) \leq b$$

is satisfied by all moderately constrained tangled tours and violated by $\overline{x}^*$. In Phase 1, we convert this inequality into a linear inequality that induces a facet of the convex hull of the set of all moderately constrained tangled tours and is violated by $\overline{x}^*$. We start out with an integer vector a, an integer b, and a (possibly empty) set $\mathcal{I}$ such that

- all moderately constrained tangled tours $\overline{x}$ have $a^T\overline{x} \leq b$,
- $a^T\overline{x}^* > b$,
- $\mathcal{I}$ is an affinely independent set of moderately constrained tangled tours,
- $a^T\overline{x} = b$ whenever $\overline{x} \in \mathcal{I}$.

Algorithm 5.3 maintains these four invariants while adding new elements to $\mathcal{I}$ and adjusting a and b if necessary; when $|\mathcal{I}|$ reaches $|\overline{E}_{1/2}|$, the current cut $a^T\overline{x} \leq b$ induces a facet of the convex hull of all moderately constrained tangled tours.

Algorithm 5.3 From a cut to a facet-inducing cut:

while $|\mathcal{I}| < |\overline{E}_{1/2}|$
do find an integer vector v, an integer w, and
a moderately constrained tangled tour $\overline{x}^0$ such that
- $v^T\overline{x} = w$ whenever $\overline{x} \in \mathcal{I}$,
- some moderately constrained tangled tour $\overline{x}$ has $v^T\overline{x} \neq w$,

and
- either $v^T\overline{x}^* \geq w$ and $v^T\overline{x} \geq w$ for all moderately constrained tangled tours $\overline{x}$,
or else $a^T\overline{x}^0 < b$ and $v^T\overline{x}^0 = w$;

find an integer vector a', an integer b', and
a moderately constrained tangled tour $\overline{x}'$ such that
- all moderately constrained tangled tours $\overline{x}$ have $a'^T\overline{x} \leq b'$,
- equation $a'^T\overline{x} = b'$ is a linear combination of $a^T\overline{x} = b$ and $v^T\overline{x} = w$,
- $a'^T\overline{x}' = b'$ and $(a^T\overline{x}', v^T\overline{x}') \neq (b, w)$,
- $a'^T\overline{x}^* > b'$;

$a = a'$, $b = b'$, $\mathcal{I} = \mathcal{I} \cup \{\overline{x}'\}$;
end
return a and b;

In early iterations of the **while** loop in Algorithm 5.3, Concorde gets its v and w by scanning the list of inequalities

$$\begin{aligned} \overline{x}_e &\geq 0 \quad \text{such that } e \in \overline{E}_{1/2}, \\ -\overline{x}_e &\geq -1 \quad \text{such that } e \in \overline{E}_{1/2}, \\ \overline{x}_{0u} &\geq 0 \quad \text{such that } u \in \{1, 2, \ldots, k\} : \end{aligned}$$

if any of any of these inequalities $v^T\overline{x} \geq w$ happens to satisfy

$$v^T\overline{x} = w \text{ whenever } \overline{x} \in \mathcal{I},$$

then it provides the v and the w for use, with an arbitrary moderately constrained tangled tour $\overline{x}^0$, in the next iteration. If this source dries up and yet $|\mathcal{I}| < |\overline{E}_{1/2}|$, then Concorde finds v as a nonzero solution of the system

$$\begin{aligned} v^T\overline{x} &= 0 \text{ for all } \overline{x} \text{ in } \mathcal{I}, \\ v_e &= 0 \text{ for all } e \text{ outside } \overline{E}_{1/2}, \end{aligned}$$

it sets $w = 0$, and it lets $\overline{x}^0$ be the moderately constrained tangled tour such that $\overline{x}^0_e = 0$ for all e in $\overline{E}_{1/2}$.

To add new elements to $\mathcal{I}$ and to adjust a and b if necessary, Concorde's implementation of Algorithm 5.3 uses a function TILT, which, given integer vectors a, v, integers b, w, and a moderately constrained tangled tour $\overline{x}^0$ such that

- if all moderately constrained tangled tours $\overline{x}$ have $v^T\overline{x} \leq w$, then $a^T\overline{x}^0 < b$ and $v^T\overline{x}^0 = w$,

returns a nonzero integer vector a', an integer b', and a moderately constrained tangled tour $\overline{x}'$ such that

- all moderately constrained tangled tours $\overline{x}$ have $a'^T\overline{x} \leq b'$,
- inequality $a'^T\overline{x} \leq b'$ is a nonnegative linear combination of $a^T\overline{x} \leq b$ and $v^T\overline{x} \leq w$,
- $a'^T\overline{x}' = b'$ and $(a^T\overline{x}', v^T\overline{x}') \neq (b, w)$.

In the iterations of the **while** loop in Algorithm 5.3 where v and w are drawn from the list of inequalities, Concorde calls TILT $(a, b, v, w, \overline{x}^0)$ for $(a', b', \overline{x}')$; in the iterations where v is computed by solving a system of linear equations and $w = 0$, Concorde computes

$$\begin{aligned} (a^+, b^+, \overline{x}^+) &= \text{TILT}(a, b, v, 0, \overline{x}^0), \\ (a^-, b^-, \overline{x}^-) &= \text{TILT}(a, b, -v, 0, \overline{x}^0) \end{aligned}$$

and then it sets

$$\begin{aligned} a' = a^+,\ b' = b^+,\ \overline{x}' = \overline{x}^+ \quad &\text{if } a^{+T}\overline{x}^* - b^+ \geq a^{-T}\overline{x}^* - b^-, \\ a' = a^-,\ b' = b^-,\ \overline{x}' = \overline{x}^- \quad &\text{otherwise.} \end{aligned}$$

Algorithm 5.4 implements TILT by the *Dinkelbach method* of fractional programming (see, for instance, Sect. 5.6 of Craven (1988) or Sect. 4.5 of Stancu-Minasian (1997)).

Algorithm 5.4 TILT $(a, b, v, w, \overline{x}^0)$:

$\overline{x}$ = moderately constrained tangled tour that maximizes $v^T\overline{x}$;
$\lambda = v^T\overline{x} - w$, $\mu = b - a^T\overline{x}$;
if $\lambda = 0$
then return $(v, w, \overline{x}^0)$;
else **if** $\mu = 0$
then return $(a, b, \overline{x})$;
else return TILT $(a,\ b,\ \lambda a + \mu v,\ \lambda b + \mu w,\ \overline{x})$;
end
end

5.3.2. PHASE 2. Let us write

$$E_0 = \{e :\ e \subset \{1, 2, \ldots, k\}, \overline{x}^*_e = 0\},$$
$$E_1 = \{e :\ e \subset \{1, 2, \ldots, k\}, \overline{x}^*_e = 1\};$$

in this notation, a weakly constrained tangled tour $\overline{x}$ is moderately constrained if and only if

$$\overline{x}_e = 0 \text{ whenever } e \in E_0 \quad \text{and} \quad \overline{x}_e = 1 \text{ whenever } e \in E_1.$$

The linear inequality $a^T\overline{x} \le b$ constructed in PHASE 1 separates $\overline{x}^*$ from all moderately constrained tangled tours and induces a facet of their convex hull; in PHASE 2, we find integers Δ_e $(e \in \overline{E}_0 \cup \overline{E}_1)$ such that the inequality

$$a^T\overline{x} + \sum(\Delta_e\overline{x}_e : e \in \overline{E}_0 \cup \overline{E}_1) \le b + \sum(\Delta_e : e \in \overline{E}_1)$$

separates $\overline{x}^*$ from all weakly constrained tangled tours and induces a facet of their convex hull. A way of computing the Δ_e one by one originated in the work of Gomory (1969) and was elaborated by Balas (1975), Hammer, Johnson, and Peled (1975), Padberg (1973,1975), Wolsey (1975a, 1975b), and others; it is known as *sequential lifting*; its application in our context is described in Algorithm 5.5. Both **while** loops in this algorithm maintain the invariant

$a^T\overline{x} \le b$ induces a facet of the convex hull of
all weakly constrained tangled tours $\overline{x}$ such that
$\overline{x}_e = 0$ whenever $e \in F_0$ and $\overline{x}_e = 1$ whenever $e \in F_1$.

Concorde implements PHASE 2 as a streamlined version of Algorithm 5.5, where the problem of finding $\overline{x}^{\max}$ may include constraints $\overline{x}_e = 0$ with $e \notin F_0 \cup F_1$ and certain edges may be deleted from F_0 without finding $\overline{x}^{\max}$ and updating $a^T\overline{x} \le b$; for details, see Sect. 4.4 of Applegate et al. (2001).

Algorithm 5.5 Sequential lifting

$F_0 = \overline{E}_0$, $F_1 = \overline{E}_1$;
while $F_1 \neq \emptyset$
do $f =$ an edge in F_1;
find a weakly constrained tangled tour $\overline{x}^{\max}$ that
maximizes $a^T\overline{x}$ subject to
$\overline{x}_e = 0$ whenever $e \in F_0 \cup \{f\}$,
$\overline{x}_e = 1$ whenever $e \in F_1 - \{f\}$;
replace $a^T\overline{x} \leq b$ by $a^T\overline{x} + (a^T\overline{x}^{\max} - b)\overline{x}_f \leq a^T\overline{x}^{\max}$;
delete f from F_1;
end
while $F_0 \neq \emptyset$
do $f =$ an edge in F_0;
find a weakly constrained tangled tour $\overline{x}^{\max}$ that
maximizes $a^T\overline{x}$ subject to
$\overline{x}_e = 0$ whenever $e \in F_0 - \{f\}$,
$\overline{x}_f = 1$;
replace $a^T\overline{x} \leq b$ by $a^T\overline{x} + (b - a^T\overline{x}^{\max})\overline{x}_f \leq b$;
delete f from F_0;
end

5.3.3. PHASE 3. Let $\overline{E}$ denote the edge-set of the complete graph with vertex-set $\overline{V}$. Naddef and Rinaldi (1992) call inequalities

$$\sum(a_e x_e : e \in \overline{E}) \geq b \tag{18}$$

tight triangular if $a_e \geq 0$ for all e and

$$\min\{a_{uw} + a_{wv} - a_{uv} : u \neq v, u \neq w, v \neq w\} = 0 \text{ for all } w;$$

their arguments can be used to justify the following claims.

