AN ALGORITHM FOR THE TRAVELING SALESMAN PROBLEM

Jean D. C. Little
Massachusetts Institute of Technology
Katta G. Murty*
Indian Statistical Institute
Dura W. Sweeney
International Business Machines Corporation

Caroline Karol
Case Institute of Technology
(Received March 5, 1962)

A branch and bound algorithm is presented for solving the traveling salesman problem. The set of all tours (traversable solutions) is broken up into successively small subsets by a procedure called branching. For each subset a lower bound on the length of the tours therein is calculated. Eventually a subset is found that contains a single tour whose length is less than or equal to some lower bound for every tour. The motivation of the branching and the calculation of the lower bounds are based on ideas frequently used in solving assignment problems. Computationally, the algorithm extends the size of the problem that can reasonably be solved without using methods special to the particular problem.

THE TRAVELING salesman problem is easy to state: A salesman is required to visit each of n cities only once and return to the start. In what order should he visit the cities to minimize the total distance traveled? For distances we can substitute time, cost, or other measure of effectiveness as desired. Distance is easy between all city pairs are presumed known.

The problem has become famous because it combines ease of statement with difficulty of solution. The difficulty is entirely computational, not a solution obviously exists. There are (n−1)! possible tours, one order of which must give minimum cost. (The minimum cost could conceivably be infinite—it is conventional to assign an infinite cost to travel between city pairs that have no direct connection.)

The traveling salesman problem recently achieved national prominence.

* Work done while on a study assignment at Case Institute of Technology.
† Work done while a Sloan Fellow at M.I.T.

when a salesmen is up to 100 cities long.

We shall not delve into the details of the algorithm, but shall just mention the main steps involved:

1. Construct an initial tour by using the nearest neighbor method or another heuristic.
2. Use the algorithm to improve the initial tour by finding a better tour.
3. If the algorithm finds a tour that is better than the current best tour, update the best tour.
4. Repeat steps 2 and 3 until no better tour is found.

After a tour is found, the total distance traveled is calculated. This process is repeated until the best tour is found.
Traveling Salesman Problem

973

time a soap company used it as the basis of a promotional contest. Prices up to $10,000 were offered for finding the most correct lines in a particular 33-city problem. Quite a few people found the best tour. (The eye-breaking contest for these successful mathematicians was to complete a statement of 25 words or less on "I like . . . because . . . .") A number of people, perhaps a little over-educated, wrote the company that the problem was impossible—an interesting misinterpretation of the state of the art.

For the early history of the problem, see Flood.60 In recent years a number of methods for solving the problem have been put forward. Some suffer from inefficiency, others produce solutions that are not necessarily optimal, and still others require intuitive judgments that would be hard to program on a computer. For a detailed discussion, see Gonzalez.66 We shall restrict our discussion to methods that (1) guarantee optimality, (2) seem reasonable to program, and (3) are general, i.e., not ad hoc to the specific numerical problem.

Among such methods the approach that has been carried furthest computationally is that of dynamic programming. Held and Karp64 and Held and Gonzalez66 have independently applied the method and have solved various test problems on computers. Gonzalez programmed an IBM 1620 to handle problems up to 10 cities. In his work the time to solve a problem grew somewhat faster than exponentially as the number of cities increased.

A 5-city problem took 10 seconds, a 10-city problem took 8 minutes, and the addition of one more city multiplied the time by a factor, which, for 10 cities, had grown to 3. Storage requirements expanded with similar rapidity.

Held and Karp64 have solved problems up to 13 cities by dynamic programming using an IBM 7090. A 13-city problem required 17 seconds.

At the current level of computing power, it may be feasible to solve 20-city problems in a few minutes on an IBM 7090. A 20-city problem would require almost 10 hours. Storage requirements, however, may become prohibitive before then. For larger problems than 13 cities, Held and Karp develop an approximation that seems to work well but does not guarantee an optimal tour.

We have found two papers in which the problem has been approached by methods similar to our "branch and bound" algorithm. Rosenman, Swart, and Grover63 in an unpublished paper apply ideas that they have called combinatorial programming.64 To illustrate their method they present a 13-city problem. It was solved in 8 man-days. We have solved their problem by hand in about 34 hours. Eastman,64 in an unpublished doctoral thesis and laboratory report, presents a method of enumeration and several variations on it. His work and ours contain strong similarities. However, to use our terminology, his ways of choosing branches and of
calculating bounds are different from ours. He basically solves a sequence
of assignment problems that give his bounds. We have a simpler method,
and for branching we use a device with quite a different motivation. The
biggest problem Eastman solves is 10 cities and he gives no computation
times, so his effective comparison is difficult to make.
Most published problems are symmetric, i.e., the distance from city i
to city j is the same as from j to i. The algorithm to be presented next
works for asymmetric problems; in fact, it seems to work better. As-
ymmetric problems arise in various applications. As an example from pro-
duction scheduling, suppose that there is a production cycle of some
fixed period, during which an assembly line must produce each of n dif-
ferent models. The cost of switching from model i to model j is cij. What
size of production scale minimizes total setup cost? This is a traveling sale-
sman problem in which it would not necessarily be expected that z = cij.
To summarize, 18 cities is the largest problem which we know that
has been solved by a general method which guarantees optimality and
which can reasonably be programmed for a computer. Our method ap-
preciably increases this number. However, the time required increases
at least exponentially with the number of cities and eventually, of course,
becomes prohibitive. Detailed results are given below.

