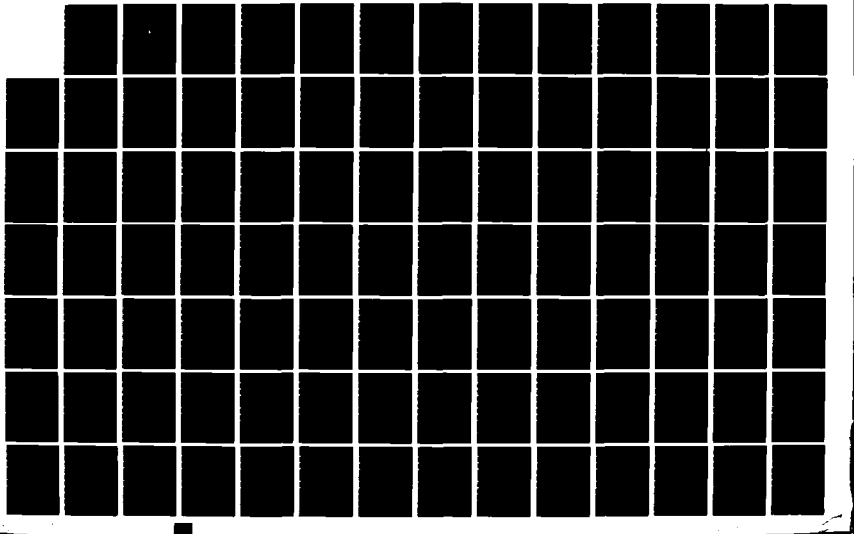
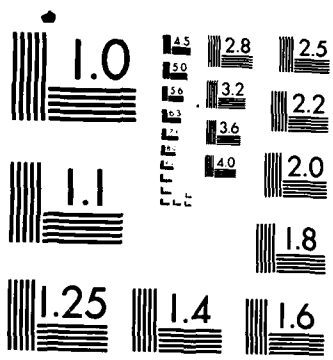
AD-A162 118 ALGORITHMS AND HEURISTICS FOR TIME-WINDOW-CONSTRAINED TRAVELING SALESMAN PROBLEMS(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA B J CHUN ET AL SEP 85 1/2

UNCLASSIFIED F/G 12/1 NL





(2)

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

ALGORITHMS AND HEURISTICS FOR
TIME-WINDOW-CONSTRAINED
TRAVELING SALESMAN PROBLEMS

by

Chun, Bock Jin
and
Lee, Sang Heon

September 1985

Thesis Advisor: Richard E. Rosenthal

Approved for public release; distribution is unlimited.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD A162 114	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Algorithms and Heuristics for Time-Window-Constrained Traveling Salesman Problems		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis; September 1985
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Chun, Bock Jin Lee, Sang Heon		8. CONTRACT OR GRANT NUMBER(s) -
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943-5100		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943-5100		12. REPORT DATE September 1985
		13. NUMBER OF PAGES 103
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
Heuristic, Algorithm, Time Window, Hard Time Window, Soft Time Window, Slack, Branch and Bound, Nearest Neighbor, Penalty Cost, Traveling Salesman Problem, State-Space Relaxation

20. ABSTRACT (Continue on reverse side if necessary and identify by block number) This thesis reports on methods for solving traveling salesman problems with time-window constraints. Two types of time windows are considered: hard time windows, which are inviolable, and soft time windows, which are violable at a cost. For both cases, we develop several heuristic procedures, including some that are based on Stewart's [Ref. 6] effective heuristics for the traveling salesman problem without time-window constraints. In addition, we develop exact algorithms for each case, which are based on the state-space relaxation dynamic programming method of Christofides, Mingozzi, and Toth [Ref. 5]. Computational experience is reported for all the heuristics and algorithms we develop.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Approved for public release; distribution is unlimited.

Algorithms and Heuristics for Time-Window-Constrained Traveling Salesman Problems

by

Chun, Bock Jin
Major, Republic of Korea Air Force
B.S., Korea Air Force Academy, 1976
and
Lee, Sang Heon
Major, Republic of Korea Army
B.S., Korea Military Academy, 1977

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN OPERATIONS RESEARCH

from the

NAVAL POSTGRADUATE SCHOOL
September 1985

Authors: Bock Jin Chun
Chun, Bock Jin

Sang Heon Lee
Lee, Sang Heon

Approved by: Richard E. Rosenthal
Richard E. Rosenthal, Thesis Advisor

Alan R. Washburn
Alan R. Washburn, Second Reader

Alan R. Washburn
Alan R. Washburn, Chairman,
Department of Operations Research

K. T. Marshall
Kneale T. Marshall,
Dean of Information and Policy Sciences

ABSTRACT

This thesis reports on methods for solving traveling salesman problems with time-window constraints. Two types of time windows are considered: hard time windows, which are inviolable, and soft time windows, which are violable at a cost. For both cases, we develop several heuristic procedures, including some that are based on Stewart's [Ref.6] effective heuristics for the traveling salesman problem without time-window constraints. In addition, we develop exact algorithms for each case, which are based on the state-space relaxation dynamic programming method of Christofides, Mingozzi, and Toth [Ref.5]. Computational experience is reported for all the heuristics and algorithms we develop.

TABLE OF CONTENTS

I. INTRODUCTION 9
A. OVERVIEW 9
B. THE TRAVELING SALESMAN PROBLEM 11
C. TSP WITH TIME WINDOW CONSTRAINTS 13

II. HEURISTIC TSP SOLUTION 16
A. OVERVIEW 16
1. Tour Construction Procedures 16
2. Tour Improvement Procedures 21
3. Composite Procedure 23
B. CCAO 23
1. Algorithm 23
2. Example 24
3. Computational Results 27

III. THE TSP WITH HARD TIME WINDOW CONSTRAINTS 32
A. INTRODUCTION 32
B. HEURISTIC SOLUTION TECHNIQUES FOR HARD TIME WINDOWS 33
1. Nearest Neighbor 33
2. SCCO 35
3. SCAO 42
4. SLACK 43
C. EXACT SOLUTION TECHNIQUES FOR HARD TIME WINDOWS 46
1. State-Space Relaxation Procedure 46
2. Additional Condition 50
3. Branch and Bound Procedure 53

IV. THE TSP WITH SOFT TIME WINDOW CONSTRAINTS 57
A. INTRODUCTION 57
B. HEURISTIC SOLUTION TECHNIQUES FOR SOFT TIME WINDOWS 58
1. Nearest Neighbor 58
2. SCCO 59
3. SCAO 61
C. EXACT SOLUTION TECHNIQUES FOR SOFT TIME WINDOWS 62
1. State-Space Relaxation Procedure 62
2. Additional Condition 66
3. Branch and Bound Procedure 69

V. COMPUTATIONAL EXPERIENCE 70
A. TEST PROBLEMS 70
B. COMPUTATIONAL RESULTS 73
1. Hard Time Windows 73
2. Soft Time Windows 75

VI. CONCLUSIONS AND RECOMMENDATIONS 77

APPENDIX A: TEST PROBLEM [1] 79

APPENDIX B: TEST PROBLEM [2] 80

APPENDIX C: TEST PROBLEM [3] 81

APPENDIX D: TEST PROBLEM [5] 82

APPENDIX E: TEST PROBLEM [6] 83

APPENDIX F: TEST PROBLEM FOR THE SCCO 84

APPENDIX G: TEST PROBLEM [1-1] 85

APPENDIX H: TEST PROBLEM [1-2] 86

APPENDIX I: TEST PROBLEM [1-3] 87

APPENDIX J: TEST PROBLEM [1-4] 88

APPENDIX K: TEST PROBLEM [2-1] 89

APPENDIX L: TEST PROBLEM [2-2] 90

APPENDIX M: TEST PROBLEM [2-3] 91

APPENDIX N: TEST PROBLEM [2-4] 92

APPENDIX O: TEST PROBLEM [3-1] 93

APPENDIX P: TEST PROBLEM [3-2] 94

APPENDIX Q: TEST PROBLEM [3-3] 95

APPENDIX R: TEST PROBLEM [3-4] 96

APPENDIX S: TEST PROBLEM [4-1] 97

APPENDIX T: TEST PROBLEM [4-2] 98

APPENDIX U: TEST PROBLEM [4-3] 99

APPENDIX V: TEST PROBLEM [4-4] 100

LIST OF REFERENCES 101

INITIAL DISTRIBUTION LIST 103

LIST OF TABLES

I	COMPUTATIONAL RESULTS OF CCCO, CCAO	31
II	COMPUTATIONAL RESULTS OF THE HARD TIME WINDOWS	74
III	COMPUTATIONAL RESULTS OF THE SOFT TIME WINDOWS	76

LIST OF FIGURES

1.1 Example of Nonconvexity of (1.10) in Two Dimensions 15
2.1 Concept of the Clarke - Wright Savings Heuristic 18
2.2 Initial Subtour and Insertion 25
2.3 Intermediate Subtour and Insertions 26
2.4 Final Tour of CCAO 27
3.1 Diagram for Hard Time Window Case 32
3.2 Current Tour before Modified-Oropt 37
3.3 Improved Tour after Modified-Oropt 38
3.4 Subtour in SCCO Procedure 39
3.5 Intermediate Subtour in SCCO Procedure 40
3.6 Final Subtour for SCCO 41
3.7 Optimal Route of Four Nodes Problem 52
4.1 Diagram for Soft Time Window Case 58
5.1 Unconstrained Solution Obtained by CCAO 71
5.2 Unconstrained Solution Obtained by Nearest Neighbor Heuristic 72

I. INTRODUCTION

A. OVERVIEW

Consider a traveling salesman having to visit n cities or customers. He starts from a depot and needs to visit each of the other n-1 cities only once and then return to the depot. The cost of traveling between any pair of cities (expressed in terms of distance, time or cost, etc), say from city i to j, is given as c_{ij} in a cost matrix C. The problem is to design a tour through the n cities that would minimize the total cost of the tour. This is known as the Traveling Salesman Problem which is a well-known classical operations research problem.