Theorem 1. *If* (18) *is satisfied by all weakly constrained tangled tours, if* $a_e \geq 0$ *for all* e*, and if* $a_{uw} + a_{wv} \geq a_{uv}$ *for all choices of distinct* u, v, w*, then* (18) *is satisfied by all tangled tours.*

Theorem 2. *If a linear inequality induces a facet of the convex hull of all weakly constrained tangled tours and if it is tight triangular, then it induces a facet of the graphical traveling salesman polytope.*

These theorems are the reason why we choose weakly constrained tangled tours as an intermediate stop on the way from moderately constrained tangled tours to all tangled tours.

The linear inequality $a^T\overline{x} \leq b$ constructed in PHASE 2 separates $\overline{x}^*$ from all weakly constrained tangled tours and induces a facet of their convex hull; since Concorde substitutes in PHASE 1

$$2 - \sum(\overline{x}_e : e \subset \{1, 2, \ldots, k\}, u \in e)$$

for each $\overline{x}_{0u}$, we have $a_{0u} = 0$ for all u; since $a^T\overline{x} \leq b$ induces a facet of the convex hull of all weakly constrained tangled tours, it follows that $a_e \geq 0$ for all e. It is a trivial matter to construct a hypergraph $\overline{\mathcal{H}}$ on $\overline{V} - \{0\}$ and positive integers $\lambda_Q (Q \in \overline{\mathcal{H}})$ such that the linear form

$$\sum(\lambda_Q \sum(\overline{x}_e : e \subseteq Q) : Q \in \overline{\mathcal{H}})$$

is identically equal to $a^T\overline{x}$ (Concorde does it by a greedy heuristic aiming to minimize $\sum \lambda_Q$); since every weakly constrained tangled tour $\overline{x}$ and every subset Q of $\overline{V} - \{0\}$ satisfy

$$2|Q| = 2\sum(\overline{x}_e : e \subseteq Q) + x(Q, \overline{V} - Q),$$

the inequality

$$\sum(\lambda_Q x(Q, \overline{V} - Q) : Q \in \overline{\mathcal{H}}) \geq \sum(2\lambda_Q |Q| : Q \in \overline{\mathcal{H}}) - 2b \qquad (19)$$

separates $\overline{x}^*$ from all weakly constrained tangled tours and induces a facet of their convex hull; Theorem 1 guarantees that (19) is satisfied by all tangled tours.

Theorem 2 points out an easy way of converting (19) into a linear inequality that induces a facet of the graphical traveling salesman polytope and is violated by $\overline{x}^*$:

- subtract $\sum_{w=0}^{k} \delta_w \overline{x}(\{w\}, \overline{V} - \{w\})$ from the left-hand side of (19) and
- subtract $\sum_{w=0}^{k} 2\delta_w$ from the right-hand side of (19)

with $\delta_0, \delta_1, \ldots, \delta_k$ chosen to make the resulting inequality tight triangular; specifically,

$$\delta_w = \min\{\tau(u, v, w) : u \neq v, u \neq w, v \neq w\},$$

where

$$\begin{aligned}\tau(u, v, w) = &\sum(\lambda_Q : Q \in \overline{\mathcal{H}}, u \in Q, v \in Q, w \notin Q) + \\ &\sum(\lambda_Q : Q \in \overline{\mathcal{H}}, u \notin Q, v \notin Q, w \in Q)\end{aligned}$$

for all choices of distinct points u, v, w of $\overline{V}$. (This procedure fails to work if and only if the left-hand side of the new, tight triangular, inequality turns out to be $0^T\overline{x}$; it can be shown that this will happen if and only if (19) is a positive multiple of the subtour inequality $\overline{x}(\{0\}, \overline{V} - \{0\}) \geq 2$, which induces a facet of the graphical traveling salesman polytope.)

Concorde's pricing mechanism (see Section 8) is incompatible with negative coefficients in hypergraph constraints; for this reason, it settles in its implementation of Algorithm 5.1 for adding to $\mathcal{L}$ the hypergraph inequality resulting when the variable $\overline{x}$ of (19) is changed to x. Still, even (19) is often tight triangular, in which case it induces a facet of the graphical traveling salesman polytope: the greedy heuristic used to construct $\overline{\mathcal{H}}$ and $\lambda_Q (Q \in \overline{\mathcal{H}})$ tends to minimize the number of vertices w such that $\delta_w > 0$.

5.4. Experimental findings on the 100,000-city Euclidean TSP

In Table 1, we report the lower bounds obtained by applying local cuts to our 100,000-city instance. Here we let $t_{\max}$ vary from 8 up to 22, increasing the parameter by 2 in each run. Note that individual runs are started from

Table 1. Local Cuts on 100,000-city TSP

$t_{\max}$	$\leq$ Gap to Optimal
8	0.138%
10	0.126%
12	0.119%
14	0.112%
16	0.107%
18	0.103%
20	0.100%
22	0.097%

scratch (we do not pass the LP and cut pool from one run to the next, as we do in the large-scale runs described in Section 9.3 and in Section 9.4).

6. The core LP

The LPs that need to be solved during a TSP computation contain far too many columns to be handled directly by an LP solver. It is therefore necessary to combine the cutting-plane method with the dual concept known as *column generation*. For background material on linear programming and column generation, we refer the reader to standard reference works by Chvátal (1983), Schrijver (1986), Nemhauser and Wolsey (1988), Wolsey (1998), and Vanderbei (2001).

The column generation technique is used to explicitly solve only a *core LP* containing columns corresponding to a small subset of the complete set of edges. The edges not in the core LP are handled by computing from time to time the reduced costs that the corresponding columns would give with respect to the current LP dual solution; if any columns have negative reduced cost, then some subset of them can be added to the core LP. We discuss several aspects of this procedure below (see also Section 8), together with techniques for keeping the size of the core LPs small by deleting cutting planes that no longer appear to be useful in solving the particular TSP instance.

6.1. Initial edge-set

The first systematic use of core edge-sets is in the work of Land (1979). Her test set of problem instances included a number of 100-city examples (so 4,950 edges), but she restricted the number of core edges to a maximum

of 500. The initial edge-set she chose consisted of the union of the edges in the best tour she found together with the 4-nearest neighbor edge-set (that is, the 4 minimum cost edges containing each city).

Land's edge-set was also used by Grötschel and Holland (1991); they ran tests with the initial set consisting of the union of the best available tour and the k-nearest neighbor edge-set, for $k \in \{0, 2, 5, 10\}$. Jünger et al. (1994) carried out further tests, using $k \in \{2, 5, 10, 20\}$. In this later study, the conclusion was that $k = 20$ produced far too many columns and slowed down their solution procedure, but the smaller values of k all led to reasonable results.

A different approach was adopted by Padberg and Rinaldi (1991), taking advantage of the form of their tour-finding heuristic. In their study, Padberg and Rinaldi compute a starting tour by making k independent runs of the heuristic algorithm of Lin and Kernighan (1973); for their initial edge-set, they take the union of the edge-sets of the k tours. Although we do not use repeated runs of Lin-Kernighan to obtain an initial tour, the Padberg-Rinaldi tour-union method remains an attractive idea since it provides an excellent sample of the edges of the complete graph (with respect to the TSP).

In Table 2, we compare the Padberg-Rinaldi idea with the k-nearest set on our 100,000-city instance. The tours were obtained with short runs of the Chained Lin-Kernighan heuristic of Martin et al. (1991), using the implementation described in Applegate et al. (2003); the short runs each used $|V|/100 + 1$ iterations of the heuristic. (For k-nearest, we also include the edges in the best available tour).

In each case, we ran the cutting-plane separation algorithms we described in Sections 2, 3, 4, and 5 (with the local cuts' $t_{\max}$ set to 8), together with column generation over the edges of the complete graph on the 100,000 points. For each of the edge-sets we report the initial number of

Table 2. Initial Edge-set for 100,000-city TSP

Edge-set	$\|E\|$-initial	$\|E\|$-final	CPU Time (seconds)
2-Nearest	146,481	231,484	24,682
3-Nearest	194,569	240,242	40,966
4-Nearest	246,716	270,096	26,877
5-Nearest	300,060	312,997	26,257
10 Tours	167,679	224,308	23,804
50 Tours	215,978	247,331	24,187

edges and the final number of edges in the core LP (after the cutting-plane and column-generation routines terminated), as well as the total running times on a EV67-based (667 MHz) Compaq AlphaServer ES40. We choose the union of 10 tours in our implementation—it has the lowest CPU time in the test and it maintains the smallest edge-set during the computations.

6.2. Adding and deleting edges

The results in Table 2 indicate the growth of the cardinality of the core edge-set as the combined cutting-plane and column generation algorithm progresses. To help limit this growth, we are selective about the edges that are permitted to enter the core and we also take steps to remove edges from the core if they do not appear to be contributing to the LP solution.

When an edge is found to have negative reduced cost in our pricing routine, it is not directly included in the core edge-set, but rather it is added to a queue of edges that are waiting for possible inclusion in the core. The Concorde code will at irregular intervals (determined by the increase of the optimal value of the LP relaxation) remove the N edges from the queue having the most negative reduced cost (we use $N = 100$), and add these to the core LP. An LP solver is then used to obtain new optimal primal and dual solutions, and the reduced costs for the remaining edges in the queue are recomputed; any edge in the queue that has reduced cost greater than some small negative tolerance (we use negative 0.00001) is removed from the queue.

After an edge e is added to the LP, we monitor the corresponding component of x^* at each point when the LP solver produces a new solution. If for L_1 consecutive LP solves the value of x^*_e is less than some small tolerance ε_1, then we remove edge e from the core LP. (We set $L_1 = 200$ and $\varepsilon_1 = 0.0001$.)

6.3. Adding and deleting cuts

Like in the case for edges, when a cutting plane is found by a separation routine, it is attached to the end of a queue of cuts that are waiting to be added to the core LP. The Concorde code repeatedly takes the first cut from the queue for processing and checks that it is violated by the current x^* by at least some small tolerance (we use 0.002). If the cut does not satisfy this condition, then it is discarded; otherwise it is added to the core LP. (Note that the current x^* may perhaps not be the same as the vector that was used in the separation algorithm that produced the cut.)

After k cuts have been added to the LP (we use $k = 2000$ in our code for large instances) or after the cut queue becomes empty, an LP solver is called to compute optimal primal and dual solutions for the new core LP. If the slack variable corresponding to a newly added cut is in the optimal LP basis, then the cut is immediately deleted from the LP. Otherwise, the cut remains in the LP and a copy of the cut is added to a pool of cuts that is maintained for possible later use by the separation routines. (Subtour inequalities are not added to the pool, since an efficient exact separation routine for them is available.).

Once a cut is added, after each further point where the LP solver produces new solutions, we check the dual variable corresponding to the cut.