THE ALGORITHM
The basic method will be to break up the set of all tours into smaller sub-
sets and to calculate for each of them a lower bound on the cost (length)
of the best tour therein. The bounds guided the partitioning of the
subsets and eventually identify an optimal tour—when a subset is such
that contains a single tour whose cost is less than or equal to the lower
bounds for all other subsets, that tour is optimal.
The subsets of tours are conveniently represented as the nodes of a
tree and the process of partitioning as a branching of the tree. Hence, we
called the method 'branch and bound.'

The algorithm will simultaneously be explained and illustrated by a
numerical example. The explanation does not require reference to its
example, however, for those readers who wish to skip it.

Notation
The tours of the traveling salesman problem form a matrix. Let v:
set of cities be indexed by i = 1, ..., n. The entry in row i and column j of the
matrix is the cost for going from city i to city j. Let
\[ C = [c_{ij}] \]
be the cost matrix.

C will start out as the original cost matrix of the problem but will undergo
various transformations as the algorithm proceeds.

Traveling
A tour, i, can be represented
\[ i = (i_1, i_2, ..., i_m) \]

which form a circuit going to i_i. This represents an arc or edge of the
traveling salesman problem. The sum of the matrix elements
\[ z(i) \]

is the cost of the tour.

Notice that the i_\i always picks

Notice that that i always picks

\[ x_i, y_i \]

Notice that that i always picks

\[ x_i, y_i \]

Notice that that i always picks

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]

\[ z(i) = \sum c_{ij} \]

\[ x_i, y_i \]
from ours. He basically solves a sequence of his bounds. We have a simpler method, with quite a different motivation. The
n = 10 cities and he gives no computation second are difficult to make.
re symmetric, i.e., the distance from city i to a. The algorithm to be presented also
i to city, in fact, it seems to work better. Asym-
mus applications. As an example from pro-
state there is a production cycle of some time-
ockly time must produce each of a different
from model r to model j, is c_{i,j}. What order
what is we will call next set up? This is a traveling sal-
not necessarily be expected that c_{i,i} = c_{i,j}.
the largest problem which we know about even with a method which guarantees optimality and
struggle for a computer. Our method ap-
n. However, the time required increases as
number of cities and eventually, of course, it
fails. The algorithm proceeds.

To Algorithm

break up the set of all tours into smaller and
as for each of them a lower bound on the cost
n. The bounds guide the partitioning of the
set an optimal tour—when a subset is found
then cost is less than or equal to the lower
set tour is optimal. Consequently represented as the nodes of a
as a branching of the tree. Hence we
and bound.

because the planning does not require reference to the
nals who wish to skip it.

t salesman problem form a matrix. Let the
node of the tree. The entry in row i and column j of
the salesmen problem. Let the

\[ c_{i,j} \] = cost matrix.

\[ \text{cost matrix of the problem but will undergo} \]

\[ \text{algorithm proceeds.} \]

**Traveling Salesman Problem**

A tour, \( t \), can be represented as a set of ordered city pairs, e.g.,

\[ t = [(i_1, v_1), (i_2, v_2), \ldots, (i_n, v_n)] \]

which form a circuit going to each city once and only once. Each \((i,j)\) represents an arc or leg of the trip. The cost of a tour, \( c_t \), under a matrix, \( C \), is the sum of the matrix elements picked out by \( t \) and will be denoted by \( z(t) \):

\[ z(t) = \sum_{i,j} c_{i,j} \]

Notice that if \( t \) always picks out one and only one cost in each row and
in each column. Also, let

\[ X, Y, T \] = nodes of the tree;

\[ w(X) = \text{a lower bound on the cost of the tours of } X, \text{i.e., } z(t) \geq w(X) \]

for a tour of \( X \);

\[ a = \text{the cost of the best tour found so far in the algorithm.} \]

**Lower Bounds**

A useful concept in constructing lower bounds will be that of reduction. If a constant, \( \delta \), is subtracted from each element of a row of the cost matrix, the cost of any tour under the new matrix is \( \delta \) less than under the old.

This is because every tour must contain one and only one element from
that row. The relative costs of all tours are unchanged, however, and so
any tour optimal under the old will be optimal under the new.

The process of subtracting the smallest element of a row from each
element in the row will be called reducing the row. A matrix with non-
negative elements and at least one zero in each row and column will be
called a reduced matrix and may be obtained, for example, by reducing
rows and columns. If \( z(t) \) is the cost of a tour \( t \) under a matrix before
reduction, \( z(t) \) the cost under the matrix afterward, and \( \delta \) the sum
of constants used in making the reduction, then

\[ z(t) = k + a(t). \]

Since a reduced matrix contains only nonnegative elements, a constitutes
a lower bound on the cost of \( t \) under the old matrix.

Consider then the 8-city problem shown in Fig. 1. Reduction of
the matrix by rows, then columns, gives the matrix of Fig. 2. The total re-
duction is 48 so that \( z(t) \leq 48 \) for all \( t \).