The TSP is called Euclidean when the cities that must be visited all lie on the same plane and the cost of traveling between any pair of cities is the Euclidean distance between them.

The TSP is an NP-complete problem [Ref. 1, 2]. All known exact solution methods have a rate of growth of the computation time which is exponential in n. On the other hand, heuristic solution methods have a rate of growth of the computation time which is a low order polynomial in n and have been experimentally observed to perform well. For this reason, there has been an extensive amount of research directed at TSP heuristics.

In this thesis we consider adding time window constraints to the TSP. That is, if t_i is the time that the salesman visits city i, then t_i must satisfy $l_i \leq t_i \leq u_i$, where l_i and u_i are the specified lower and upper bounds of a time window. This problem is not as well studied as the unconstrainted TSP, but there have been a few approaches used on the problem.

Psaraftis [Ref. 3] has presented a dynamic programming model and solution procedure for two dial-a-ride problems, which are similar to time-window constrained TSPs. Baker [Ref. 4] has presented an exact algorithm using a branch and bound procedure which is effective for very small n. Christofides et al. [Ref. 5] have presented a dynamic programming state space relaxation procedure to compute bounding information within a branch and bound algorithm.

The objective of this study is to develop exact and heuristic algorithms which will provide an optimal or near optimal tour that visits each city in its given time window. We are given a depot location, a set of x,y co-ordinates for n cities and a set of time windows .

A common application of the TSP is in vehicle routing problems. A set of customer orders must be partitioned among several vehicles. Given a partition, the problem then decomposes into one TSP for each vehicle. Because of this prospective application and in deference to the difficulty of large TSPs (with or without time constraints), we confine our research and computation to small-scaled problems (less than 30 nodes).

We consider two different kinds of time window constraints: hard time windows and soft time windows. Hard windows cannot be violated. Soft windows can be violated, but a penalty cost must be paid when they are. The penalties can be defined individually for each customer, and they can differ for early and late arrivals. Generally, the penalty for arriving before the lower time window bound is much less than the penalty for arriving after the upper bound. In Chapter III, we present the hard time window approach and in Chapter IV, we present the soft time window approach.

We developed several Fortran programs for solving the TSP and time-constrained TSP. For the TSP, we use Stewart's [Ref. 6] recent heuristics, CCCO and CCAO. For the time

constrained TSP problems, we develop some new heuristics, some of which are modification of Stewart's heuristics for the unconstrained problem. We also developed exact algorithms for both hard and soft windows using Christofides et al.'s [Ref. 5] method of state-space-relaxation dynamic programming and branch and bound. This is described in Chapters III and IV.

Finally, we describe a hybrid of the heuristic and exact programs. The hybrid uses the overall structure of the exact program, but the upper bounds are obtained with the heuristic. This is discussed in Chapters III and IV.

B. THE TRAVELING SALESMAN PROBLEM

A tour is a chain which passes through all the n nodes and in which the first and the last nodes coincide. A tour is also known as a Hamiltonian cycle.

Let a tour be denoted by $t = (i_1, i_2, \ldots, i_n, i_1)$ and the cost of this tour be

$$C(t) = \sum_{j=1}^{n-1} c_{i_j i_{j+1}} + c_{i_n i_1}.$$

Here $(i_1, i_2, \ldots, i_n)$ is a permutation of the integers from 1 to n, giving the order in which the cities are visited.

The Traveling Salesman Problem can be defined as follows. Given a graph $G = \{ N, A \}$ composed of a set of nodes N, a set of arcs A connecting these nodes, and a cost (distance) c_{ij} associated with each arc (i,j) in A. The TSP is the problem of finding the minimum cost tour of the nodes in N. The following mathematical formulation of the TSP is from Stewart [Ref. 6].

$$\text{MIN} \sum_{i,j} c_{ij} x_{ij} \qquad (1.1)$$

S.T

$$\sum_{i} x_{ij} = 1 \qquad j = 1,\ldots,n \qquad (1.2)$$

$$\sum_{j} x_{ij} = 1 \qquad i = 1,\ldots,n \qquad (1.3)$$

$$\sum_{\substack{j=1 \\ j\neq i}}^{n} y_{ij} - \sum_{\substack{j=2 \\ j\neq i}}^{n} y_{ji} = 1 \qquad i = 1,\ldots,n \qquad (1.4)$$

$$y_{ij} - u\, x_{ij} \leq 0 \qquad i = 2,\ldots,n \qquad (1.5)$$

$$j = 1,\ldots,n$$

$$i \neq j$$

$$x_{ij} = 0, 1 \quad \text{for all} \quad (i,j) \qquad (1.6)$$

$$y_{ij} \geq 0 \quad \text{for all} \quad (i,j) \qquad (1,7)$$

where

$x_{ij} = \begin{cases} 1 & \text{if arc}(i,j) \text{ is on the tour} \\ 0 & \text{otherwise} \end{cases}$

y_{ij} are continuous variables that force the final solution to be on the tour (i.e. include every node in the same route)

u is a number $\geq n-1$, and

n is the number of nodes in the set N.

The constraints (1.2) and (1.3) ensure that each node will be visited exactly once, while constraints (1.4) and (1.5) force the final solution to be a single tour that starts and ends at node 1 (depot). This formulation is not directly used in our TSP programs, but is of interest in a general discussion of the problem.

C. TSP WITH TIME WINDOW CONSTRAINTS

The time-constrained Traveling Salesman Problem is a variation of the TSP that includes time window constraints on the time to visit some of the cities. The hard time-constrained TSP is to find the minimum cost tour subject to visiting each city within its time window.

For the time-constrained TSP model, we define a continuous nonnegative variable, t_i, to be the time that the salesman visits city i. Since the salesman must return to city 1 (depot) at the end of the tour, the formulation includes an additional variable, t_{n+1}, the total time required to complete the tour.

We assume that a complete, symmetric, nonnegative distance matrix, $|C_{ij}|$, is known and that time is a scalar transformation of distance. Thus time and distance may be used interchangeably.

The following mathematical formulation of the TSP with time constraints is from Baker [Ref. 4].

$$\text{MIN} \quad t_{n+1} - t_1 \tag{1.8}$$

$$\text{S.T} \quad t_i - t_1 \geq c_{1i} \qquad i = 2,3,\ldots,n \tag{1.9}$$

$$|t_i - t_j| \geq c_{ij} \qquad j = 3,4,\ldots,n \tag{1.10}$$

$$2 \leq i < j$$

$$t_{n+1} - t_i \geq c_{1i} \qquad i = 2,3,\ldots,n \tag{1.11}$$

$$t_i \geq 0 \qquad i = 1,2,\ldots,n+1 \tag{1.12}$$

$$l_i \leq t_i \leq u_i \qquad i = 2,3,\ldots,n \tag{1.13}$$

where t_i = the time that the salesman visits city i

$|x|$ = the absolute value of x

c_{ij} = the shortest time required to travel from city i to city j

l_i = the lower bound on the time window for the salesman to visit city i
by assumption all $l_i \geq 0$

u_i = the upper bound on the time window for the salesman to visit city i
$u_i \geq l_i$, for all i

The constraints (1.9) through (1.12) ensure a nonnegative arrival time at city i, t_i, be obtained for each city (node 2 through node n) on the tour. The constraint (1.9) guarantees that t_1, the time that the salesman leaves the node 1 (depot) will be the smallest t value. The absolute value constraint (1.10) ensures that the arrival times of any two city differ by amount of time sufficient to allow the salesman to travel between the two cities. The constraint (1.11) guarantees that t_{n+1}, the time the salesman returns to the depot, will be the largest t value. The inequalities (1.12) and (1.13) are nonnegativity and the time window constraints respectively.