If for L_2 consecutive LP solves the dual variable is less than some fixed tolerance ε_2, then the cut is deleted from the LP. (We use $L_2 = 10$ and $\varepsilon_2 = 0.001$.) This deletion condition is weaker than the standard practice of deleting cuts only if they are slack (that is, only if they are not satisfied as an equation by the current x^*); for our separation routines on large TSP instances, we found that the core LP would accumulate a great number of cuts that were satisfied as an equation by the current x^* if we only removed slack inequalities.

It may well happen in our computation that a cut is deleted from the LP, but then added back again after a number of further LP solves. Although this is obviously a waste of computing time, when working on large instances it seems necessary to be very aggressive in attempting to keep the core LP small (both for memory considerations and to help the LP solver—the results in Section 9 indicate that the solution of the LP problems is the dominant part of a TSP computation on large instances).

(Notice that the tolerances and constants we use for cuts and edges differ by large amounts. These values were obtained through computational experiments, and they are dependent on the method used to solve the core LP problems.)

7. Cut storage

The storage of cutting planes and their representation in an LP solver account for a great portion of the memory required to implement the Dantzig et al. algorithm. In this section we discuss the methods used by Concorde to attempt to reduce this demand for memory when solving large problem instances.

There are three places in the computer code where we need to represent cuts: in the LP solver, in the external LP machinery, and in the pool of cuts. We discuss below representations for each of these components.

7.1. Cuts in the LP solver

The cutting planes we use in our implementation can all be written as

$$\mathcal{H} \circ x \geq \mu(\mathcal{H}). \tag{20}$$

This is how we like to think about the cuts, but we need not be restricted to this form when carrying out a computation. Indeed, the degree equations (4) give us a means to dramatically alter the appearance of a cut. For example, we can write (20) as

$$\sum_{S \in \mathcal{F}} x(\{e : e \subseteq S\}) \leq I_{\mathcal{H}} \tag{21}$$

for some constant $I_{\mathcal{H}}$. The representation (21) can further be altered by replacing some sets S by their complements $V - S$. Moreover, starting with any form, we can add or subtract multiples of degree equations to further manipulate a cut's appearance. We make use of this freedom in selecting the representation of the cut in the LP solver.

The most important criterion for choosing the LP representation of a cut is the number of nonzero entries the representation adds to the constraint matrix. The amount of memory required by the solver is proportional to the total number of nonzeros, and, other things being equal, LP solvers are more efficient in solving LPs with fewer nonzeros.

We compare four different representations: (1) each cut in the form given in (20) (the "outside" form), (2) each cut in the form given in (21) (the "inside" form), (3) each cut in the inside form with individual sets S complemented if it decreases the number of nonzero coefficients among the columns in the core LP, and (4) individual cuts in the form (either outside, or inside with complemented sets) that gives the least number of nonzeros.

For each of these representations we consider three different algorithms for selecting multiples of the degree equations to add or subtract from the individual cuts to reduce the number of nonzeros among the columns in the core LP.

In the first method, for each cut we simply run through the cities in order, and subtract the corresponding degree equation from the cut if it is advantageous to do so (that is, if it will decrease the number of nonzeros).

The second method is a greedy algorithm that, for each cut, first makes a pass through all of the cities and counts the number of nonzeros that can be saved by subtracting the corresponding degree equation. It then orders the cities by nonincreasing values of this count and proceeds as in the first method.

The third method is similar to greedy, but adjusts the counts to reflect the current status of the cut. In this algorithm, we keep two priority queues, one containing cities whose corresponding degree equation can be advantageously subtracted from the cut and another containing cities whose degree equation can be advantageously added to the cut. Both priority queues are keyed by the number of nonzeros that the operations save, that is, cities that save more nonzeros have priority over cities that save fewer nonzeros. At each step of the algorithm, we select the cuts having the maximum key and either subtract or add the corresponding equation. We then update the keys appropriately, possibly inserting new cities into the queues. The algorithm terminates when both queues are empty. (Notice that this algorithm permits equations to be added or subtracted a multiple number of times.)

We tested these algorithms and representations on core LPs taken from our computations on 21 small problems from the TSPLIB collection maintained by Reinelt (1991); the 21 instances range in size from 1,000 cities to 7,397 cities. The results are reported in Table 3, in multiples of the number of nonzeros in the outside representation.

Table 3. Nonzeros in LP Representations

Algorithm	Outside	Inside	Complemented	Best
None	1.00	2.81	2.07	0.90
Straight	0.54	1.40	1.11	0.49
Greedy	0.52	1.19	0.95	0.48
Queues	0.49	1.07	0.87	0.45

The immediate conclusions are that the outside form of cuts clearly dominates the inside form (even with complementation) and that the reduction routines appear to be worthwhile. Although the "best" form of cuts is slightly preferable to the outside form, we use the outside form in our implementation since this simplification leads to a more efficient routine for computing the reduced costs of columns that are not in the core LP.

For our reduction algorithm, we use the queue-based routine: it is efficient and is slightly better than the other two methods.

The representations chosen in earlier computational studies vary greatly from research team to research team. Dantzig et al. (1954), Hong (1972), Clochard and Naddef (1993), Jünger et al. (1994), and Naddef and Thienel (2002b) appear to have used the outside form of subtour inequalities, whereas Miliotis (1978), Land (1979), Padberg and Hong (1980), Crowder and Padberg (1980), Grötschel and Holland (1991), and Padberg and Rinaldi (1991) all appear to have used the inside form. Clochard and Naddef (1993) and Naddef and Thienel (2002b) used the outside form of combs and more general inequalities, and Land (1979) used a special outside form for blossoms, but each of the other studies used the inside form for all cuts other than subtours. The complementation of subtours was probably carried out in most of the studies that used the inside form of the inequalities, but Padberg and Rinaldi (1991) is the only paper that mentions complementing other inequalities—they consider complementing the handles in combs.

7.2. External storage of cuts

It is not sufficient to use the LP solver's internal list of the current cutting planes as our only representation of the cuts. The trouble is that this internal representation does not support the computation of the reduced costs of the edges not present in the core LP. What is needed is a scheme for storing the cuts in their implicit form, that is, as hypergraphs $\mathcal{H} = (V, \mathcal{F})$, where $\mathcal{F}$ is a family of subsets of V. The most important criteria for evaluating such an external representation scheme are the total amount of storage space required and the ease with which the data structure supports a fast edge pricing mechanism.

In our implementation, we choose a very compact representation of the cuts, one that fits in well with the pricing routine that we describe in Section 8. Before we discuss the representation, we present some alternative schemes that have appeared in earlier studies.

7.2.1. Previous Work The external representation of cuts is first treated in Land (1979). Land's technique for storing subtour inequalities is to pack a collection of pairwise disjoint subsets of V into a single vector of length $|V|$, where the subsets correspond to the vertex sets of the subtours. The entries of the vector provide a labeling of the vertices such that each of the subsets is assigned a distinct common label. This representation was particularly useful for Land since her separation routines (connectivity cuts and a version of subtour shrinking) naturally produced collections that were pairwise disjoint.

Land used this same technique for storing the handles of blossom inequalities. She got around the problem of storing the teeth of the blossoms by requiring that these edges be part of the core LP. This meant that the routines for pricing out edges outside the core were never needed to compute the reduced cost of a tooth edge.

Grötschel and Holland (1991) also used column generation, but they did not report any details of the external representation scheme used in their study, writing only: "After some experiments we decided to trade space for time and to implement space-consuming data structures that allow us to execute pricing in reasonable time."

A similar philosophy of trading off space for time was adopted by Padberg and Rinaldi (1991). They used a full $|V|$-length vector to represent each of the cuts. The representation, however, allowed them to compute the reduced cost of an edge by making a very simple linear scan through the cuts, performing only a constant amount of work for each cut.

A more compact representation was used by Jünger et al. (1994). They store a cut for the hypergraph $\mathcal{H} = (V, \mathcal{F})$ by keeping the sets in $\mathcal{F}$ as an array that lists the vertices in each set, one after another. The lists are preceded by the number of sets, and each set in the list is preceded by the number of vertices it contains. In the typical case, the length of the array will be much less than $|V|$, but the extraction of the cut coefficient on an individual edge is more costly than in Padberg and Rinaldi's approach.

Our scheme is similar to Jünger et al., but uses a different representation for the individual sets in the cut. It suffers the same drawback in extracting individual edge coefficients, but the pricing mechanism we describe in Section 8 does not make use of single coefficient extraction, using instead a strategy that calls for the simultaneous pricing of a large group of edges. This simultaneous pricing allows us to spend a small amount of CPU time, up front, setting up an auxiliary data structure that will speed the coefficient generation for the edges in the pricing group. This strategy allows us to take advantage of the compact cut representation without incurring a significant penalty in the performance of our pricing engine.

7.2.2. Hypergraphs The majority of the space used in Jünger et al.'s cut representation is occupied by the lists of the vertices in individual sets. To improve on this, we must either find a way to write these lists more compactly or find a way to avoid writing them altogether. We postpone a

discussion of list compression, and first consider a method for reducing the number of lists.

The idea is simple: we reuse old representations. To carry this out, if we have a collection of cuts $\mathcal{H}_1 = (V, \mathcal{F}_1), \mathcal{H}_2 = (V, \mathcal{F}_2), \ldots, \mathcal{H}_r = (V, \mathcal{F}_r)$, we represent separately the sets $\mathcal{S} = \mathcal{F}_1 \cup \mathcal{F}_2 \cup \cdots \cup \mathcal{F}_r$ and the hypergraphs $\mathcal{H}_1, \mathcal{H}_2, \ldots, \mathcal{H}_r$. In this setup, the hypergraphs are lists of pointers to the appropriate sets, rather than lists of the sets themselves. The benefit of this split representation is that the number of sets in $\mathcal{S}$ is typically considerably less than the sum of the cardinalities of the individual $\mathcal{F}_i$'s. (This split representation also helps our are pricing routine, as we describe in Section 8.)

When a cut $\mathcal{H} = (V, \mathcal{F})$ is added to the core LP, we add to $\mathcal{S}$ the sets from $\mathcal{F}$ that do not already appear in $\mathcal{S}$, and we build a list of pointers for $\mathcal{H}$ into $\mathcal{S}$. To delete a cut $\mathcal{H} = (V, \mathcal{F})$, we delete $\mathcal{H}$'s list of pointers as well as all sets in $\mathcal{S}$ that were only being referenced by $\mathcal{H}$. To do this efficiently, we keep track of the number of times each set in $\mathcal{S}$ is being referenced (increasing the numbers when we add a cut, and decreasing them we delete a cut), and remove any set whose reference number reaches 0. The elements of $\mathcal{S}$ are stored as entries in an array, using a hash table to check whether a prescribed set is currently in $\mathcal{S}$.