**Branching**

The splitting of the set of all tours into disjoint subsets will be repre-
sented by the branching of a tree, as illustrated in Fig. 3. The node con-
taining 'all tours' is self-explanatory. The node containing \( i,j \) represents
all tours which include the city pair \((i,j)\). The node containing \( i,j \) repre-
### Fig. 1. Cost matrix for a 6-city problem. A typical tour might be \( t=(1,2,3,4,5,6) \), which has the cost length \( x=1+2+2+5+6+2=17 \).

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(\infty)</td>
<td>27</td>
<td>45</td>
<td>16</td>
<td>30</td>
<td>26</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>(\infty)</td>
<td>16</td>
<td>1</td>
<td>30</td>
<td>25</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>13</td>
<td>(\infty)</td>
<td>35</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>21</td>
<td>16</td>
<td>25</td>
<td>(\infty)</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>46</td>
<td>27</td>
<td>48</td>
<td>(\infty)</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>23</td>
<td>5</td>
<td>5</td>
<td>9</td>
<td>5</td>
<td>(\infty)</td>
</tr>
</tbody>
</table>

The node containing \((i,j)\) represents all tours that include \((i,j)\) but not \((j,i)\), whereas \((i,j)\) represents all tours that include both \((i,j)\) and \((j,i)\). In general, by tracing from a node, \(X\), back to the start, we can pick up which city pairs are committed to appear in the tours of \(X\) and which are forbidden from appearing. If the branching process is carried far enough, some node will eventually represent a single tour. Notice that at any stage...

### Fig. 2. Cost matrix after reducing rows and columns. Circled numbers are values of \(b(i)\).

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(\infty)</td>
<td>11</td>
<td>(\infty)</td>
<td>14</td>
<td>10</td>
<td>61</td>
</tr>
<tr>
<td>2</td>
<td>(\infty)</td>
<td>15</td>
<td>(\infty)</td>
<td>29</td>
<td>24</td>
<td>(\infty)</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>13</td>
<td>(\infty)</td>
<td>55</td>
<td>5</td>
<td>(\infty)</td>
</tr>
<tr>
<td>4</td>
<td>(\infty)</td>
<td>(\infty)</td>
<td>9</td>
<td>(\infty)</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>41</td>
<td>22</td>
<td>43</td>
<td>(\infty)</td>
<td>(\infty)</td>
</tr>
<tr>
<td>6</td>
<td>13</td>
<td>(\infty)</td>
<td>4</td>
<td>(\infty)</td>
<td>5</td>
<td>(\infty)</td>
</tr>
</tbody>
</table>

### Fig. 3.3

**Bus 2** starts the calculation by solving the problem into \(C\), setting \(X=1\) to replace the cost of the best tour so far in \(t\).

**Bus 3** selects \((k,l)\), the city pair with the smallest cost. The two tours are split because they are both new, and the cost of each\(c((k,l))\) is added to the cost of the best tour so far. Since city \(i\) appears in \((k,l)\) and \((l,i)\), the sum of the costs of the two city pairs is added to the cost of the best tour so far.
The process, the union of the sets represented by the terminal nodes is the set of all tours.

When a node $X$ branches into two further nodes, the node with the newly committed city pair will frequently be called $Y$ and the node with its newly forbidden city pair $Z$.

**Flow Chart**

The workings of the algorithm will be explained by tracing through the flowchart of Fig. 4.

---

**Box 1** starts the calculation by putting the original cost matrix of the problem into $C$, setting $X=1$ to represent the node, 'all tours,' and setting the cost of the best tour as far to infinity.

**Box 2** selects $(k_j)$, the city pair on which to base the next branching. The goal in doing this is to split the tours of $X$ into a subset $(Y)$ that is quite likely to include the best tour of the node and another $(Z)$ that is quite unlikely to include it. Possible low cost tours to consider for $Y$ are those involving an $(i_j)$ for which $c(i_j)=0$.

Consider therefore the tours for tours that do not contain $(i_j)$, i.e., possible tours for $Z$. Since city $i$ must be reached from some city, these tours must incur at least the cost of the smallest element in row $i$, excluding $(i_j)$. Since city $j$ must connect to some city, the tours must incur at least the cost of the smallest element in column $j$, excluding $(i_j)$. Call the sum of these two costs $\theta(i_j)$. We shall choose $(i_j)$ to be that city pair that gives the largest $\theta(i_j)$. [This amounts to a search over $(i_j)$ such that $c(i_j)=0$, since otherwise $\theta(i_j)=0$.] Notice that, if $c(i_j)$ is
START
1. \( C \), original cost matrix

2. Reduce \( C \); label \( w(X) \) and \( k \) for each \( X \) in \( (\cdot, X) \) as usual.\( \uparrow \)

3. Choose \( X \), for next tree extension, so that \( w(X) = \max \{ w(Y) \}, \) where \( Y \) is a feasible pair; 1st branch of \( X \) is \( a \), reducing \( c_{ij} \), with \( c_{ij} \) in column \( j \), increasing \( c_{ij} \);

4. Make a branch from \( X \) to \( Y \), the \( i,j \) node. Delete row \( i \) and column \( j \) in \( C \). Find \( f(i, j) \);\( \uparrow \)