Unfortunately, Baker's proposed model for the time-constrained TSP is very difficult to solve, because constraint (1.10) is nonconvex. Therefore, we will not use this formulation directly in cur program.

Figure 1.1 illustrates the nonconvexity of constraints (1.10) for one i,j pair, the example $| t_3 - t_4 | \geq 5$. The feasible region for this constraint is the union of two disjoint sets. Taken all together, constraints (1.10) define 2^m disjoint sets sets where $m = (n-1)(n-2)/2$, which are very difficult to work with.

We can see that the time-constrained TSP is very different from TSP, even in formulation.

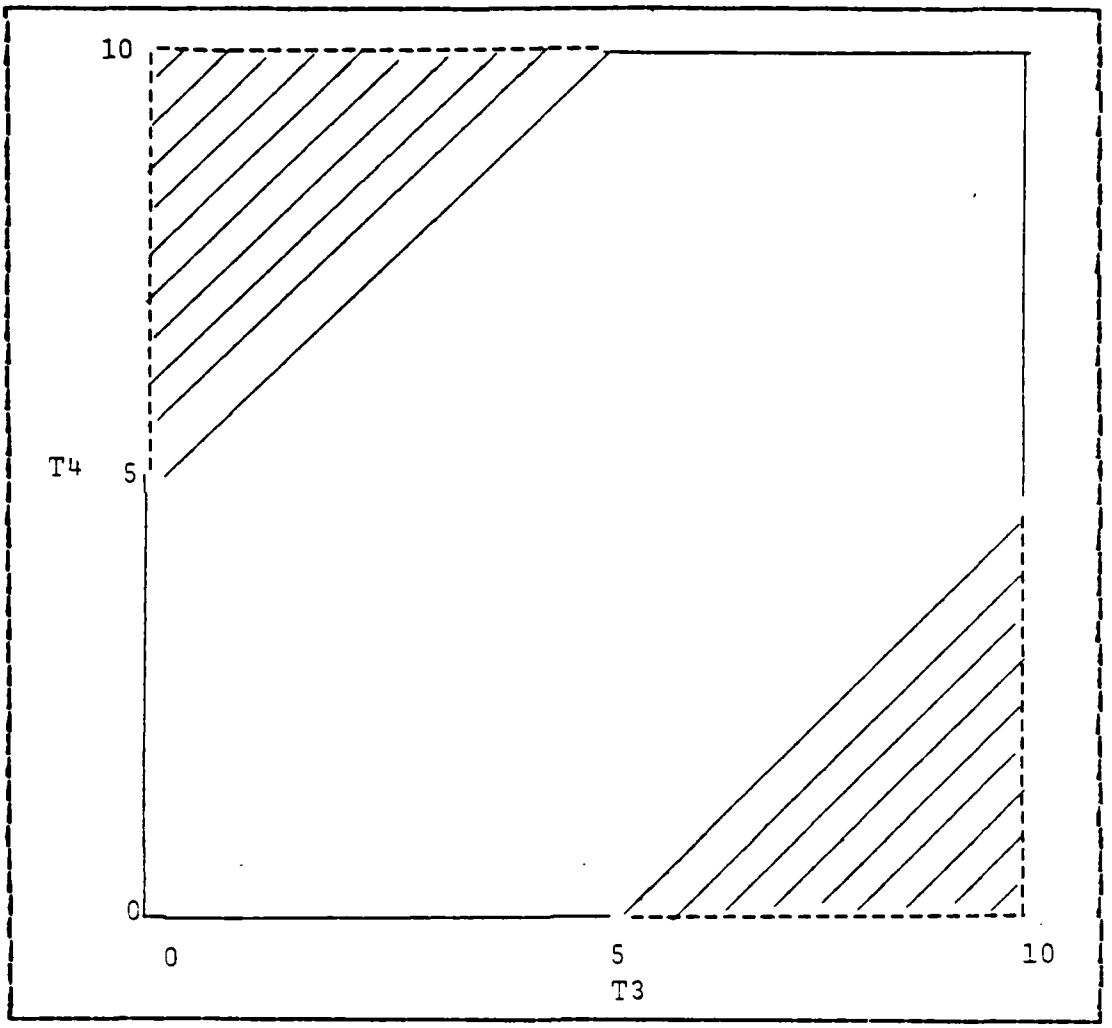


Figure 1.1 Example of Nonconvexity of (1.10) in Two Dimensions.

II. HEURISTIC TSP SOLUTION

A. OVERVIEW

Many heuristic procedures have been developed for solving TSP. Our purposes in this Chapter are to examine some of the well-known heuristics, to review Stewart's [Ref. 6] recent heuristic, and to compare these approximate techniques on the basis of efficiency and accuracy on a small number of examples.

In general, heuristic procedures are categorized by three broad classes: tour construction procedures, tour improvement procedures, and composite procedures [Ref. 7]. Tour construction procedures start with a single node and successively add nodes till a tour is built. Tour improvement procedures attempt to find a better tour given an initial tour. Composite procedures construct a starting tour from one of the tour construction procedures and then try to find a better tour using one or more of the tour improvement procedures.

1. Tour Construction Procedures

There are many methods available for constructing an initial tour. Procedures which have been generally used are given below.

a. Nearest Neighbor (Rosenkrantz et al. [Ref. 8])

Step 1. Start with any node as the beginning of a subtour.

Step 2. Find the node closest to the last node added to the subtour. Add this node to the current subtour.

Step 3. Repeat step 2 until all nodes are contained in the tour. Then, join the first and last node.

b. Clarke and Wright Savings (Clarke and Wright [Ref. 9])

Step 1. Select any node as the central depot which we dencte as node 1.

Step 2. Compute savings $s_{ij} = c_{1i} + c_{1j} - c_{ij}$

for $i,j = 2,3,\ldots,n.\ i \neq j$

Step 3. Order the savings from largest to smallest.

Step 4. Starting with the largest savings on the list, subtours are assembled such that the next node added has the largest remaining savings - provided that a constraint is not violated. Once a pair of nodes i and j have been linked, they remained linked.
Repeat until all nodes have been assigned.

Here, the quantity s_{ij} is the amount of travel saved if node j is visited directly after i, as opposed to having separate trips from the depot to nodes i and j. Figure 2.1 demonstrates the procedure for two nodes i and j.

c. Insertion Procedures (Rosenkrantz et al. [Ref. 8])

An insertion algorithm constructs a feasible tour by successively adding one node to an existing subtour. This procedure takes a subtour of k nodes at iteration k and attempts to determine which node not in the subtour should join the subtour next (the Selection step). Then it determines where in the subtour it should be inserted (Insertion

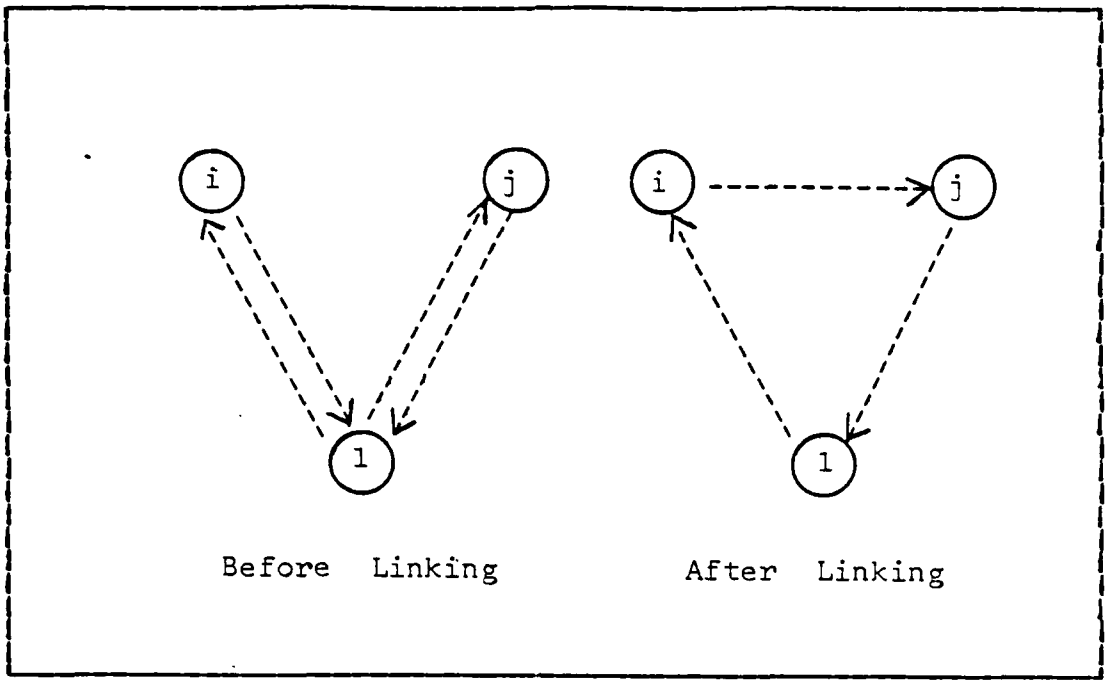


Figure 2.1 Concept of the Clarke - Wright Savings Heuristic.

step). Stewart [Ref. 6] presented the following general algorithmic structure.