7.2.3. Tour intervals In Subsection 2.4 we discussed that fact that as our core LP matures, the LP solution vector x^* approximates the incidence vector $\bar{x}$ of an optimal tour, and hence $x^*(S, V-S) \approx \bar{x}(S, V-S)$ for most subsets S of V. For this reason, the sets that appear in cutting planes can be expected to have a small $\bar{x}(S, V-S)$ value. An interpretation of this is that sets from our cutting planes can be expected to consist of a small number of intervals in an optimal tour.

We put this to the test, using a pool for the 7,397-city TSPLIB instance pla7397. The pool consists of 2,868,447 cuts and 1,824,418 distinct sets in $\mathcal{S}$. (The large pool of cuts was accumulated over a branch-and-cut run; working with this small TSP instance allowed us to build the pool in a reasonable amount of computing time.) The average number of vertices in the members of $\mathcal{S}$ is 205.4, but the sets can be represented using an average of only 5.4 intervals from a specific optimal tour—a savings of a factor of 38. This is typical of the compression ratios we obtained in other tests, so we adopt the interval list approach as our representation scheme, using the best available initial tour as a substitute for an optimal tour.

At the start of our TSP computation, we reorder the cities in the problem so that the best tour we have found up to that point is simply $0, 1, 2, \ldots, |V| - 1$. A set is then recorded as an array of pairs of integers giving the starting and ending indices of the intervals that make up the set, with a separate field giving the total number of intervals stored in the array.

7.3. Pool of cuts

Like the external representation for the LP, the pool needs to store cuts in the implicit (hypergraph) form. In this case, the important criteria for evaluating a representation are the amount of storage space required and the speed with which we can compute, for a prescribed x^*, the slacks of the individual cuts in the pool, that is, $\mu(\mathcal{H}) - \mathcal{H} \circ x^*$. Our approach is to use the methods we adopted in our external LP representation, but take advantage of the distribution of the references into $\mathcal{S}$ to further reduce the storage requirements.

Consider again the pla7397 pool we mentioned above. The 2,868,447 cuts make a total of 33,814,752 references to sets, and the sets make a total of 696,150,108 references to cities. Representing each of the cuts as an $|V|$-length integer vector would require approximately 78 Gigabytes of memory on machines that uses 4 bytes to represent an integer. The representation of Jünger et al. (1994) lowers the storage requirement considerably, calling for a little over 1.3 Gigabytes of memory. To achieve this, we use only 2 bytes for each reference to a city, taking advantage of the fact that the instance has less than 2^{16} cities in total. (Recall that a byte can represent 8 binary digits.) The split interval representation we described above would use 4 bytes for each interval, to specify its ends; 2 additional bytes for each set in $\mathcal{S}$, to specify the number of intervals used by the set; 3 bytes for each reference to a set in $\mathcal{S}$, using the fact that we have less than 2^{24} sets; and 2 additional bytes for each cut, to specify the number of sets. The collection $\mathcal{S}$ has a total of 1,824,418 sets that use a total of 9,877,792 intervals, so the pool can be stored in approximately 143 Megabytes of memory with this representation.

7.3.1. Removing Edge Teeth The savings in storage for the split interval representation is roughly a factor of 9 over the set-list representation for the pla7397 pool. This magnitude of savings is a big step towards making the direct use of large pools possible, but it is also somewhat disappointing, given the factor of 38 compression we obtained in the representation of the sets in $\mathcal{S}$. The poor showing can be explained by examining the data in Table 4, giving the number of sets of each size used by the cuts in the pool. Nearly two-thirds of the sets have cardinality just two, and over four-fifths of the sets of cardinality five or less. This means that, for the majority of the set references, we are not winning very much over the set-list representation.

An aggressive way to turn this lopsided distribution of set sizes to our advantage is to simply remove all of the references to sets of cardinality two, using a teething-like algorithm to reconstruct the sets on-the-fly. We did not pursue this idea, however, due to the considerable time this would add to the algorithm for searching for violated cuts among the entries in the pool.

Table 4. Sets in the pla7397 Pool of Cuts

Size	Number	Percentage
2	21342827	63.1%
3	3255535	9.6%
4	2267058	6.7%
5	736881	2.2%
6	651018	1.9%
7	261641	0.8%
8	337799	1.0%
9	140427	0.4%
≥ 10	4821566	14.3%

7.3.2. Set References Another route for decreasing the space requirements of our pool representation is to be more efficient in specifying our references to sets in $\mathcal{S}$. The straightforward method of using an integer (or 3-byte) value for each reference is clearly wasteful if one considers the distribution of the references among the cuts: there are 1,362,872 sets in $\mathcal{S}$ that are referenced only once, whereas the top ten most used sets are each referenced more than 100,000 times. An efficient representation should use less space to reference the sets that appear most often in the cuts. One way to do this is to use only a single byte to reference the most frequently used sets, use two bytes for next most frequently used sets, and restrict the use of three or four bytes to the sets that have very few references. There are a number of techniques for implementing this strategy. For example, the first byte in a set reference could either specify a set on its own, or indicate that a second, third, or forth byte is needed. This is the strategy we adopt. If the first byte of our set reference contains a number, K, between 0 and 127, then the set index is simply K. If K is between 128 and 191, then the difference $K - 128$ together with a second byte are used to represent the index. If K is between 192 and 223 then $K - 192$ plus two additional bytes are used, and if $K > 224$ then $K - 224$ plus three additional bytes are used. (In the pla7397 pool, no reference requires 4 bytes, but we want to have a representation that would also work for much larger pools.)

Using 3 bytes for each of the 33,814,752 sets contributes 97 Megabytes to the storage requirements for the split interval representation of the pla7397 pool. The compressed form of the references brings this down to 62 Megabytes.

This technique for expressing the set references using variable-length byte strings is a very rudimentary data compression technique. Better results can be obtained using an encoding developed by Huffman (1952) (see Cormen et al. (1990) or Knuth (1968)). Working with a byte-level Huffman code, this would save about 1 Megabyte from the 62 Megabyte total. If we are willing to work at the bit level, then the set reference storage can be reduced to 54 Megabytes for the pool. It should be noted, however, that working with binary encodings would result in some computational overhead to address the individual bits in the reference strings.

7.3.3. Interval References The list of intervals accounts for approximately 38 Megabytes of the storage requirement for the pla7397 pool. Since there are only 245,866 distinct intervals among the lists, adding another level of indirection, together with a method for compressing the interval references, could result in savings similar to what we achieved for the set references. Rather than doing this, however, we use a direct method to reduce the 4-byte per interval requirement that the straightforward technique of writing the ends of the intervals provides. The idea is to represent the interval from i to j as the number i together with the offset $j - i$, using 2 bytes to write i and either 0, 1 or 2 bytes to write j. This reduces the storage needed for the intervals down to 28 Megabytes.

This offset technique can be pushed a little further, writing the entire list of intervals for a prescribed set as a sequence of 1 or 2 byte offsets (splitting any interval that contains city 0 into two smaller intervals, to avoid the issue of intervals wrapping around from city $|V| - 1$ to city 0). This is the method we use in our code, and it lowers the storage requirement for the interval lists down to approximately 23 Megabytes, for the pla7397 pool.

7.3.4. Summary The memory requirements for the representations of the pla7397 pool are summarized in Table 5. The entries are progressive in the

Table 5. Memory Requirements for the pla7397 Pool of Cuts

Representation	Size (Megabytes)
$\|V\|$-length vector	80940
List of sets	1403
Split lists of cities	826
Split intervals	143
Compressed set references	109
Start plus offset intervals	99
List of interval offsets	96

sense that "Compressed set references" works with the split-interval representation, "Start plus offset intervals" uses the compressed set references, and so on.

In our computer code, we use the split interval representation of the pool, with compressed set references and lists of interval offsets (the "List of interval offsets" entry from Table 5).

8. Edge pricing

Column generation for the TSP is a simple operation for small instances—we simply run through all of the edges not currently in the core LP, compute their reduced costs, and add any negative edges into the core. This method breaks down on larger instances, however, due both to the time required

to price the entire edge-set and to the large number of negative reduced-cost edges that the pricing scans will detect. In this section we describe the techniques we adopt to handle these two difficulties.

8.1. Previous Work

We begin with a discussion of some earlier TSP column generation systems, starting with the paper of Land (1979).

In Land's study, the problem instances are described by specifying the geometric coordinates of the cities. Thus, she did not explicitly store the complete set of edges, relying instead on the ability to compute the cost between any prescribed pair of cities in order to carry out a pricing scan. This implicit treatment of the edge-set raised two problems. First, since her external LP representation did not explicitly list the teeth in blossom inequalities, the reduced costs for these edges could not be computed from the representation. Secondly, edges in the core LP that were not in the optimal basis but were set to the upper bound of 1, could appear to have negative reduced costs in the pricing scan. To deal with these "spurious infeasibilities", she maintained a data structure (consisting of an ordered tree and an extra list for edges that did not fit into the tree) to record the teeth edges and the non-basic 1-edges, and skipped over these pairs of cities during the pricing scan. Every negative edge that was detected in a scan was added to the core until a limit of 500 core edges was reached. After that, any further negative edges were swapped with core edges that were both non-basic and set to 0.

The pricing scans in Land's code were carried out at fixed points in the solution process, where the code moved from one phase to another. These transition points were determined by the completion of subtour generation, the completion of blossom generation, the detection of an integral LP solution, and the production of an LP optimal value that was above the cost of the best computed tour.

The issue of spurious edges was avoided by Grötschel and Holland (1991), using an original approach to edge generation. Before starting the LP portion of their TSP code, they passed the entire edge-set through a preprocessor that eliminated the majority of the edges from further consideration. Working with this reduced set, they maintained the core edges and the non-core edges as disjoint lists and restricted the pricing scans to the non-core set. Their scans were carried out after the cutting-plane phase of the computation ended without finding a new cut. If additional edges were brought into the core LP, their code reentered the cutting-plane phase and continued to alternate between cutting and pricing until all non-core edges had nonnegative reduced cost.

The preprocessor used by Grötschel and Holland takes the output of a run of the Held and Karp (1971) lower-bound procedure, computes the implicit reduced costs, and uses the resulting values to indicate edges that

cannot be contained in any tour that is cheaper than the best tour computed up to that point. These edges could thus be discarded from further consideration. For the instances they studied, this process resulted in sufficiently many deletions to permit them to implement their explicit edge list approach without running into memory difficulties.