5. Make a branch from \( X \) to \( Y \), the \( k,l \) node. Delete row \( k \) and column \( l \) in \( C \). Find \( f(k, l) \);\( \uparrow \)

6. Select each \( X \), then which \( w(X) \) on the multistart reduced matrix, which has smallest \( w(X) \).

7. Suppose next \( X \) from which to base \( Y \) on the multistart reduced matrix, which has smallest \( w(X) \).

8. If \( w(Y) < w(X) \), then \( \text{Yes} \); otherwise, \( \text{No} \).

9. Does \( X = Y \) of base \( Y \)?

10. Set up \( C \) for \( X \):

   (1) \( C \), original cost matrix
   (2)  Row pairs \( (i, j) \) connected to be in tours of \( X \).
   (3)  For each such \( (i, j) \), delete row \( i \) and column \( j \) of \( C \). For each path among the \( (i, j) \) found starting city \( i \) and ending city \( j \) in \( X \), and set \( c_{ij} = \infty \). For each \( k, l \) path among the \( (i, j) \) found starting city \( k \) and ending city \( l \) in \( X \), set \( c_{kl} = \infty \).
   (4)  Reduce \( C \).
   (5)  Label \( X \) with \( w(X) + \gamma \) (sum of reducing constants).

11. \( w(Y) < w(X) \)?

12. \( \gamma = w(Y) \), since \( \text{new} \).

Fig. 6. Flow chart of the algorithm.

Legend:

1. \( C \), original cost matrix
2. \( w(X) \), reducing \( c_{ij} \), with \( c_{ij} \) in column \( j \), increasing \( c_{ij} \);
3. \( f(i, j) \), branch of \( X \) is \( a \), reducing \( c_{ij} \);
4. \( f(k, l) \), branch from \( X \) to \( Y \), the \( k,l \) node.
5. \( w(Y) \), smallest \( w(Y) \) on the multistart reduced matrix, which has smallest \( w(X) \).
6. \( X = Y \) of base \( Y \), which \( w(X) \) on the multistart reduced matrix, which has smallest \( w(X) \).
7. \( w(Y) < w(X) \)
8. \( w(Y) \), reducing \( c_{ij} \), with \( c_{ij} \) in column \( j \), increasing \( c_{ij} \).
9. \( \text{No} \), \( \text{Yes} \)
10. \( C \), original cost matrix
11. \( f(i, j) \), row pairs \( (i, j) \) connected to be in tours of \( X \).
12. \( f(k, l) \), row pairs \( (i, j) \) connected to be in tours of \( X \).
13. \( \text{Yes} \), \( \text{No} \)
at infinity and then row $i$ and column $j$ are reduced, the sum of the reducing constants is $\phi(i,j)$.

For the example, the $\phi(i,j)$ values are written in small circles placed in the cells of the zero of Fig. 2. The largest $\phi$ is $\phi(1,4)=10+0=10$ and so (1,4) will be the first city pair used for branching.

Bar 4 extends the tree from node X to Y. As will be shown below, $c(Y)=w(X)+\phi(4,5)$. In the example $w(Y)=10+48=58$ and the node, so labeled in Fig. 3b.

Bar 5 sets up Y. Since the city pair (1,2) is now committed to the tree, row k and column l are no longer needed and are deleted from C. Next, notice that (1,2) will be part of some connected path generated by the city pairs that have been committed to the tours of Y. Suppose the path starts at city p and ends at city q. (Possibly p = k or $q = l$ or both.) The contracting of m to p should be forbidden (it would create a subtour a circuit with less than a city) and no subtour can be part of a tour. Therefore, let $c(m,p)=\infty$.

After these modifications C can perhaps be reduced in the following sense: row m, column p, any column that had a zero in row k, and any row that had a zero in column l. All other rows and columns contain some zero that cannot have been disturbed. Let k be the sum of the new reducing constants. The lower bound for Y will now be shown to be

$$w(Y)=w(X)+k.$$
therefore in \( X_i \). If we let \( a(i,j) \) be the cost of the uncommitted city pairs of \( X_i \) under \( C_i \) and \( a_1(i) \) be the cost of the uncommitted city pairs of \( X_i \) under \( C_0 \),

\[
a(i, j) = \sum a(i, j) + a_1(i),
\]

or using (2), assumed true for \( X_i \),

\[
x(i) = w(X_i) + \sum a(i, j) + a_1(i)
\]

\[
= w(X_i) + a_1(i)
\]

so that (2) is true for \( X_i \), as was to be shown.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>∞</td>
<td>14</td>
<td>27</td>
<td>0</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>∞</td>
<td>14</td>
<td>0</td>
<td>28</td>
<td>23</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>13</td>
<td>∞</td>
<td>0</td>
<td>55</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>∞</td>
<td>0</td>
<td>9</td>
<td>∞</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>4</td>
<td>22</td>
<td>43</td>
<td>∞</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

Fig. 5. (a) Matrix after deletion of row 1 and column 4. (b) First branching.

Equation (3) is used to calculate the lower bounds in Boxes 4, 5, and 10. That these lower bounds are valid is established by (2) and the non-negativity of the elements of \( C_i \).