Step 1. (Initial Subtour)
Obtain a TSP tour for a subset of the nodes $N' \subset N$ in G.

Step 2. (Selection Step)
Find a node $k \in N-N'$ to be added to the existing subtour.

Step 3. (Insertion Step)
Choose an arc(i,j) in the subtour on N'.
Insert node k between i and j and add k to N'.

Step 4. If N = N', then stop
(We have a Hamiltonian cycle).
Otherwise, return to step 2.

There are many variations on this algorithmic structure depending on the procedures chosen for executing steps 1,2 and 3.

Wiorkowski and McElvain [Ref. 10], Or [Ref. 11], Stewart [Ref. 12] and Norback and Love [Ref. 13] all present insertion algorithms that use the convex hull of the set of nodes N for the initial subset N'. Nemhauser and Hardgrave [Ref. 14] have shown that there exists an optimal tour for every Euclidean TSP in which the relative order of the nodes on the boundary of the convex hull is preserved. This means that the optimal tour visits nodes on the boundary of the convex hull in the same order as if the boundary itself were followed.

Further justification for the use of the convex hull for the initial subtour is shown empirically by Stewart's [Ref. 6] computational experiment. He compared several insertion heuristics both with and without the convex hull as the starting solution. The results show that all the insertion algorithms are improved by the use of the convex hull. Some are improved substantially, others only moderately.

Many criteria have been suggested for the selection of the node to be inserted in an insertion procedure.

(1) Nearest Neighbor (Rosenkrantz et al. [Ref. 8]). Choose the node k that is nearest a node in the current tour. I. e. , find $k = \underset{j}{\text{argmin}}\ c_{ij}$ s.t. $j \in N-N'$, $i \in N'$.

(2) Cheapest Insertion (Rosenkrantz et al. [Ref. 8]). Choose the node k that may be inserted at minimal increased cost. I.e., find
$k = \underset{m}{\text{argmin}}\ c_{im} + c_{mj} - c_{ij}$ s.t. $m \in N-N'$, $i,j \in N'$.

(3) Farthest Insertion (Rosenkrantz et al. [Ref. 8]). Choose the node k that is farthest from a node current subtour. I.e., find $k = \underset{j}{\text{argmax}}\ c_{ij}$ s.t. $j \in N-N'$, $i \in N'$

(4) Arbitrary Insertion (Rosenkrantz et al. [Ref. 8]). Choose node k randomly from among N-N'.

(5) Ratio Insertion (Stewart [Ref. 12]). Choose the node k such that the proportional increase in cost is minimal. I.e., find $k = \underset{m}{\text{argmin}}\ (c_{im} + c_{mj}) / c_{ij}$ s.t. $m \in N-N'$, $i,j \in N'$.

(6) Perpendicular Distance (Wiorkowski and McElvain [Ref. 10]). Choose the node k that is closest to an arc in the current subtour.

(7) Ratio Times Distance (Or [Ref. 11]). Choose the node k such that the product of ratio and distance is minimized. I. e. , find

$k = \underset{m}{\text{argmin}}\ ((c_{im} + c_{mj}) / c_{ij}) \times (c_{im} + c_{mj} - c_{ij})$

s.t. $m \in N-N'$, $i,j \in N'$.

(8) Greatest Angle (Norback and Love [Ref. 13]). Choose the node k and arc i, j such that the angle formed by the two arcs (i,k) and (k,j) is a maximum. I.e., find $k = \underset{m}{\text{argmax}}\ \text{angle}\{ \text{arc}(i,m), \text{arc}(m,j) \}$ s.t. $m \in N-N'$, $i,j \in N'$.

The insertion criteria that have been used fall into two categories. [Ref. 6]

1. Cheapest Insertion

Insert the node $k \in N-N'$ between those two connected nodes $i,j \in N'$ that minimize the quantity $c_{ik} + c_{kj} - c_{ij}$.

2. Identical Insertion and Selection

Do selection and insertion in the same step.

2. Tour Improvement Procedures

The best known procedures of this type for the TSP are the branch exchange heuristics [Ref. 7]. These branch exchange heuristics work as follows.

Step 1. Find an initial tour. This tour may be chosen randomly from the set of all possible tours, or it may be generated by one of the tour building procedures above.

Step 2. Improve the tour using one of the branch exchange heuristics.

Step 3. Continue step 2, until no additional improvement can be made.

For a given k, we define a k-change of a tour as consisting of the deletion of k branches in a tour and their replacement by k other branches to form a new tour. A tour is k-opt if it is not possible to improve the tour via a k-change. In general the larger the value of k, the more likely it is that a k-opt solution will be optimal. Unfortunately, the number of operations necessary to test all k exchange is proportional to n^k, where n is the number of nodes in the TSP. Due to this complexity, values of k = 2 and k = 3 are most commonly used [Ref. 7]. The 2-opt and 3-opt heuristics were introduced by Lin [Ref. 15] and the k-opt procedure, for $k \geq 3$ was presented by Lin and Kernighan [Ref. 16].

Or [Ref. 11] has designed a modified 3-opt that considers only a small percentage of 3-branch exchanges. This modified 3-opt called Oropt by Stewart [Ref. 6]

considers only those branch exchanges which are composed of a string of one, two, or three adjacent nodes being inserted between two other nodes in the current tour. By limiting the number of exchanges that are considered in this way, Oropt requires many fewer calculations than a full 3-opt.

Stewart [Ref. 6] made an experiment of the convex hull, cheapest angle insertion algorithm (CCA) which will be discussed in the next section as a stand-alone algorithm and with each of the three post-processors. The algorithms are designated CCA, CCA2, CCAO, and CCA3 for the convex hull cheapest insertion stand-alone, with 2-opt, with Oropt and with 3-opt respectively. He drew two conclusions from his computational results. First, the 3-opt requires substantially more time than either the 2-opt or the Oropt. Second, the 2-opt is dominated by the Oropt and the 3-opt in quality of solution.

In computation time, Oropt only looks at approximately $3n^2$ of the n possible 3-opt exchanges on each pass. There are n ways to select the first branch, times 3 ways to select the second branch, and n-2 ways to select the third branch.

This accounts for the fairly close times for the 2-opt and Oropt. The quality of CCAO solutions dominate CCA2 solutions. On the other hand, there is little or no difference between the Oropt and 3-opt in terms of solution quality.

Stewart's main conclusion from the above experiment is that the Oropt performs as well as a 3-opt in a small percentage of the computer time required by a 3-opt, and it should be preferred to both the 2-opt and the 3-opt for Euclidean TSP's.

3. Composite Procedure

The basic composite procedure is a combination of the tour construction and branch exchange procedures. It is obtained by appending a branch exchange procedure to the tour construction algorithm as a post-processor. The procedure can be stated as follows [Ref. 17].

Step 1. Obtain an initial tour using one of the tour construction procedures.

Step 2. Apply a branch exchange procedure to the solution produced by the step 1. Stop when no further improvement can be made.

The composite procedure is relatively fast computationally and gives good results [Ref. 18].

B. CCAO

1. Algorithm

The CCAO algorithm designed by Stewart [Ref. 6] uses the convex hull of the nodes in N for its initial subtour. Then it inserts the nodes not currently in the subtour where they may be inserted most cheaply (the Cheapest Insertion criterion). It selects the node k to be inserted at each iteration according to how large an angle is formed by the two arcs that must be added to the current subtour (Selection criterion) in order to insert k. Finally it uses an Oropt to make local improvement on the tour constructed in the first stage. CCAO means Convex Hull, Cheapest Insertion, Angle Selection, Oropt.

Algorithm : CCAO

Input : Number of nodes, x and y co-ordinates of all nodes.

Output: Ordered list of tour, total cost.

Step 1 . (Initial Subtour)

Find the convex hull of the set of nodes N. Call the set of nodes on the boundary N'. Let the initial subtour be the nodes of N' in the same order as they appear on the convex hull.

Step 2 . (Cheapest Insertion)

For each node $m \in N-N'$, find

$$(i_m, j_m) = \underset{i,j}{\text{argmin}} \; c_{im} + c_{mj} - c_{ij}$$

s.t. $i, j \in N'$, i, j : connected.

Step 3 . (Greatest Angle Selection)

For the next insertion, select the node that maximizes the angle between the arcs (i_m, m) and (m, j_m) over all $m \in N-N'$.