Padberg and Rinaldi [1991] did not use preprocessing, but obtained a similar effect by implementing an approach that permitted them to quickly eliminate a large portion of the edge-set based on the LP reduced costs. At the start of their computation, they stored a complete list of the edge costs, ordered in such a way that they could use a formula (involving a square root computation) to obtain the two endpoints of an edge with a prescribed index. Their pricing scans were carried out over this list, and possible spurious infeasibilities were handled by checking that candidate edges having negative reduced cost were not among the current nonbasic core edges that have been set to their upper bound of 1. Each of the remaining negative reduced cost edges were added to the core set after the scan. Taking advantage of this simple rule, Padberg and Rinaldi stopped the computation of the reduced cost of an edge once it had been established that the value would indeed be negative.

Edge elimination comes into the Padberg and Rinaldi approach once a good LP lower bound for the problem instance has been obtained. Their process works its way through the complete set of edges, adding to a "reservoir" any edge that could not be eliminated based on its reduced cost and the value of the best computed tour. If the number of edges in the reservoir reaches a prescribed maximum, then the elimination process is broken off and the index k of the last checked edge in the complete list is recorded. ¿From this point on, pricing scans can be carried out by working first through the edges in the reservoir and then through the edges in the complete list starting at index k. Further elimination rounds attempt to remove edges from the reservoir and increase the number of preprocessed edges by working from k and adding any non-eliminated edges to the free places in the reservoir.

Padberg and Rinaldi also introduced the idea of permitting the status of the LP to determine when a pricing scan should be carried out, rather than using fixed points in the solution process. Their stated purpose is to try to keep the growth in the size of the core edge-set under control. To achieve this, they wanted to avoid the bad pricing scans that can arise after a long sequence of cutting-plane additions. Their strategy was simple: carry out a pricing scan after every five LP computations.

Jünger et al. (1994) adopt the Padberg-Rinaldi reservoir idea, but they also keep a precomputed "reserve" set of edges and do not price over the entire edge-set until each edge in the reserve set has nonnegative reduced cost. When their initial set of core edges is built from the k-nearest neighbors, the reserve set consists of the $(k+5)$-nearest graph. The default value for k is 5, but Jünger et al. also carried out tests using other values.

Following the Padberg and Rinaldi approach, Jünger et al. carry out a price scan after every five LP computations. They remark that, with this setup, the time spent on pricing for the instances in their test set (which included TSPs with up to 783 cities) was between 1% and 2% of the total running time.

8.2. Underestimating the Reduced Costs

A pricing mechanism for larger problem instances must deal with two issues that were not treated in the earlier studies. First, in each of the above approaches, the entire edge-set is scanned, edge by edge, at least once, and possibly several times. This would be extremely time consuming for instances having 10^6 or more cities. Secondly, although the Padberg and Rinaldi approach of carrying out a pricing scan after every five LP computations is aimed at keeping the size of the core set of edges under control, early on in the computation of larger instances, far too many of the non-core edges will have negative reduced costs (if we begin with a modestly sized initial core edge-set) to be able to simply add all of these edges into the core LP. This latter problem is a subtle issue in column generation and we have no good remedy, using only a simple heuristic for selecting the edges (see Section 6.2 and Section 8.4). The first problem, on the other hand, can be dealt with quite effectively using an estimation procedure that allows us to skip over large numbers of edges during a pricing scan, as we now describe.

Suppose we would like to carry out a scan with a core LP specified by the cuts $\mathcal{H}_1 = (V, \mathcal{F}_1), \mathcal{H}_2 = (V, \mathcal{F}_2), \ldots, \mathcal{H}_r = (V, \mathcal{F}_r)$. The dual solution provided by the LP solver consists of a value y_v for each city v and a nonnegative value Y_j for each cut $\mathcal{H}_j$. Let $\mathcal{S} = \mathcal{F}_1 \cup \cdots \cup \mathcal{F}_r$ be the collection of member sets of the hypergraphs and, for each S in $\mathcal{S}$, let $\pi_j(S)$ denote the number of times S is included in $\mathcal{F}_j$, for $j = 1, \ldots, r$. (Recall that the members of $\mathcal{F}_j$ need not be distinct.) For each S in $\mathcal{S}$, let

$$Y_S = \sum(\pi_j(S)Y_j : j = 1, \ldots, r).$$

The reduced cost of an edge $e = \{u, v\}$, having cost c_e, is given by the formula

$$\alpha_e = c_e - y_u - y_v - \sum(Y_S : e \cap S \neq \emptyset, e - S \neq \emptyset).$$

The computational expense in evaluating this expression comes from both the number of terms in the summation and the calculations needed to determine the sets S that make up this sum. A quick estimate can be obtained by noting that each of these sets S must contain either u or v. To use this we can compute

$$\bar{y}_v = y_v + \sum (Y_S : v \in S \text{ and } S \in \mathcal{S})$$

for each city v and let

$$\bar{\alpha}_e = c_e - \bar{y}_u - \bar{y}_v.$$

For most edges, $\bar{\alpha}_e$ will be a good approximation to α_e. Moreover, since $\bar{\alpha}_e$ is never greater than α_e, we only need to compute α_e if the estimate $\bar{\alpha}_e$ is negative.

The simplicity of the $\bar{\alpha}_e$ computation makes it possible to work through the entire set of edges of a TSP instance in a reasonable amount of time, even in the case of the 500 billion edges that make up an instance having 10^6 cities. It still requires us to examine each edge individually, however, and this would not work for instances much larger than 10^6 cities. Moreover, the time needed to explicitly pass through the entire edge-set would restrict our ability to carry out multiple price scans. What we need is a scheme that is able to skip over edges without computing the $\bar{\alpha}_e$'s. Although we cannot do this in general, for geometric problem instances we can accomplish this by taking advantage of the form of the edge-cost function when carrying out a pricing scan. The approach we take is similar to that used by Applegate and Cook (1993) to solve matching problems.

Suppose we have coordinates (v_x, v_y) for each city v and that the cost $c_{\{u,v\}}$ of an edge $\{u, v\}$ satisfies

$$c_{\{u,v\}} \geq t|u_x - v_x| \quad \text{for some positive constant } t, \text{ independent of } u \text{ and } v. \tag{22}$$

For these instances, we have

$$\alpha_e \geq \bar{\alpha}_e \geq t|u_x - v_x| - \bar{y}_u - \bar{y}_v.$$

So we only need to consider pricing those edges $\{u, v\}$ such that

$$t|u_x - v_x| - \bar{y}_u - \bar{y}_v < 0. \tag{23}$$

This second level of approximation allows us to take advantage of the geometry. (Condition (23) holds for each "EDGE_WEIGHT_TYPE" in the TSPLIB, other than the EXPLICIT and SPECIAL categories.)

At the start of a pricing scan, we compute $tv_x - \bar{y}_v$ for each city v and build a linked list of the cities in nondecreasing order of these values. We then build a permutation of the cities to order them by nondecreasing value of v_x. With these two lists, we begin processing the cities in the permuted order. While processing city v, we implicitly consider all edges $\{u, v\}$ for cities u that have not yet been processed. The first step is to delete v from the linked list. Then, since $u_x \geq v_x$ for each of the remaining cities u, we can write the inequality (23) as

$$tu_x - \bar{y}_u < tv_x - \bar{y}_v. \tag{24}$$

We therefore start at the head of the linked list, and consider the cities u in the linked list order until (24) is no longer satisfied, skipping over all of the

cities further down in the order. For each of the u's that we consider, we first compute $\bar{\alpha}_{\{u,v\}}$ and then compute $\alpha_{\{u,v\}}$ only if this value is negative.

This cut-off procedure is quite effective in limiting the number of edges that need to be explicitly considered. For the cost functions that are supported by *kd*-trees (see Bentley (1992)), however, it would be possible to squeeze even more performance out of the pricing routine by treating the $\bar{y}$ values as an extra coordinate in a *kd*-tree (as David S. Johnson (personal communication) proposed in the context of the Held-Karp lower bound procedure) and replacing the traversal of the linked list by a nearest-neighbor search.

8.3. Batch Pricing

Coming out of the approximate pricing, we have edges for which we must explicitly compute α_e. As we discussed in Section 7.2, extracting these reduced costs from the external LP representation is considerably more time consuming than with the memory-intensive representation used by Padberg and Rinaldi (1991). It is therefore important, in our case, to make a careful implementation of the pricing scheme.

Jünger et al. (1994) were faced with a similar problem. Their method is to build a pricing structure before the start of a pricing scan, and use this to speed up their computations. The structure consists of lists, for each city v, of the hypergraphs that contain v in one of their member sets. Working with the inside form of cuts, they compute the reduced cost of an edge $\{v, w\}$ by finding the intersection of the two hypergraph lists and working through the sets to extract the appropriate coefficients.

We also build a pricing structure, but the one we use is oriented around edges, rather than cities. To make this work, we price large groups of edges simultaneously, rather than edge by edge. This fits in naturally with the pricing scan mechanism that we have set up, since we can just hold off on computing the necessary α_e's until we have accumulated some prescribed number of edges (say 20,000).

In our setup, to compute the reduced costs of a prescribed set, U, of edges, we can begin with the $\bar{\alpha}_e$ values that have already computed. To convert $\bar{\alpha}_e$ to α_e, we need to add $2 * Y_S$ for each set S in $\mathcal{S}$ that contains both ends of e. The structure we build to carry this out consists of an incidence list for each city v, containing the indices of the edges of U that are incident with v. For each set S in $\mathcal{S}$ having $Y_S > 0$, we run though each city v in S and carry out the following operation. We consider, in turn, each edge in v's incidence list and check whether the other end of the edge is marked. If it is indeed marked, then we add $2 * Y_S$ to the edge's $\bar{\alpha}_e$ value. Otherwise, we simply move on to the next edge. After all of the edges have been considered, we mark the city v and move on to the next city in $\mathcal{S}$. The marking mechanism can be handled by initially setting a mark field to 0

for all cities, and, after each set is processed, incrementing the value of the integer label that we consider to be a "mark" in the next round.

After all sets have been processed, the $\bar{\alpha}_e$ values have been converted to α_e for each $e \in U$. We then attach each of the edges having negative α_e to a queue that holds the edges that are candidates for entering the core LP (see Subsection 6.2). Finally, the incidence structure is freed, and we wait for the next set of edges that need to be priced.

8.4. Cycling through the edges

Our pricing strategy does not require us to implicitly consider the entire edge-set during each pricing scan, but only that a good sampling of the potentially bad edges be scanned. We therefore use the pricing algorithm in two modes, as in Jünger et al. (1994). In the first mode, we only consider the k-nearest edges for some integer k (say $k = 50$), and in the second mode we treat the full edge-set. In both cases, the routine works its way through the edges, city by city, accumulating the set U of edges having $\bar{\alpha}_e < 0$ that will be priced exactly. Each time it is called, the search picks up from the last city that was previously considered, wrapping around to city 0 whenever the end of the city list is reached.