For the example, the matrix of Fig. 5 shows the deletion of row 1 and column 4. The connected path containing \((1,4)\) is \((1,4)\) itself, so that \((m,p) = (4,1)\) and we set \(c(4,1) = m\). Looking for reductions, we find that row 2 can be reduced by 1. Then \(w(2) = 48 + 1 = 49\) as shown.

It may be worth giving another example of finding \((m,p)\). Suppose the committed city pairs were \((2,1)\), \((1,4)\), \((4,5)\), and \((5,1)\) were \((1,4)\). Then the connected path containing \((5,1)\) would start at 2 and end at 1 to yield \((m,p) = (3,2)\).

**Box 6** checks to see whether a single tour node is near.

**Box 7** selects the next node for the algorithm finished—whether:

*no choice is made.

The way shown here is to pick the node with the smallest lower bound. This leads to the fewest nodes in the tree.
of the uncommitted city pairs of \( y \).

\( u(4) \),

\( y + u(4) \)

(b)

out 4. (b) First branching.

ends in boxes 4, 5, and 10.

the deletion of row 1 and

is (1,4) itself, so that

reduction, we find that

= 49 as shown.

ending \((m,p)\). Suppose the

5,6) and \((4,4)\) were (1,4),

start at 4 and end at 3 to

is near.

here are a number of ways

e is to pick the node with

best nodes in the tree.

is finished—whether the

Fig. 6. Final tour.
best tour so far has a cost less than or equal to the lower bounds on all
terminal nodes of the tree.

Box 9 is a time saver. Most branching is from \( Y \) nodes, i.e., to the
right. Such branching involves crossing out rows and columns and other
manipulations that can be done on the matrix left over from the previous
branching. When this case occurs, Box 9 detects it and the algorithm
returns directly to Box 3.

Box 10 takes up the alternate case of setting up an appropriate lower
bound and reduced matrix for an arbitrary \( X \). Starting from the original
cost matrix, rows and columns are deleted for city pairs committed to the
tours of \( X \); infinites are placed to block subtours and at forbidden city
pairs, and the resulting matrix is reduced. The lower bound can be com-
puted from (3) by thinking of \( X \) in (3) as a starting node with \( w(X) = 0 \)
and matrix equal the original cost matrix. Some different ways of reducing
a matrix may lead to different sums for the reducing constants, the reas-
cribed \( w(X) \) is substituted for the former one.

Boxes 11 and 12 finish up a single tour node. By the time \( C \) is a \( 2 \times 2 \)
matrix, there are only two feasible \((x,y)\) left and they complete a tour.
Since the box is entered with a reduced matrix, the costs of the final two
 nitments are zero, and \( z = w(Y) \) by (2). If \( z < a \), the new tour is the best
yet and is read off the tree to be saved.

Returning to the example, Box 7 picks 14 as the second node for branch-
ing and, since this is a branching to the right, \( C \) is already available in re-
duced form. As shown in Fig. 6, the next branching is on the basis of (2.1)
with \((w,p) = (4,2)\). Next, we go to the right from 21 on the basis of (3.5)
with \((w,p) = (6,5)\) and then from 56 on the basis of (3.5) with \((w,p) = (6,3)\).
At this point \( C \) is a \( 2 \times 2 \) matrix, and we jump to Box 11 to finish the tour.
We find \( z = 63 \), which is stored as a cut, on returning to Boxes 7 and 8
we see that \( 14 \) has a lower bound of 28. To set up this node we go through
Box 10. After the next branching, however, Box 8 shows that the problem
is finished.

Discussion

At this point, let us stand back and review the general motivation of
the algorithm. It proceeds by branching, crossing off a row and column,
blocking a subtour, reducing the cost matrix to set a lower bound and then
repeating. Although it is clear that the optimal solution will eventually be
found, why should these particular steps be expected to be efficient? For
all of the, the reduction procedure is an efficient way of building up lower
bounds and also of evoking likely city pairs to put into the tour. Branching is
doing so to maximize the lower bound on the \( E \) node without worrying too
much about the \( I \) node. The reasoning here is that the \( I \) node represents
a smaller problem, putting the emphasis on optimal tours are re-
strictions into the crossing out of a

An Insight into the a

subtour creates a restricting the notation \( z \) into a single city, the
subtour to the

the restrictions into which

plashed rather once.

Finally, unlike \( R \), here, there is an external any stage be conven-
tstage. A trial branch is

investigated. For

how the next branch is

leading to the ultimate

there are just too many

likely to lead to a

A Variety of Substitution several that a

Calculations. For

Go to the Right

It is computationally

beach, known as the

node a
tour. As a
there will be sublist

of Box 10.

One consequence

of whether the

Threshold Above the

A large problem

difficulty of high spee
lower bounds on all
Y nodes, i.e., to the
of columns and other
ter from the previous
and the algorithm.

appropriate lower
vexity of the original
not be committed to the
ity; i.e., changes can be made with any N ≥ 0.
out ways of reducing
these, the second
the time C is a 2 X 2
completes a tour.
utes of the final tour
on the basis of (2.1)
the basis of (1.6)
11 to finish the tour.
to Route 7 and 8.
and the problem
al motivation of the
row and column,
our bound and then
it will eventually be
efficient? First
up lower bounds
we are primarily
about worrying about
11 a fiid a

A smaller problem, one with the 4th row and 5th column crossed out. By noting the emphasis on a large lower bound for the larger problem, non-optimal tours are ruled out faster.