I.e, find $k = \underset{m}{\text{argmax}} \; \text{angle}\{ (i_m, m), (m, j_m) \}$

s.t. $m \in N-N'$.

Insert k between i_k and j_k and add k to N'.

step 4 : If N' = N, go to step 5.
Otherwise return to step 2.

step 5 : Apply an Oropt to the current tour. Stop when no further improvements can be found.

End of algorithm CCAO

2. Example

Figures 2.2 - 2.4 illustrate the above algorithm on the TSP defined as test problem [1] in Appendix A. First the convex hull is generated for an initial starting subtour. This subtour consists of nodes 2,13,12,14,5,15,7,4. A solid line marks the boundary of the convex hull in Figure 2.2.

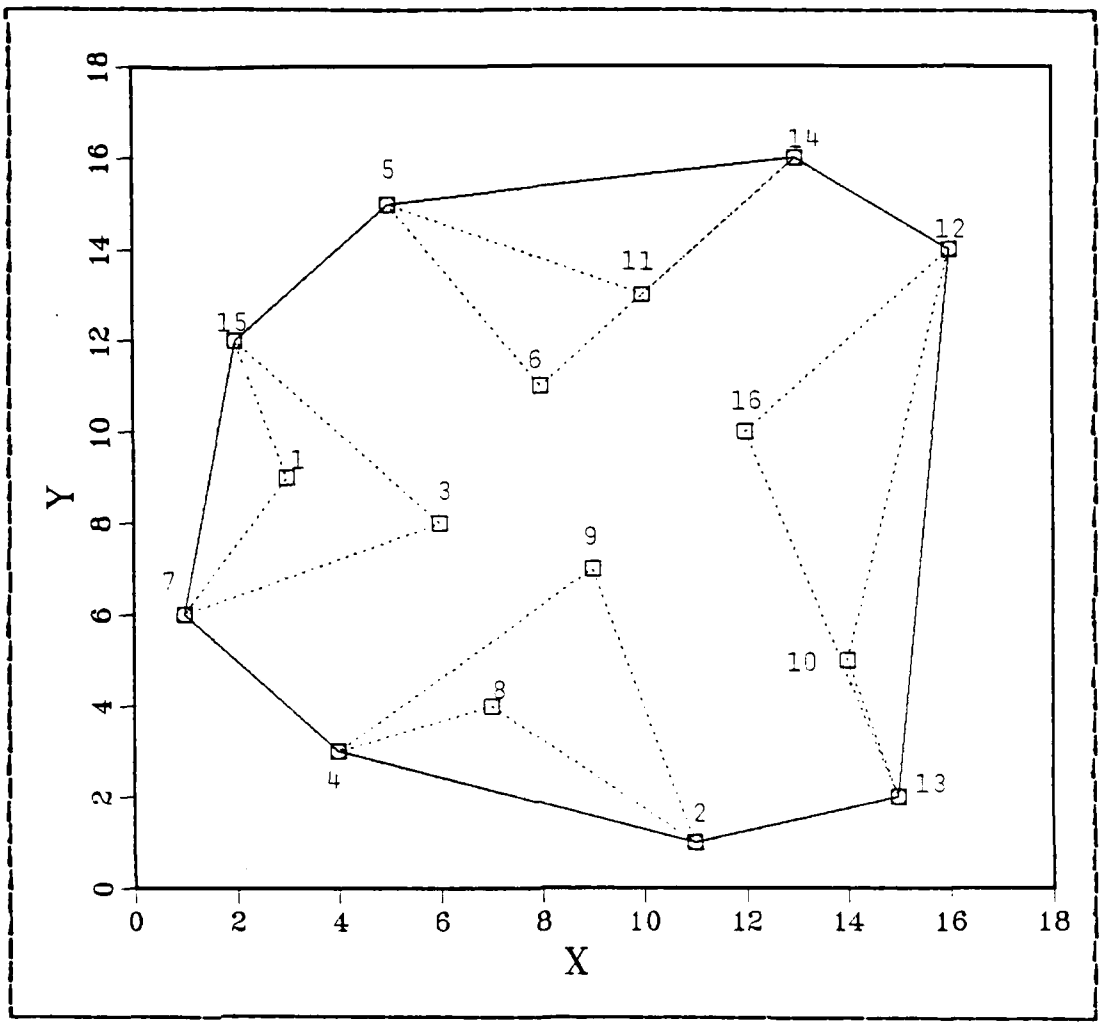


Figure 2.2 Initial Subtour and Insertion.

In step 2, each of the interior nodes (1,3,6,8,9,10,11,16) is associated with a pair of connected nodes on the initial subtour (the dashed lines in Figure 2.2). In step 3, the dashed lines that form the greatest angle (closest to 180°) identify the node to be inserted (node 10 in this example).

Figure 2.3 shows the problem after the first three insertions(node 10, node 1 and node 8 in that order). Notice that some nodes not in the subtour are associated

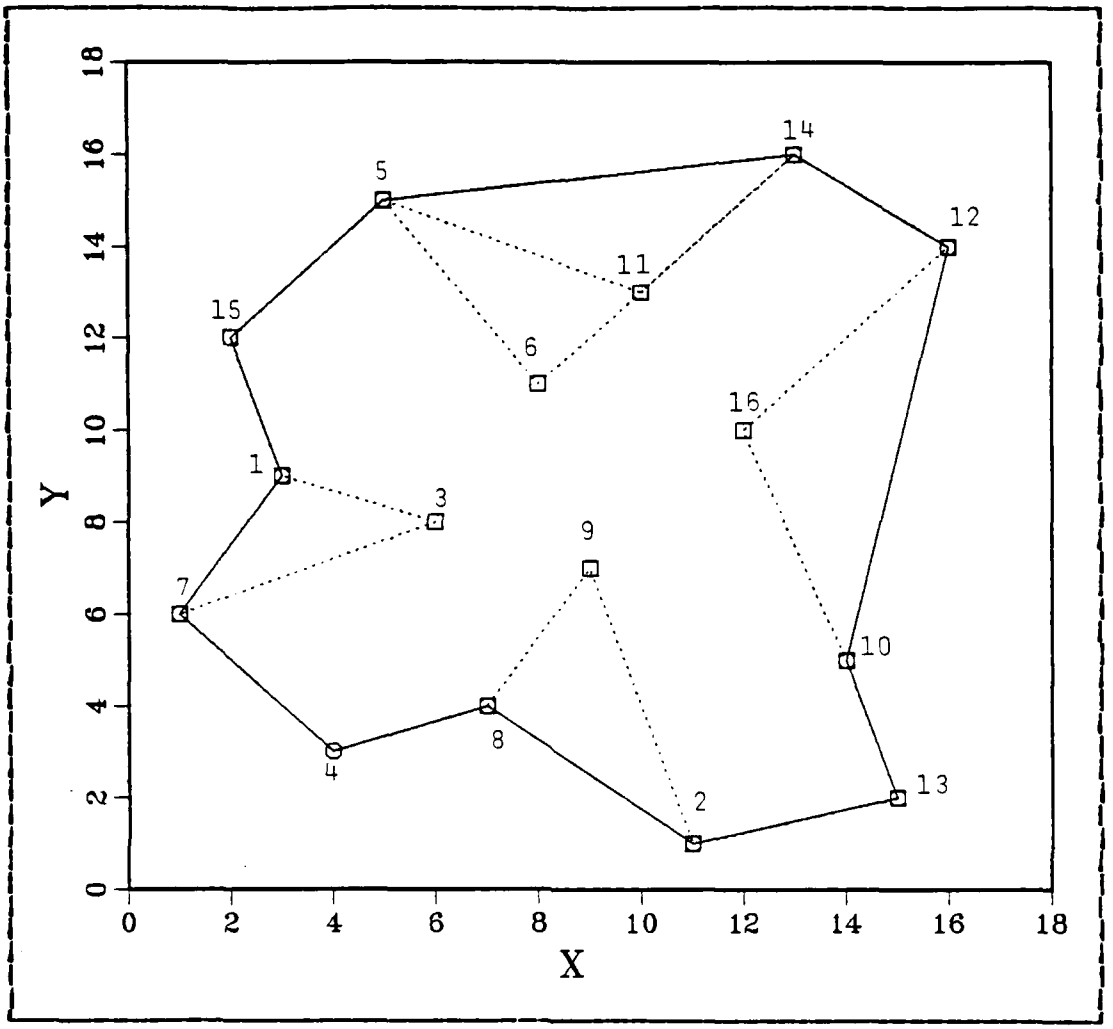


Figure 2.3 Intermediate Subtour and Insertions.

with new node pairs. Figure 2.4 shows the final tour for stage one. This tour is now passed to an Oropt post-processor. In this case the tour from stage one appears from inspection to be optimal, and Oropt will find no improvement.