The approximate pricing algorithm is reset whenever the y_v's and Y_j's are updated. A reset involves the computation of the new Y_S's and $\bar{y}_i$'s, as well as the creation of a new linked list order. If successive calls to the algorithm, without a reset, allow us to process all of the edges, then we report this to the calling routine and carry out no further approximate pricing until the next reset occurs; this provides information that can be used to terminate a pricing scan.

8.5. Permitting negative edges

During our column generation procedure we take advantage of the fact that if z^* is the objective value of the LP solution and p is the sum of the reduced costs of all edges having negative reduced cost, then $z^* + p$ is a lower bound for the TSP instance. This observation follows from LP duality, using the dual variables corresponding to the $x_e \leq 1$ constraints for each edge e.

In our computations, we terminate the column generation process when $|p|$ falls below some fixed value (0.1 in our code). This small penalty in the lower bound is accepted in order to avoid the bad behavior that can arise as we try to complete the column generation while maintaining a small core LP.

9. Computational results

The tests reported in this section were carried out on a Compaq AlphaServer ES40 Model 6/500, with 8 GBytes of random access memory, running

True64 Unix (version 4.0F). The processor speed of the AlphaServer is 500 MHz; the SPEC CPU2000 benchmarks are SPECint2000 = 299 and SPECfp2000 = 382. The Concorde code was compiled with "cc -arch host -04 -g3"; the LP solver used was ILOG CPLEX (version 7.1). In all of our tests, we utilize only a single processor of the AlphaServer (it contains a total of 4 processors).

9.1. Subtour bound

Let V denote the set of cities for an instance of the TSP and let E denote the edge-set of the complete graph on V. Recall that the subtour bound for the TSP is the optimal value of

$$\text{Minimize} \sum (c_e x_e : \ e \in E) \tag{25}$$

$$\text{subject to}$$

$$x(\{v\}, V - \{v\}) = 2 \ \text{ for all } v \in V \tag{26}$$

$$x(S, V - S) \geq 2 \ \text{ for all } S \in V, S \neq \emptyset, S \neq V \tag{27}$$

$$0 \leq x_e \leq 1 \ \text{ for all } e \in E, \tag{28}$$

that is, the optimal value of the LP obtained by appending the set of all subtour inequalities to the degree equations for the instance.

As an initial test of our cutting-plane implementation, we compute the subtour bound for a randomly generated 1,000,000-city Euclidean instance. Our test instance was obtained by specifying the options "-s 99 -k 1000000" in the Concorde code; just as in David Johnson's E1M.0, its cities are points with integer coordinates drawn uniformly from the 1,000,000 by 1,000,000 grid and the cost of an edge is the Euclidean distance between the corresponding points, rounded to the nearest integer.

In Table 6, we report the running times (in CPU hours on the Compaq AlphaServer ES40 5/600) for various choices of the subtour separation heuristics we described in Section 2. In each case, we run the code until the exact separation routine returns without any cuts, and the column generation routine returns without any edges. The running times do not vary widely, but the results indicate that for this type of uniformly distributed TSP instance, it is preferable to include only a subset of the separation heuristics. In the remaining tests in this section, we will restrict our subtour

Table 6. 1,000,000-city Subtour Bound: Choice of Cuts

Cuts	CPU Hours
Connect, interval, shrink, exact	26.75
Connect, shrink, exact	26.89
Connect, interval, exact	24.65
Connect, exact	24.63

cuts to the combination of the connected-component and interval heuristics, together with the Padberg-Rinaldi exact separation routine.

The tests reported in Table 6 were not run with our standard initial edge-set, consisting of the union of 10 heuristically generated tours. Although the tour-union idea performs very well when we are trying to compute a strong lower bound on a TSP instance, an initial set that is more closely tied to the subtour bound is preferable in the present case. In choosing such a set, we begin by computing an approximation to the reduced costs that would occur if we optimized over the subtour-bound LP. One way to do this is to first solve the *degree LP* consisting of only the constraints (26) and (28). The degree LP is a simple network optimization problem that can be solved via a combinatorial primal-dual algorithm, combined with a column generation routine to price over the complete graph, as in Applegate and Cook (1993) and Miller and Pekny (1995). Once we have the dual solution for the degree LP, we can select for each city v the k edges having the least reduced-cost among the edges meeting v. The tests reported in Table 6 were run with the initial edge-set consisting of 4 least-reduced-cost edges meeting each city, together with a tour generated by a greedy algorithm.

In Table 7, we compare the total CPU time needed to obtain the 1,000,000-city subtour bound starting with three different initial edge-sets. The "10 Tours" set is obtained using short runs of Chained Lin-Kernighan to generate the tours; the "4-Nearest" set consists of the 4 least-cost edges meeting each city, together with a tour generated by a greedy algorithm; the "Fractional 4-nearest" is the set used in Table 6.

Table 7. 1,000,000-city Subtour Bound: Choice of Initial Edge-set

Edge-set	CPU Hours
10 Tours	44.26
4-Nearest	30.91
Fractional 4-Nearest	24.31

The best of the results reported in Table 7 was obtained with the fractional 4-nearest edge-set. The running time can be improved further, however, by increasing the density of the set, as we report in Table 8. The best of the runs in this test used the fractional 6-nearest, taking just over 19 hours to compute the subtour bound for this 1,000,000-city instance.

In Table 9, we indicate the growth in the running time used to compute the subtour bound for randomly generated Euclidean instances ranging from size 250,000 up to 1,000,000. The results suggest that the running time is growing as a quadratic function of the number of cities. To explore this, we give a rough profile of the running time for the 1,000,000-city instance in Table 10. In the profile, the "50-nearest pricing" entry is the time spent in repeatedly computing the reduced costs over the 50-nearest edge-set (the first phase of our column generation procedure) and the "Full

Table 8. 1,000,000-city Subtour Bound: Size of Initial Edge-set

Edge-set	CPU Hours
Fractional 4-Nearest	24.31
Fractional 5-Nearest	19.62
Fractional 6-Nearest	19.02
Fractional 7-Nearest	19.16
Fractional 8-Nearest	20.13
Fractional 9-Nearest	20.92
Fractional 10-Nearest	21.97

Table 9. Growth of Running Time for Subtour Bound

Number of Cities	CPU Hours
250,000	1.74
500,000	6.47
1,000,000	24.31

pricing" entry is the time spent to price over the complete set of edges (this was only carried out once, since no negative reduced cost edges were found in this second phase of our column generation procedure).

Table 10. Growth of Running Time for Subtour Bound

Task	CPU Time
Initial edges and tour	0.71%
Connect cuts	0.07%
Interval cuts	0.28%
Exact cuts	0.64%
50-nearest pricing	1.47%
Full pricing	0.70%
LP solve after adding edges	34.42%
LP solve after adding cuts	61.71%

The results of Table 9 and Table 10 suggest that the time spent in the LP solver grows quadratically with the number of cities. In Table 11, we report the number of nonzero coefficients in the final core LP for the three runs that were used in Table 9. The growth in the size of the LP problems

Table 11. Number of Nonzeros in Final LP

Number of cities	Nonzeros
250,000	1,518,920
500,000	3,027,376
1,000,000	6,059,563

appears to be linear, indicating the quadratic behavior is either within the LP solver itself or it is due to the strategy we use for adding cuts and edges to the LP (we add cuts in groups of 2,000 and we add edges in groups of 100; see Section 6).

9.2. 85,900-city TSPLIB instance

The largest test instance in Reinelt's (1991) TSPLIB contains 85,900 cities. This "pla85900" instance was contributed by David S. Johnson; it arose in a programmable logic array application at AT&T in 1986. The cities in pla85900 are specified as coordinates in R^2, and the edge costs are the Euclidean distances rounded up to the next integer.

The best known tour for pla85900 has length 142,384,358; the tour was found by Hisao Tamaki using the algorithm described in Tamaki (2002). The best known lower bound for this instance is 142,307,500, showing that Tamaki's tour is no more than 0.055% away from optimal. This lower bound was found by Concorde, using a short branch-and-cut run.

Although 85,000 cities is well below the target for our large-scale implementation, the code is still an effective way to obtain a good lower bound in a reasonably short amount of CPU time. To illustrate this, we ran Concorde using the connect-interval-exact combination of subtour cuts, together with the separation routines described in Section 4 and the local cuts procedure with $t_{\max} = 8$. In Figure 1, we plot the gap (to Tamaki's tour) versus the CPU time.

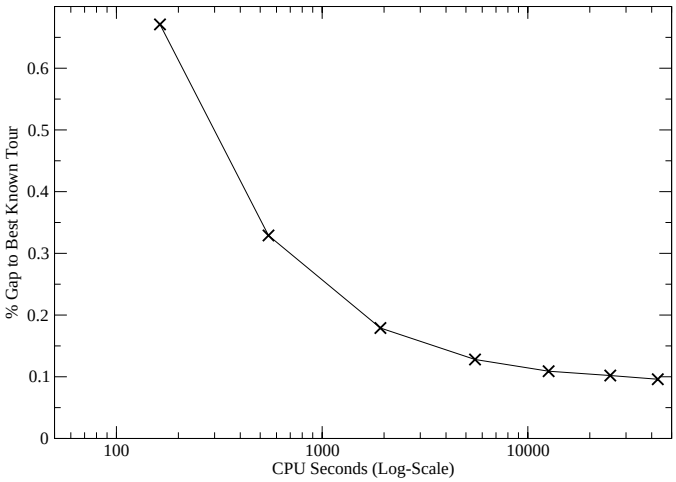

Fig. 1. Concorde run on pla85900

The plot in Figure 1 shows that the improvement in the lower bound tails off as we exhaust the useful cuts that can be supplied by our separation routines, but in under 10 hours of CPU time the code was able to obtain an optimality gap of under 0.1%.

9.3. 1,000,000-city Euclidean TSP

The best known tour for David Johnson's randomly generated Euclidean 1,000,000-city instance E1M.0 has length 713,302,702; it was found by Keld Helsgaun in 2002 using a variant of the LKH heuristic described in Helsgaun (2000).

We ran Concorde on E1M.0 using the same selection of cutting-plane separation routines that were used in Section 9.2, but in this case we gradually increased the size of the local cuts' $t_{\max}$ parameter from 0 up to 28. Each successive run in this study was initialized with the LP and cut pool that were produced in the previous run. The results are reported in Table 12. The "Bound" column gives the final lower bound that was achieved by each run and the "Gap" column reports the % gap to the cost of the tour found by Helsgaun, that is $100 * (713302702 - Bound)/Bound$. The cumulative CPU time is reported in days, again using the Compaq AlphaServer ES40 Model 6/500.