Insight into the operation of the algorithm is gained by observing that the crossing out of a row and column and the blocking of the corresponding subset creates a new traveling salesman problem having one fewer city. Using the notation of Box 3, we can think of city v and city p as canceled out of city c, say, m. Setting (m,p) = 0 = p the same as setting
(m,m) = 0. The blocking of subsets is a way of introducing the four restrictions into what is otherwise an assigned problem and is accom-
plished rather successfully by the algorithm.

Finally, unlike most mathematical programming algorithms, the out-
puts has an extensive memory. It is not required that a trial solution at
any stage be converted into a new and better trial solution at the next
stage. A trial branch can be dropped for a moment while another branch
is investigated. For this reason there is considerable room for experiment
in how the next branch is chosen. On the other hand the same property
holds to the ultimate, demise of the computation—let a sufficiently large
are just too many branches to investigate and a small increase in N
likely to lead to a large number of new nodes that require investigation.

MODIFICATIONS

A variety of embellishments to the basic method can be proposed. We
would several that are incorporated in the computer program, used in later
calculations. For the program listing itself (see SWENBERT, 19).

Go to the Right

It is computationally advantageous to keep branching to the right until
it becomes obviously unwise. Specifically, the program always branches
the N + 1 node unless its lower bound exceeds or equals the cost of a
known tour. As a result a few extra nodes may be examined, but usually
there will be substantial reduction in the number of time-consuming steps of
Box 10.

One consequence of the modification is that the calculation goes di-
rectly to a tour at the beginning. That, if the calculations are stopped
before optimality is proven, a good tour is available. There is also available
a lower bound on the optimal tour. The bound may be valuable in
deciding whether the tour is sufficiently good for some practical purpose.

Throw Away the Tree

A large problem may involve thousands of nodes and exceed the ca-
pacity of high-speed storage. Storage can be saved, although usually at
the expense of time, by noting that, at any point in the computation, the cost of the best tour so far sets an upper bound on the cost of an optimal tour. Let the calculation proceed by branching to the right (storing each terminal node) until a single tour is found with some cost, say, $a$. Normally, one would next find the terminal node with the smallest lower bound and branch from there. Instead, work back through the terminal nodes, starting from the single tour, and discard nodes from storage until one is found with a lower bound less than $a$. Then, branch again to the right all the way to a single tour or until the lower bound on some right-hand node builds to $a$. (If the branch goes to the end, a better tour may be found and a assigned a new, lower value.) Repeat the procedure: again work up the branch, discarding terminal nodes with bounds equal or greater than $a$ until the first one smaller is found; again branch to the right, etc.

The effect of the procedure is that very few nodes need be kept in storage—something on the order of a few $n$. These form an orderly sequence stretching back from the current operating node directly back to the terminal node on the leftmost branch cut of all tours.

As an illustration, consider the problem and tree of Fig. 6. The computation would proceed by laying out in storage the nodes 4.1, 7.1, 3.6, and 3.5. At the next step we find a tour with $a = 63$ and the obviously useless nodes 4.3. The tour is stored separately from the tree. Working up the branch, first 5.5 is discarded, then 5.6 and 7.1, but 7.4 has a bound less than 63. Therefore, branching begins again from there. A node 6.3 is stored and then we find the node to the right has a bound equal to $a$ and may be discarded. Working back up the tree again, 5.3 is discarded and, since the only remaining terminal node, we are finished.

The procedure saves storage but sometimes increases computation time. If the first run to the right turns up a rather poor value, i.e., large $a$, then criterion for throwing away nodes is too stiff. The calculation is forced to branch out from many nodes whose lower bounds actually exceed the cost of the optimal tour. The original method would never do this for it would never explore such nodes until it had finished exploring every node with a smaller bound. In the process, the optimal tour would be uncovered and so the nodes with larger bounds would never be examined.

**Taking Advantage of Symmetry**

If the traveling salesman problem is symmetric and $i$ is any tour, another tour with the same cost is obtained by reversing the circuit in the reverse direction. Probably the most promising way to handle this is to treat the city pair $(ij)$ as not being ordered. This leads naturally to a new and somewhat more powerful bounding procedure. Although the basic ideas are not changed so far, we have not done it.

There is another way to simplify the computations by modifying the nodes so that the nodes with no city pairs such as node $K$ branches into $(K, J)$ forbidden. The reverse cannot be in $Y$ for the reverse tour. Each of the reverse tours is then obtain from setting $c((K, J)) = \infty$. Thus, a reverse tour is probable whenever having one cost

### Computational Aid

In both hand and machine first finding, for each row $i$ is $a(k) = \text{the cost of the $k$th row}$

Then $a(k) = \sum a(k) + \theta(k)$ for computation of the $(k)$ can be a subroutines and the $\theta(k)$ as an entire new problem, one can see that we need to search the whole matrix columns need be examined.