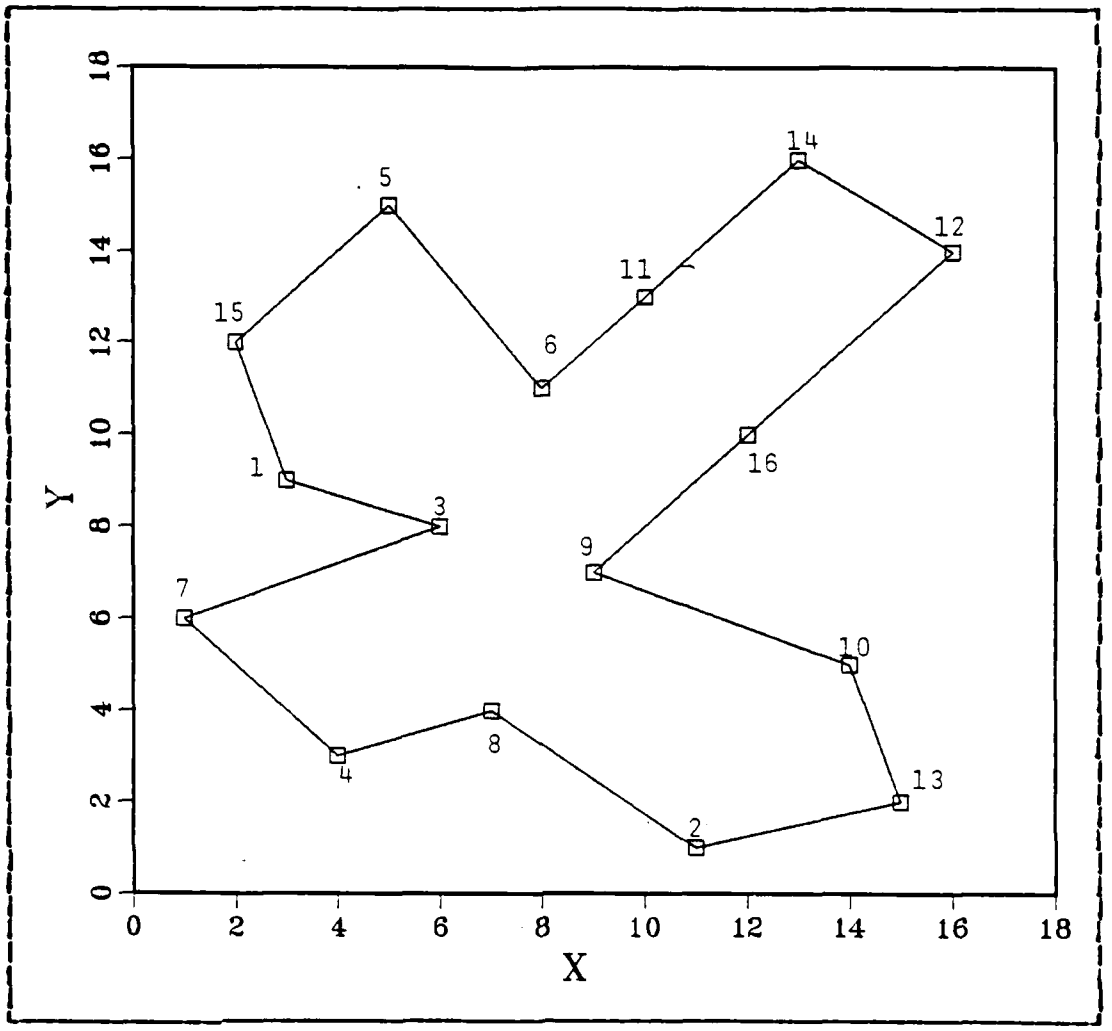


Figure 2.4 Final Tour of CCAO.

3. Computational Results

In addition to CCAO, CCCO (Convex, Cheapest, Cheapest, Oropt) has been coded for the purpose of comparison. The only difference between CCAO and CCCO is that CCCO uses the cheapest selection criterion instead of greatest angle of CCAO.

We used Sedgewick's [Ref. 19] package wrapping algorithm for finding the convex hull (initial subtour).

Starting with some point (called the anchor) that is guaranteed to be on the convex hull (say the one with the smallest y co-ordinate), take a horizontal ray in the positive direction and sweep it upward until hitting another point. This pcint is on the hull. Then start at that point and continue sweeping until hitting another point, etc. The package is completely wrapped when the first point is included again. The following algorithm finds the convex hull of an array L(1,...,n) of nodes, the node L(n+1) is used as a sentinel, that is, a copy of the first node which is used to signal completion of the procedure. The variable NH is maintained as the number of nodes so far included on the hull.

Algorithm : Package Wrapping

Input : Number of nodes, x and y co-ordinates of all nodes.

Cutput: Ordered list of convex hull and number of nodes included on the convex hull.

Step 1 . (Initialization)
Find and duplicate anchor. I.e., find NMIN = argmin y_i s.t. $i \in N$ and set

NH = 0, L(n+1) = L(NMIN).

Step 2 : (Swap nodes NH and NMIN).
Put last node found into the hull by exchanging it with the NHth node.
NH = NH + 1.
TEMP = L(NH).
L(NH) = L(NMIN).
L(NMIN) = TEMP.

Step 3 : (Compute angle)
Compute the angle from the horizontal made

by the line between L(NH) and each of the nodes not yet included on the hull.

Step 4 : (Find next hull node)
Find the node whose angle is smallest among those with angles bigger than the current value of the 'sweep' angle (the angle from horizontal to the line between L(NH-1) and L(NH)).

Step 4 : Stop when the first point is encountered again. I.e., L(n+1) = L(NMIN) .
Otherwise, go to step 2.

End of algorithm Package Wrapping

We used Sedgewick's Pseudo Angle for finding the smallest angle in step 3, which is coded as the 'THETA' function. This function returns a real number between 0.0 to 4.0 that is not the angle made by L1 and L2 with the horizontal but which has the same order properties as the true angle. If dx and dy are the delta x and y distances from some node to the anchor node, then the angle needed in this algorithm is arctangent dy/dx. However, the arctangent function is likely to be slow and it leads to at least two annoying extra conditions to compute : whether dx is zero, and which quadrant the point is in.

In this algorithm we only need to be able to compare angles, not measure them. Thus it makes sense to use a function that is much easier to compute than the true angle but has the same ordering properties as the true angle. A good candidate for such a function is simply dy / (dy + dx). Testing for exceptional conditions are still necessary, but simpler.

Function THETA (Pseudo Angle)

Input : dx,dy (delta x and y distances from some node to the anchor node).

```
Cutput : Pseudo  angle made by L1 and L2 with the
         horizontal line.

begin
dx = x(L2) - x(L1) : ax = abs(ax) :
dy = y(L2) - y(L1) : ay = abs(ay) :
if ( dx=0 ) and ( dy=0 )  then t = 0.0
                          else t = dy / (ax + ay )
if dx < 0  then  t = 2.0 - t
           else if dy < 0 then t = 4.0 + t
end
```

End of function THETA

Figure 2.2 shows how the hull is discovered in this way. We used Sedgewick's Pseudo Angle for finding the greatest angle selection point.

The data for our test problems is given in the Appendix. The computational results are summarized in Table I. As can be seen in Table I, CCAO is faster than CCCO on the small-scaled test problems (below 30 nodes), but CCCO is faster than CCAO on the moderately large sized problems (over 50 nodes). Generally, the accuracy is almost identical in both cases.

Stewart [Ref. 6] showed that in a large scaled problem, the CCAO algorithm outperforms any other insertion and selection algorithms. Thus, we are highly motivated to use a modification of the CCAO algorithm for solving the time-window constrained TSP.

TABLE I

COMPUTATIONAL RESULTS OF CCCO, CCAO

Problem Number	Number of Nodes n	Best Known Solution	CCCO		CCAO	
			% Over Best	CPU Time (sec)	% Over Best	CPU Time (sec)
[1]	16	66.6039	0.00	0.0133	0.00	0.0066
[2]	22	469.0288	0.00	0.0233	0.00	0.0100
[3]	22	278.4371	0.00	0.0166	0.00	0.0066
[5]	51	429.7000	2.72	0.1897	3.94	0.2562
[6]	76	552.9000	1.64	0.5857	1.54	0.6889

* CPU times in seconds on IBM 3033.

III. THE TSP WITH HARD TIME WINDOW CONSTRAINTS

A. INTRODUCTION

The first time-constrained TSP we consider is the case in which late arrivals are not allowed, and early arrivals must wait for the opening of the time window before they can begin to service a customer. This is called the hard time window case and it is illustrated in Figure 3.1.