Table 12. Concorde run on 1,000,000-city Euclidean TSP

Cuts	Bound	Gap	Total CPU Days
$t_{\max} = 0$	711088074	0.311%	2.1
$t_{\max} = 8$	712120984	0.166%	11.0
$t_{\max} = 10$	712651618	0.091%	23.6
$t_{\max} = 12$	712746082	0.078%	29.2
$t_{\max} = 14$	712813323	0.068%	38.9
$t_{\max} = 20$	712903086	0.056%	72.1
$t_{\max} = 24$	712954779	0.049%	154.2
$t_{\max} = 28$	713003014	0.042%	308.1

The results in Table 12 demonstrate the ability of the local cuts procedure to permit cutting-plane codes to achieve strong lower bounds on even very large problem instances. The total running time of the study was nearly one year, however, indicating that further progress needs to be made. In this test, approximately 97% of the CPU time was used by the LP solver to produce optimal primal and dual solutions after the addition of cutting planes and after the addition of edges to the core LP, so this is a natural area for future research. We will comment further on the LP solver in the next section.

9.4. World TSP

In this section we study the "World TSP", a 1,904,711-city instance available at `www.math.princeton.edu/tsp/world/`. This instance was created in 2001, using data from the *National Imagery and Mapping Agency*[1] and

[1] `http://164.214.2.59/gns/html/`

from the *Geographic Names Information System*[2] to locate populated points throughout the world. The cities in the World TSP are specified by their latitude and longitude, and the cost of travel between cities is given by an approximation of the great circle distance on the Earth, treating the Earth as a ball. (This cost function is a variation of the TSPLIB GEO-norm, scaled to provide the distance in meters rather than in kilometers.)

The distribution of the points in the World TSP is indicated in Figure 2. The best known tour for this instance was again found by K. Helsgaun, using a variant of the LKH heuristic; the length of the tour is 7,519,173,074 meters.

Fig. 2. World TSP

To study the cutting-plane method on this large instance, we repeated the test we made on the 1,000,000-city instance in the previous section. In this case, we let the local cuts' parameter $t_{\max}$ increase from 0 up to 16; the results are reported in Table 13. We did not attempt to run local cuts with larger values of $t_{\max}$ due to the overall running time of the code.

Table 13. Concorde run on 1,904,711-city World TSP

Cuts	Bound	Gap	Total CPU Days
$t_{\max} = 0$	7500743582	0.245%	12.0
$t_{\max} = 8$	7504218236	0.199%	22.0
$t_{\max} = 12$	7508333052	0.144%	77.9
$t_{\max} = 14$	7510154557	0.120%	163.6
$t_{\max} = 16$	7510752016	0.112%	256.1

As in the 1,000,000-city test, the CPU usage is dominated by the time spent in the LP solver after the addition of cutting planes and after the

[2] `http://geonames.usgs.gov/`

addition of edges. In this case, the portion of time spent solving LP problems was approximately 98% of the total CPU time (and the percentage was growing as the run progressed).

9.5. Conclusions

The computational tests on the 1,000,000-city and World TSPs demonstrate the effectiveness of the mix of cutting planes that have been developed for the TSP. Of particular interest for general large-scale applications of the cutting-plane method may be the cut-alteration procedures (Section 4) and the local-cut procedure (Section 5), since both of these themes can be adapted for applications beyond the context of the TSP.

The tests also indicate the need for further research into solution methods for large-scale LP problems—the CPU time in our tests was dominated by the time spent in the LP solver. We comment on this in more detail below.

To give an indication of the properties of the LP problems that arose in our computations, we isolated a single LP that was created by adding 2,000 subtour inequalities to a previously solved core LP during our test of the World TSP. We refer to this problem as LP2000; it was taken from the end of the World TSP run.

In Table 14, we report some statistics on the size of LP2000. Note that the number of rows includes the original 1,904,711 degree constraints, so at this point in the computation the core LP contained 731,182 cutting planes. Note also the sparsity of LP2000 (the "Nonzeros" entry counts the number of nonzero coefficients in the constraint matrix); this is due in part to the internal representation of the LP we described in Section 7.1.

Table 14. LP2000 Statistics

Rows	Columns	Nonzeros	Nonzeros per Column
2,635,893	4,446,024	23,397,782	5.26

Starting with the optimal basis for the previously solved core LP, the CPLEX code produced an optimal solution for LP2000 in 46,341 seconds on the Compaq AlphaServer ES40 6/500, using the dual-steepest-edge simplex algorithm (starting with unit norms).

Although the running time of the CPLEX solver is remarkably small for a problem of the size and complexity of LP2000, our study certainly suggests that the solution of LP problems remains the bottleneck in implementations of the Dantzig et al. method for large-scale instances. One possibility for overcoming this difficulty is to explore the use of alternative methods for solving the LP problems, rather than relying on the simplex algorithm. Indeed, LP2000 can be solved in approximately 11,000 seconds if we use the CPLEX barrier code and run on all 4 processors of the AlphaServer ES40

(at the present time there is no effective way to run the simplex algorithm in a parallel environment on LP instances of the size and shape of LP2000). A difficulty with this approach, however, is that our cutting-plane separation routines have not been designed to deal effectively with the dense solutions produced by barrier codes (as opposed to the basic solutions found by the simplex algorithm). Although it is possible to use a crossover routine to obtain a basic optimal solution from a barrier solution, in this instance the CPLEX crossover function required over six days of CPU time to carry out the conversion.

Acknowledgements. We would like to thank the late Michael Pearlman for his tireless technical support that provided us with a superb computational platform for carrying out the tests reported in this study.

References

1. Agarwala, R., D. L. Applegate, D. Maglott, G. D. Schuler, A. A. Schäffer. 2000. A fast and scalable radiation hybrid map construction and integration strategy. Genome Research **10**, 350–364.
2. Applegate, D., R. Bixby, V. Chvátal, W. Cook. 1995. Finding cuts in the TSP (A preliminary report). DIMACS Technical Report 95-05. DIMACS, Rutgers University, New Brunswick, New Jersey, USA.
3. Applegate, D., R. Bixby, V. Chvátal, W. Cook. 1998. On the solution of traveling salesman problems. Documenta Mathematica Journal der Deutschen Mathematiker-Vereinigung, International Congress of Mathematicians. 645–656.
4. Applegate, D., R. Bixby, V. Chvátal, W. Cook. 2001. TSP cuts which do not conform to the template paradigm. M. Jünger, D. Naddef, eds. *Computational Combinatorial Optimization*. Springer, Heidelberg, Germany. 261–304.
5. Applegate, D., R. Bixby, V. Chvátal, W. Cook. 2003. Concorde. Available at `www.math.princeton.edu/tsp` .
6. Applegate, D., W. Cook. 1993. Solving large-scale matching problems. D. S Johnson, C. C. McGeoch, eds. *Algorithms for Network Flows and Matching*. American Mathematical Society, Providence, Rhode Island, USA. 557–576.
7. Applegate, D., W. Cook, A. Rohe. 2003. Chained Lin-Kernighan for large traveling salesman problems. INFORMS Journal on Computing **15**, 82–92.
8. Balas, E. 1975. Facets of the knapsack polytope. Mathematical Programming **8**, 146–164.
9. Bentley, J. L. 1992. Fast algorithms for geometric traveling salesman problems. ORSA Journal on Computing **4**, 387–411.
10. Bixby, R., M. Fenelon, Z. Gu, E. Rothberg, R. Wunderling. 2000. MIP: Theory and practice - closing the gap. M. J. D. Powell, S. Scholtes, eds. *System Modelling and Optimization: Methods, Theory and Applications*. Kluwer Academic Publishers, Dordrecht, The Netherlands. 19–49.
11. Bixby, R., M. Fenelon, Z. Gu, E. Rothberg, R. Wunderling. 2003. Mixed-Integer Programming: A Progress Report. M. Grötschel, ed. *The Sharpest Cut, Festschrift in honor of Manfred Padberg's 60th birthday*. SIAM, Philadelphia. To appear.
12. Boyd, E. A. 1993. Generating Fenchel cutting planes for knapsack polyhedra. SIAM Journal of Optimization **3**, 734–750.
13. Boyd, E. A. 1994. Fenchel cutting planes for integer programs. Operations Research **42**, 53–64.
14. Chekuri, C. S., A. V. Goldberg, D. R. Karger, M. S. Levine, C. Stein. 1997. Experimental study of minimum cut algorithms. *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*. ACM Press, New York, USA. 324–333. (The full version of the paper is available at `www.cs.dartmouth.edu/~cliff/papers/MinCutImplement.ps.gz` .)