### Other Possibilities

If desired, the algorithm $a(k)$ solutions. Instead of discarding until eventually the entire single tours with $x = \infty$. Our modification because in some great deal—suppose the cost $p$

Quite possibly, the average assignment problem for the cost of the optimal tour, the assignment problem leave the larger lower bound to the columns that may be expected. It has been extensive and has yielded a great deal of improvement, but some others...
t in the computation, the
con cost of an optimal
to the right storing each
cost, say, $a$. For
with the smallest lower
seek through the terminal
the smallest branch to the
would end, a better tour may be
nodes from storage until
branch again to the
the root nodes.

not be kept in storage
an order sequence
it back to the terminal
of Fig. 6. The con-
the two nodes 4, 1; 3, 5, 8; and
and see obviously unless
so near. Working up the
but 1, 4 has a bound less
there. A node 6, 5 is
bound equal ($a$) and may
6, 5 is discarded and, since
re finished.
completes computation time.
not tour, i.e., large $a$, the
be calculation is forced to
actually exceed the cost
never do this to be would
node with a
would be uncovered and
examined.

with $i$ and $j$ is in any tour,
venting the circuit in the
way to handle this is to
This leads naturally to a
procedure. Although the

ideea are not changed much, considerable reprogramming is required.
Also, we have not done it.

There is another way to take advantage of symmetry and this one is
ty to incorporate into our program. All reverse tours can be prohibited
by modifying the nodes along the longest branch of the tree. These are
the nodes with no city pairs committed but some forbidden. Suppose that
such a node, $X$, branches into nodes, $Y$, with ($j$, $k$) committed, and, $Z$, with
($i$, $j$) forbidden. The reverse tours of $Y$ all have ($j$, $k$) in them. They
cannot be in $Y$ for the presence of both ($j$, $i$) and ($j$, $k$) is not possible in any
tour. Such of the reverse tours as were in $X$ are in $Y$. We may prohibit
them by setting $c(i, k) =$ as well as $c(j, k) =$ in any matrix for $Y$.
Thus, a reverse tour is prohibited as soon as the tour itself is identified to
the extent of having one committed city pair.

A Computational Aid
in both hand and machine computation $\theta(k, i)$ is easiest calculated by
by finding, for each row $k$ and column $i$ of the reduced matrix:

$\alpha(k) =$ the second smallest cost in row $k$.

$\theta(l) =$ the second smallest cost in column $l$.

Then $\theta(k, i) = a(k) + \theta(i)$ for any $(k, i)$ which has $c(k, i) =$ 0. In a hand
computation the $\alpha(k)$ can be written as an extra column to the right of the
matrix and the $\theta(l)$ as an extra row at the bottom. After working out a
few problems, one can see that when the branching is to the right there is no
need to match the whole matrix to reset $a$ and $i$, but that only certain rows
and columns need be examined.

Other Possibilities
If desired, the algorithm can be modified so as to generate all optimal
solutions. Instead of discounting nodes with $a(k, i) =$ 0, split them up further
and eventually all the terminal nodes either have $a(k, i) =$ 0 or are optimal
single tours with $a(k, i) =$ 0. Our computer program does not include this
modification because in some cases it will increase the computing time a
great deal—suppose the cost matrix were all zeros.

Quite possibly, the average computing time can be decreased by solving
the assignment problem for the original cost matrix and reducing the matrix
by the cost of the optimal assignment in Box 2. (Some methods for solving
the assignment problem leave it in reduced form.) The advantage lies in
the large lower bound with which the problem starts. The closer the
starting lower bound to the cost of the optimal tour, the less is the branch-
ing that may be expected. Our experience of the possible gains has not
been extensive and has yielded mixed results: Crome's 20-city problem was
speeded up, but some others were lengthened.
The idea that we are calling "branch and bound" is more general than the traveling salesman algorithm. A minimal solution for a problem can be found by taking the set of all feasible solutions, splitting it up into disjoint subsets, finding lower bounds on the objective function for each subset, splitting again the subset with the smallest lower bound, and so forth, until an optimal solution is found. The efficiency of the process, however, rests very strongly on the devices used to split the subsets and to find the lower bounds. As a simple example of another use of the method, if the step of setting $v(x,p) = 0$ is omitted from the traveling salesman algorithm, it solves the assignment problem. For another example, see DODD AND LAND.²⁸

<table>
<thead>
<tr>
<th>Number of cities</th>
<th>Number of problems solved</th>
<th>Mean $T$</th>
<th>Std. dev. $T$</th>
<th>Mean $\log T$</th>
<th>Std. dev. $\log T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>100</td>
<td>0.013</td>
<td>0.007</td>
<td>log 0.115</td>
<td>log 0.30</td>
</tr>
<tr>
<td>20</td>
<td>100</td>
<td>0.084</td>
<td>0.065</td>
<td>log 0.067</td>
<td>log 0.30</td>
</tr>
<tr>
<td>30</td>
<td>100</td>
<td>0.375</td>
<td>0.240</td>
<td>log 0.035</td>
<td>log 0.20</td>
</tr>
<tr>
<td>40</td>
<td>5</td>
<td>5.33</td>
<td>10.2</td>
<td>log 4.35</td>
<td>log 3.24</td>
</tr>
</tbody>
</table>

¹⁴ Obtained by plotting the cumulative frequency on log normal probability paper and fitting a straight line, except in the 40-city case for which the computation was reversed. In the case of 40 cities the log normal fits only the tail of the distribution—go per cent of the problems were directly to the solution without extra branching and thereby produced a large of probability at some volume.