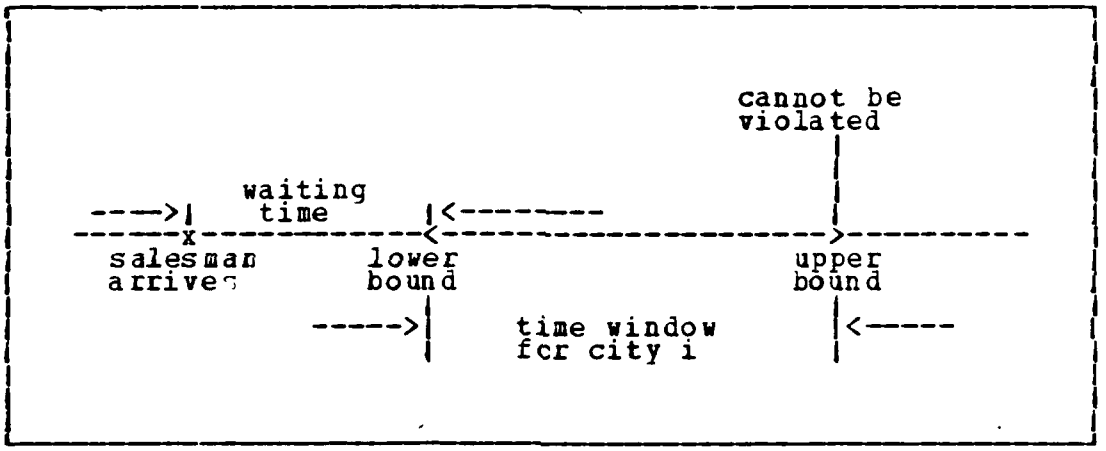


Figure 3.1 Diagram for Hard Time Window Case.

The hard time window case corresponds to military operations and to some civilian distribution problems. Meeting a deadline is considered a critical factor in this case. The soft time window case will be discussed in the nextChapter.

Consider a graph G = {N,A} composed of a set of nodes N and a set of arcs A connecting these nodes. We now define some notation to be used throughout our discussion of the time-window-constrained TSP.

l_i = Lower bound on the time window at node i
(early allowable arrival time at city i).

u_i = Upper bound on the time window at node i
(latest allowable arrival time at city i).

d_i = Time required to spend at node i.
(service time at city i).

SPEED = Constant speed at which the vehicle travels.

$dist_{ij}$ = Distance from i to j.

c_{ij} = Travel time from i to j.

Note : $c_{ij} = dist_{ij} / SPEED$.

We use c_{ij} and c(i,j) interchangeably.

depot = Depot(home) node.

L = (L(1),L(2),...,L(n)).
= A tour with n stops visited in the order L(1),L(2),...,L(n).

$ARRVT_i$ = Arrival time at city i.

$WAIT_i$ = Waiting time at node i for the hard time window.

We also use l(i), u(i), d(i), ARRVT(i), WAIT(i) and l_i, u_i, d_i, $ARRVT_i$, $WAIT_i$ interchangeably.

B. HEURISTIC SOLUTION TECHNIQUES FOR HARD TIME WINDOWS

1. Nearest Neighbor

The following is a Nearest Neighbor heuristic similar to the one used in the unconstrained TSP. At each iteration we add a new node to the end of the subtour. It is the first node that can be visited from the last node of

the subtour, taking into account any waiting time that might be necessary due to the lower time window bounds.

Algorithm : Nearest Neighbor

Input : Number of nodes, x and y co-ordinates of all nodes, time windows for all nodes.

Output : Ordered list of tour, total travel time.

Step 1 . (Initialization)
Start at the depot.
Let i=depot, $N' = \{i\}$.

Step 2 . Compute ARRVT for all nodes $k \in N-N'$ if k can be visited directly after i :
$ARRVT_k = \max \{ ARRVT_i , l_i \} + d_i + c_{ik}$.

Step 3 . If $ARRVT_k > u_k$, then stop (' no feasible solution').

Step 4 . If $ARRVT_k < l_k$, then $cost_k = l_k$.
Otherwise, $cost_k = ARRVT_k$.

Step 5 . (Nearest Neighbor Selection)
Choose the node $k \in N-N'$ such that $cost_k$ is a minimum. I.e, find
$k = \text{argmin } cost_j$ s.t. $j \in N-N'$.

Step 6 . (Insertion)
Insert k after i, add k to subtour N', and let i = k.

Step 7 . If N'=N, go to next step.
Otherwise, return to step 2.

Step 8 . Compute total travel time, then stop.
Total travel time = $\max \{ ARRVT_k , l_k \}$
$+ d_k + c_{k,depot}$.

End of algorithm Nearest Neighbor

This solution was constructed by starting at the depot and moving to the nearest neighboring customer that has not yet been visited as long as the upper bound level was not violated. This heuristic may fail to solve the problem.

2. SCCO

This algorithm is designed for the case when some of the nodes do not have time windows. We call these nodes " time free ".

SCCO is similar to the cheapest selection, cheapest insertion method for the unconstrained TSP, except that the nodes with time windows are treated differently from the time free nodes. The nodes with windows are inserted in order of increasing upper time window bound.

The time free nodes are inserted between these nodes by cheapest selection and cheapest insertion, for as long as the upper bound time window will allow. In the end, a Modified Oropt is used to improve the solution.

There is one possible difficulty with this approach. It may become impossible to reach some of the time-constrained nodes before their upper bound. In this case, we must delete some node(s) from the subtour. Whenever we see an upper bound that cannot be satisfied, we select a node to delete by the following criteria.

The first criterion is the width of the time window. Hence, time-free nodes are considered first. Then, if several nodes in the subtour are tied for the widest time window, we select for deletion the node that results in the greatest time saved. The algorithm is summarized as follows.

Algorithm : Successive Cheapest Cheapest Oropt (SCCO)

Input : number of nodes, x and y co-ordinates of all

nodes, time windows for all nodes.

Output : ordered list of tour, total travel time.

Step 1 . (Initialization)
Start at the depot.
Let i=depot, $N' = \{i\}$.

Step 2 . Set $k = \text{argmin } u_j$ s.t. $j \in N-N'$.
If k is time free node, then set k = depot.

Step 3 . Calculate $ARRVT_k$.
$ARRVT_k = \max \{ ARRVT_i, l_i \} + d_i + c_{ik}$.

Step 4 . If $ARRVT_k \leq u_k$, then go to step 5.
Otherwise, select time free node $m \in N'$ which results in the greatest time saved for deletion. Delete node m from N', go to step 3.

Step 5 . Add node k to the subtour N'.

Step 6 . Insert time free node $j \in N-N'$ between nodes i and k by cheapest insertion and cheapest selection (same as CCCO) until $ARRVT_k$ does not exceed u_k .
If $ARRVT_k < l_k$, then set $ARRVT_k = l_k$.

Step 7 . If N'=N, then go to next step.
Otherwise, let i = k and go to step 2.

Step 8 . Apply the Modified Oropt procedure to the current tour. Stop when no further improvements can be found.

End of algorithm SCCO

"Successive" means select the node successively by the smaliest upper bound. In the SCCO algorithm, if the

salesman arrives before the lower bound of the time window, adding waiting time, we set the arrival time equal to the lower bound.

The Modified Oropt procedure for improving the solution is described below. This procedure consider only those exchanges that would result in a node being inserted between two other nodes in the current tour.

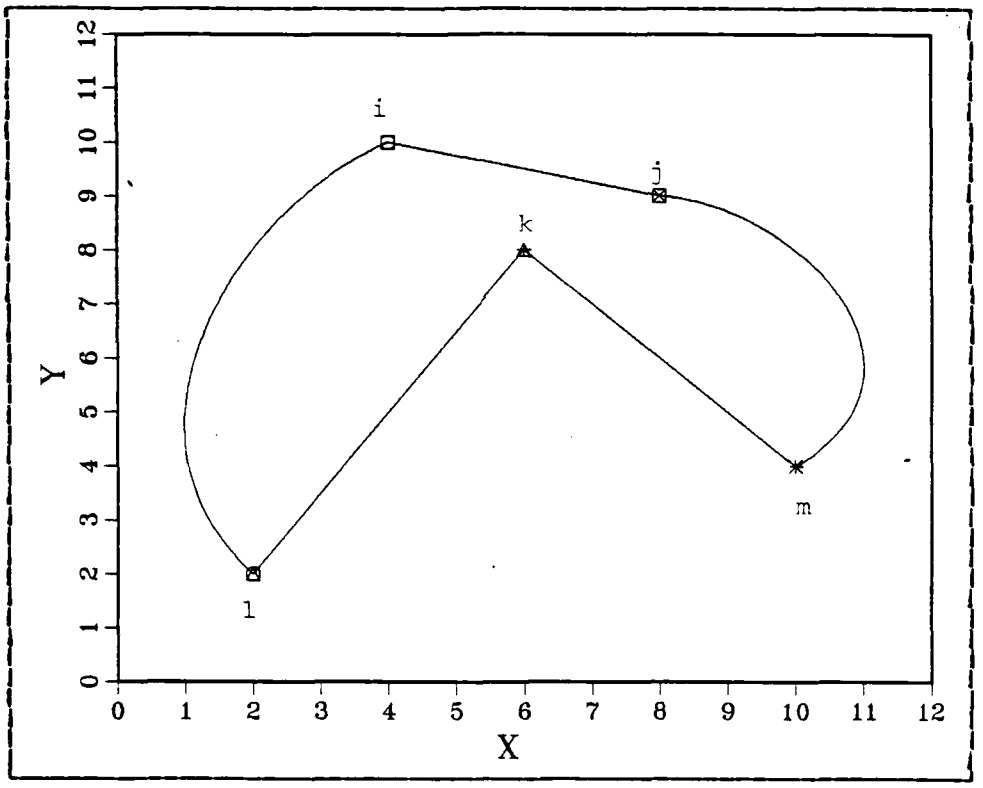


Figure 3.2 Current Tour before Modified-Oropt.