15. Christof, T., G. Reinelt. 1995. Parallel cutting plane generation for the TSP. P. Fritzson, L. Finmo, eds. *Parallel Programming and Applications*. IOS Press, Amsterdam, The Netherlands. 163–169.
16. Chvátal, V. 1973. Edmonds polytopes and weakly hamiltonian graphs. Mathematical Programming **5**, 29–40.
17. Chvátal, V. 1983. *Linear Programming*. W. H. Freeman and Company, New York, USA.
18. Clochard, J.-M., D. Naddef. 1993. Using path inequalities in a branch and cut code for the symmetric traveling salesman problem. G. Rinaldi, L. Wolsey, eds. *Third IPCO Conference*. 291–311.
19. Cook, W., P. D. Seymour. 2003. Tour merging via branch decomposition. To appear in INFORMS Journal on Computing.
20. Cormen, T. H., C. E. Leiserson, R. L. Rivest. 1990. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, USA.
21. Cornuéjols, G., J. Fonlupt, D. Naddef. 1985. The traveling salesman problem on a graph and some related integer polyhedra. Mathematical Programming **33**, 1–27.
22. Craven, B. D. 1988. *Fractional Programming*. Heldermann, Berlin, Germany.
23. Crowder, H., E. L. Johnson, M. Padberg. 1983. Solving large-scale zero-one linear programming problems. Operations Research **31**, 803–834.
24. Crowder, H., M. W. Padberg. 1980. Solving large-scale symmetric travelling salesman problems to optimality. Management Science **26**, 495–509.
25. DIMACS. 2001. 8th DIMACS implementation challenge: the traveling salesman problem. `www.research.att.com/~dsj/chtsp/` .
26. Edmonds, J. 1965. Maximum matching and a polyhedron with 0,1-vertices. Journal of Research of the National Bureau of Standards—B **69B**, 125–130.
27. Fleischer, L. 1999. Building chain and cactus representations of all minimum cuts from Hao-Orlin in the same asymptotic run time. Journal of Algorithms **33**, 51–72.
28. Fleischer, L., É. Tardos. 1999. Separating maximally violated comb inequalities in planar graphs. Mathematics of Operations Research **24**, 130–148.
29. Ford, L. R., D. R. Fulkerson. 1962. *Flows in Networks*. Princeton University Press, Princeton, NJ, USA.
30. Goldberg, A. V. 1985. A new max-flow algorithm. Technical Report MIT/LCS/TM 291. Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA.
31. Gomory, R.E. 1969. Some polyhedra related to combinatorial problems. Linear Algebra and Its Applications **2**, 451–558.
32. Grötschel, M., O. Holland. 1987. A cutting-plane algorithm for minimum perfect 2-matchings. Computing **39**, 327–344.
33. Grötschel, M., O. Holland. 1991. Solution of large-scale symmetric travelling salesman problems. Mathematical Programming **51**, 141–202.
34. Grötschel, M., M. Padberg. 1979a. On the symmetric traveling salesman problem I: inequalities. Mathematical Programming **16**, 265–280.
35. Grötschel, M., M. Padberg. 1979b. On the symmetric traveling salesman problem II: lifting theorems and facets. Mathematical Programming **16**, 281–302.
36. Grötschel, M., M. Padberg. 1985. Polyhedral theory. E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, D. B. Shmoys, eds, *The Traveling Salesman Problem*. John Wiley & Sons, Chichester, UK. 252–305.
37. Hammer, P. L., E. L. Johnson, U. N. Peled. 1975. Facets of regular 0-1 polytopes. Mathematical Programming **8**, 179–206.
38. Harel, D., R. E. Tarjan. 1984. Fast algorithms for finding nearest common ancestors. SIAM Journal on Computing **13**, 338–355.
39. Held, M., R. M. Karp. 1971. The traveling-salesman problem and minimum spanning trees: part II. Mathematical Programming **1**, 6–25.
40. Helsgaun, K. 2000. An effective implementation of the Lin-Kernighan traveling salesman heuristic. European Journal of Operational Research **126**, 106–130. The `LKH` code is available at `www.dat.ruc.dk/~keld/research/LKH/` .
41. Hong, S. 1972. *A Linear Programming Approach for the Traveling Salesman Problem*. Ph.D. Thesis. The Johns Hopkins University, Baltimore, Maryland, USA.
42. Huffman, D. A. 1952. A method for the construction of minimum-redundancy codes. Proceedings of the IRE **40**, 1098–1101.

43. Johnson, D. S., L. A. McGeoch. 1997. The traveling salesman problem: a case study. E. Aarts, J. K. Lenstra, eds. *Local Search in Combinatorial Optimization*. John Wiley & Sons, Chichester, UK. 215–310.
44. Johnson, D. S., L. A. McGeoch. 2002. Experimental analysis of heuristics for the STSP. G. Gutin, A. Punnen, eds. *The Traveling Salesman Problem and its Variations*. Kluwer Academic Publishers, Dordrecht, The Netherlands. 369–443.
45. Jünger, M., G. Reinelt, G. Rinaldi. 1995. The traveling salesman problem. M. Ball, T. Magnanti, C. L. Monma, G. Nemhauser, eds. *Handbook on Operations Research and Management Sciences: Networks*. North Holland, Amsterdam, The Netherlands.
46. Jünger, M., G. Reinelt, G. Rinaldi. 1997. The Traveling Salesman Problem. M. Dell'Amico, F. Maffioli, S. Martello, eds. *Annotated Bibliographies in Combinatorial Optimization*. John Wiley & Sons, Chichester, UK. 199–221.
47. Jünger, M., G. Reinelt, S. Thienel. 1994. "Provably good solutions for the traveling salesman problem". Zeitschrift für Operations Research **40**, 183–217.
48. Jünger, M., G. Reinelt, S. Thienel. 1995. Practical problem solving with cutting plane algorithms. W. Cook, L. Lovász, P. Seymour, eds. *Combinatorial Optimization*. DIMACS Series in Discrete Mathematics and Theoretical Computer Science **20**. American Mathematical Society, Providence, Rhode Island, USA. 111–152.
49. Jünger, M., G. Rinaldi, S. Thienel. 2000. Practical performance of minimum cut algorithms. Algorithmica **26**, 172–195.
50. Karger, D. R., C. Stein. 1996. A new approach to the minimum cut problem. Journal of the ACM **43**, 601–640.
51. Knuth, D. 1968. *Fundamental Algorithms*, Adison Wesley, Reading, Massachusetts, USA.
52. Land, A. 1979. The solution of some 100-city travelling salesman problems. Technical Report. London School of Economics, London, UK.
53. Letchford, A. N. 2000. Separating a superclass of comb inequalities in planar graphs. Mathematics of Operations Research **25**, 443–454.
54. Letchford, A. N., A. Lodi. 2002. Polynomial-time separation of simple comb inequalities. W. J. Cook, A. S. Schulz, eds. *Integer Programming and Combinatorial Optimization*. Lecture Notes in Computer Science **2337**. Springer, Heidelberg, Germany. 93–108.
55. Levine, M. 1999. Finding the right cutting planes for the TSP. M. T. Goodrich, C. C. McGeoch, eds. *Algorithm Engineering and Experimentation, International Workshop ALEXNEX'99*. Lecture Notes in Computer Science **1619**. Springer, Heidelberg, Germany. 266–281.
56. Lin, S., B. W. Kernighan. 1973. An effective heuristic algorithm for the traveling-salesman problem. Operations Research **21**, 498–516.
57. Marchand, H., A. Martin, R. Weismantel, L. A. Wolsey. 1999. Cutting planes in integer and mixed-integer programming. Technical Report CORE DP9953. Université Catholique de Louvain, Louvain-la-Neuve, Belgium.
58. Martin, O., S. W. Otto, E. W. Felten. 1991. Large-step Markov chains for the traveling salesman problem. *Complex Systems* **5**, 299–326.
59. Miliotis, P. 1978. Using cutting planes to solve the symmetric travelling salesman problem. Mathematical Programming **15**, 177–188.
60. Miller, D. L., J. F. Pekny. 1995. A staged primal-dual algorithm for perfect b-matching with edge capacities. ORSA Journal on Computing **7**, 298–320.
61. Naddef, D. 2002. Polyhedral theory and branch-and-cut algorithms for the symmetric TSP. G. Gutin, A. Punnen, eds. *The Traveling Salesman Problem and its Variations*. Kluwer Academic Publishers, Dordrecht, The Netherlands. 29–116.
62. Naddef, D., G. Rinaldi. 1992. The graphical relaxation: A new framework for the symmetric traveling salesman polytope. Mathematical Programming **58**, 53–88.
63. Naddef, D., S. Thienel. 2002a. Efficient separation routines for the symmetric traveling salesman problem I: general tools and comb separation. Mathematical Programming **92**, 237–255.
64. Naddef, D., S. Thienel. 2002b. Efficient separation routines for the symmetric traveling salesman problem II: separating multi handle inequalities. Mathematical Programming **92**, 257–283.
65. Nemhauser, G. L, L. A. Wolsey. 1988. *Integer and Combinatorial Optimization*. John Wiley & Sons, New York, USA.
66. Padberg, M. W. On the facial structure of set packing polyhedra. Mathematical Programming **5**, 199–215.

67. Padberg, M. W. A note on zero-one programming. Operations Research **23**, 833–837.
68. Padberg, M., M. Grötschel. 1985. Polyhedral computations. E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, D. B. Shmoys, eds, *The Traveling Salesman Problem.* John Wiley & Sons, Chichester, UK. 307–360.
69. Padberg, M. W., S. Hong. 1980. On the symmetric travelling salesman problem: a computational study. Mathematical Programming Study **12**, 78–107.
70. Padberg, M. W., M. R. Rao. 1982. Odd minimum cut-sets and b-matchings. Mathematics of Operations Research **7**, 67–80.
71. Padberg, M. W., G. Rinaldi. 1990a. An efficient algorithm for the minimum capacity cut problem. Mathematical Programming **47**, 19–36.
72. Padberg, M. W., G. Rinaldi. 1990b. Facet identification for the symmetric traveling salesman polytope. Mathematical Programming **47**, 219–257.
73. Padberg, M. W., G. Rinaldi. 1991. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. SIAM Review **33**, 60–100.
74. Pulleyblank, W. R. 1973. *Faces of Matching Polyhedra.* Ph.D. Thesis. Department of Combinatorics and Optimization, University of Waterloo, Waterloo, Ontario, Canada.
75. Reinelt, G. 1991. TSPLIB – A traveling salesman problem library. ORSA Journal on Computing **3**, 376–384. An updated version of the library is available at `http://www.iwr.uni-heidelberg.de/iwr/comopt/software/TSPLIB95/`.
76. Schieber, B., U. Vishkin. 1988. On finding lowest common ancestors: simplification and parallelization. SIAM Journal on Computing **17**, 1253–1262.
77. Schrijver, A. 1986. *Theory of Linear and Integer Programming.* John Wiley & Sons, Chichester, UK.
78. Stancu-Minasian, I. M. 1997. *Fractional Programming.* Kluwer, Dordrecht, The Netherlands.
79. Tamaki, H. 2002. Alternating cycles contribution: a tour merging strategy for the traveling salesman problem. Submitted.
80. Tarjan, R. E. 1975. Efficiency of a good but not linear set union algorithm. Journal of the Association of Computing Machinery **22**, 215–225.
81. Tarjan, R. E. 1983. *Data Strutures and Network Algorithms.* SIAM, Philadelphia.
82. Tarjan, R. E., J. van Leeuwen. 1984. Worst-case analysis of set union algorithms. Journal of the Association of Computing Machinery. **31**, 245–281.
83. Vanderbei, R. J. 2001. *Linear Programming: Foundations and Extensions.* Kluwer Academic Publishers, Boston, USA.
84. Wenger, K. M. 2002. A new approach to cactus construction applied to TSP support graphs. W. J. Cook, A. S. Schulz, eds. *Integer Programming and Combinatorial Optimization.* Lecture Notes in Computer Science **2337**. Springer, Heidelberg, Germany. 109–126.
85. Wolsey, L. A. 1975. Faces for a linear inequality in 0-1 variables. Mathematical Programming **8**, 165–178.
86. Wolsey, L. A. Facets and strong valid inequalities for integer programs. Operations Research **24**, 367–372.
87. Wolsey, L. A. 1998. *Integer Programming.* John Wiley & Sons, New York, USA.