CALCULATIONS

PROBLEMS UP to 10 cities can be solved easily by hand. Although we have made no special study of the time required for hard problems, our experience is that a 10-city problem can be solved in less than an hour.

The principal testing of the algorithm has been by machine on an IBM 7090. Two types of problems have been studied: (1) symmetric distance matrices with elements consisting of uniformly distributed 3-digit random numbers and (2) various published problems and subproblems constructed therefrom by deleting cities. Most of the published problems have been made up from road atlases or maps and are symmetric.

The random distance matrices have the advantage of being statistically well defined. Average computing times are displayed in Table I and curve (a) of Fig. 7. Problems up to 20 cities usually require only a few seconds. The time grows exponentially, however, and by 40 cities is beginning to be appreciable, averaging a little over 8 minutes. As a rule of thumb, adding 10 cities to the problem multiplies the time by a factor of 10.

Fig. 7. Comparing random number distri- bution Karp's 30-city 4 Karp's 40-city proble
ad bound is more general than the usual solution for a problem with solutions, splitting it up into disjoint sets. However, the objective function for each sub-
set is a lower bound, and so forth. We divide the process into finding the best way of using the method, if the traveling salesman algorithm, on similar examples, see [2, 19] and

<table>
<thead>
<tr>
<th>Random Distance Matrix (random Problem on IBM 7090)</th>
<th>Mean ( \log T )</th>
<th>Std. dev ( \log T )</th>
</tr>
</thead>
<tbody>
<tr>
<td>log 5.025</td>
<td>log 2.42</td>
<td></td>
</tr>
<tr>
<td>log 6.027</td>
<td>log 2.09</td>
<td></td>
</tr>
<tr>
<td>log 7.67</td>
<td>log 3.24</td>
<td></td>
</tr>
<tr>
<td>log 8.52</td>
<td>log 3.74</td>
<td></td>
</tr>
</tbody>
</table>

The standard deviation of these computing times also increases with problem size as may be seen in Table 1. Because the distribution of times is skew, the simple standard deviation is a little misleading, at least for the purpose of estimating the probability of a long calculation. Conse-

Fig. 7. Computing times on IBM 7090. (a) Average times for 3-digit random number distance matrices. (b) Subproblems derived from Held and Karp’s 28-city problem. (c) Subproblems derived from Held and Karp’s 28-city problem.

- The standard deviation of these computing times also increases with problem size as may be seen in Table 1. Because the distribution of times is skew, the simple standard deviation is a little misleading, at least for the purpose of estimating the probability of a long calculation. Consequent-
uly by hand. Although we have done hand computations, our ex-
solved in less than an hour.
avez been by machines on an IBM 7090 studied: (1) asymmetric distance formulism we have distributed 3-digit random numbers and subproblems constructed on published problems have been tested symmetric.

As advantage of being statistically significant is displayed in Table 1 and curve usually require only a few seconds and by 40 cities is beginning to be

As a rule of thumb, adding me by a factor of 10.
quently, a log normal distribution has been fitted to the tail of the distribution. A use of the tabulated numbers would be, for example, as follows: A two-standard deviation on the high side in a 40-city problem would be a calculation that took (3.74)\(^4\) = 64 minutes. In other words, the probability that a 40-city random distance problem will require 64 minutes or more is estimated to be 0.023.

Symmetrical problems have usually taken considerably longer than random distance problems of the same size. To obtain a variety of problems of increasing size, we have taken published problems and abstracted subproblems of increasing size. The first 10 cities were taken, then the first 11 cities, etc., until the computing time became excessive. Cases (b) and (c) of Fig. 7 show the results for subproblems pulled out of the 25- and 48-city problems of Held and Karp. The 25-city problem took 4.7 minutes. We note that Held and Karp's conjectured optimal solution is correct.

A few miscellaneous problems have also been solved. Croes' 20-city problem took 0.126 minutes. A 64 candy knight's tour took 0.178 minutes.

ACKNOWLEDGMENT

The computing time used in the early stages of the work was provided by the Case Institute of Technology Computing Center, the M. I. T. Computer Center, and the M. I. T. School of Industrial Management Computing Facility. The time for the production runs was provided by I. B. M. on the IBM 7090 at the M. I. T. Computation Center. We wish to acknowledge the help of R. H. Gonzalez for programming certain investigations. His time was supported by the U. S. Army Research Office (Durham).

REFERENCES

Traveling Salesman Problem


The tail of the dis- would be, for example, as side in a 40-city problem (35) = 64 minutes. In other instance problem will require considerably longer than

To obtain a variety of prob-

problems and abstracted

cities were taken, then the

came excessive. Curves

problem pulled out of the

The 25-city problem itself

Karp's conjectured optimal

en solved. Croom's 20-city

hit's tour took 0.178 minutes.

of the work was provided by

the M. I. T. Compu-

tational Management Computing

as provided by I. B. M. on the

c. We wish to acknowledg-

g certain investigations. His-

cher Office (Durham).

ramming Approach to Sequencing


A Solution to the Traveling

ming," micrographed.

orial Programming," presented

micrographed.

Pattern Constraints," Ph. D. dis-

n, in augmented form: Report

vard University, July 1956.