Figures 3.2 and 3.3 are helpful to understand how the procedure works. In both figures, i,j,k,l, and m are the nodes in the current tour. Nodes l and m are considered to

be adjacent to node k. A test is then conducted to determine if node k can be located between two other nodes, such as i and j, so that it results in reduced total travel time. If it can, we make the appropriate arc exchanges, then update the total cost and route orders.

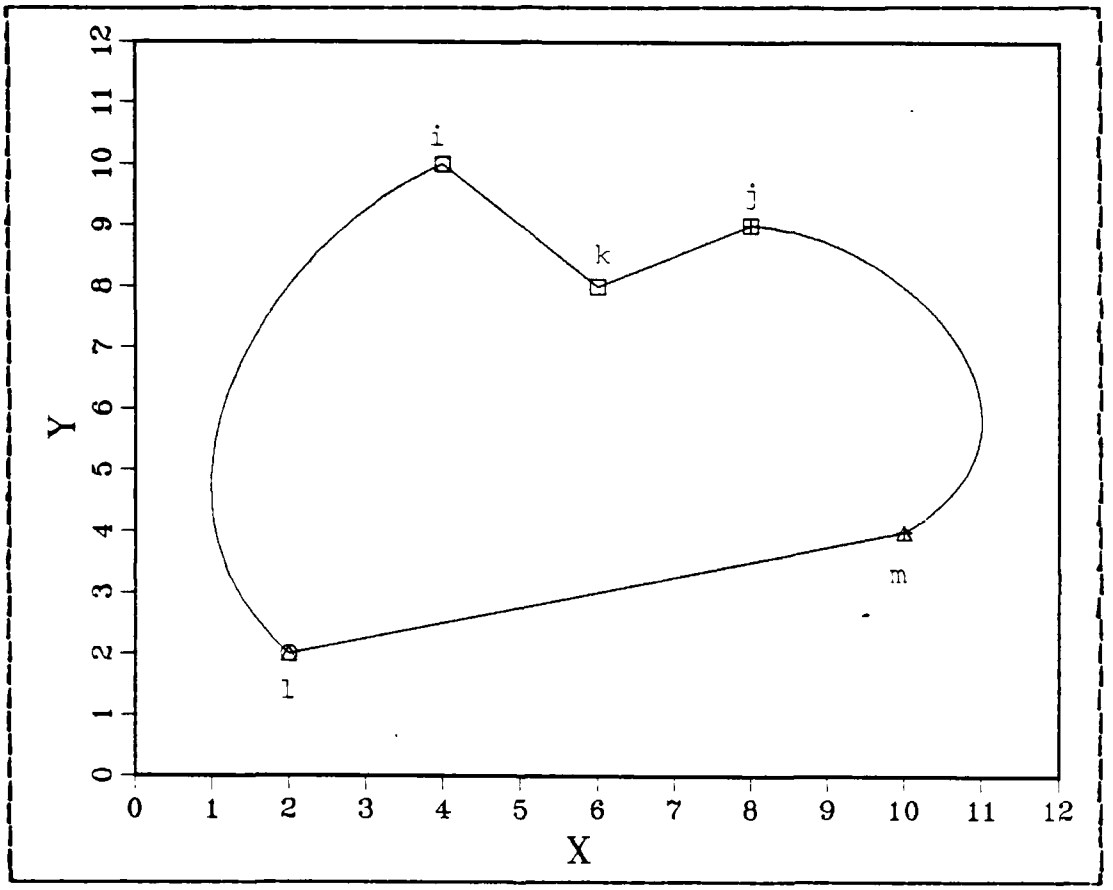


Figure 3.3 Improved Tour after Modified-Oropt.

In this example, the three arcs (i,j), (k,l), and (k,m) are removed and replaced by (i,k), (j,k), and (l,m). When no further exchanges improve the solution, the algorithm terminates.

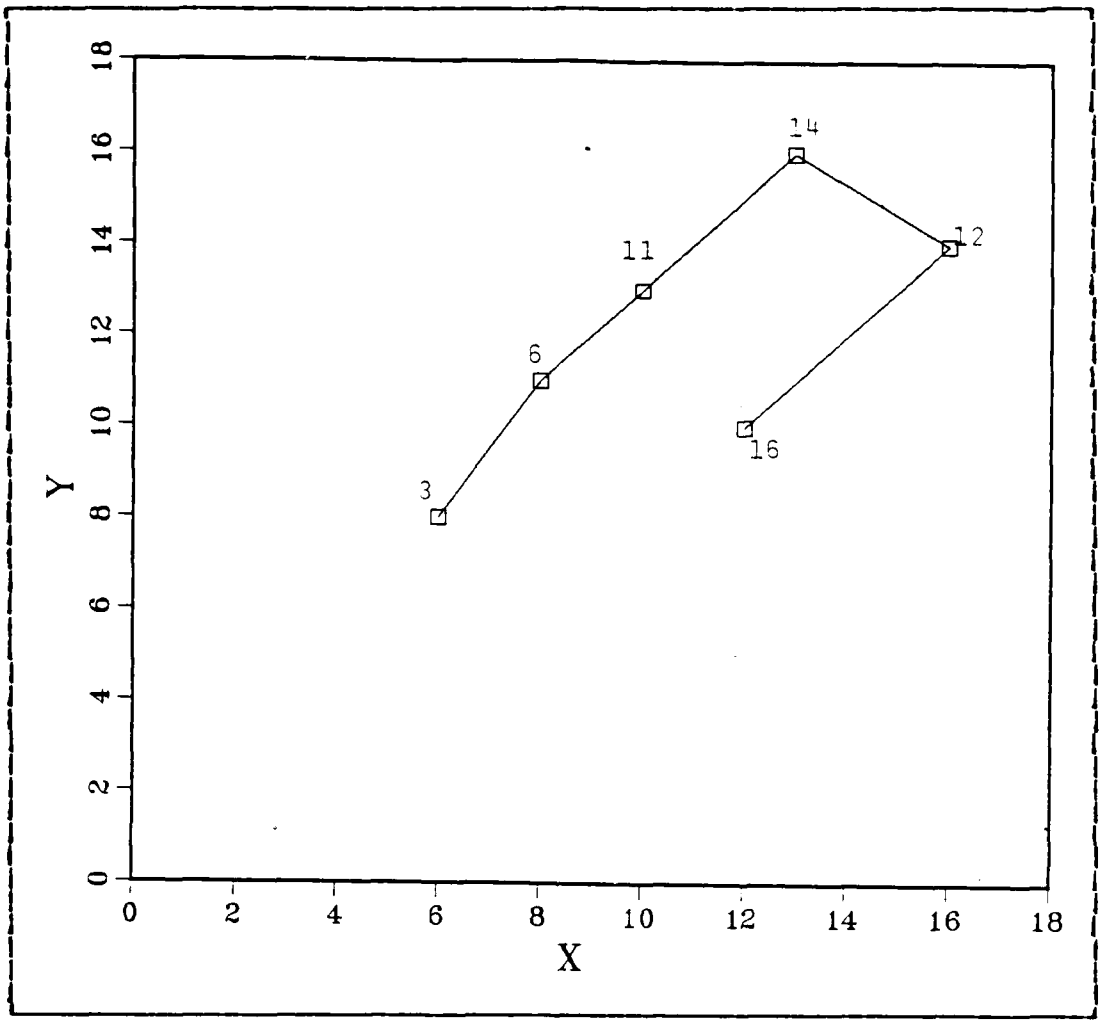


Figure 3.4 Subtour in SCCO Procedure.

Figures 3.4 - 3.6 illustrate the SCCO algorithm for the TSP with hard time windows given in Appendix F as in test problem [1]. In this problem 10 of 16 nodes have time windows. The other 6 nodes are time free.

First, the subtour starts at the depot (node 16) and we insert the node with the smallest upper bound (node 12). We examine all nodes which could be inserted between 16 and 12 as long as the upper bound on node 12 is observed. In

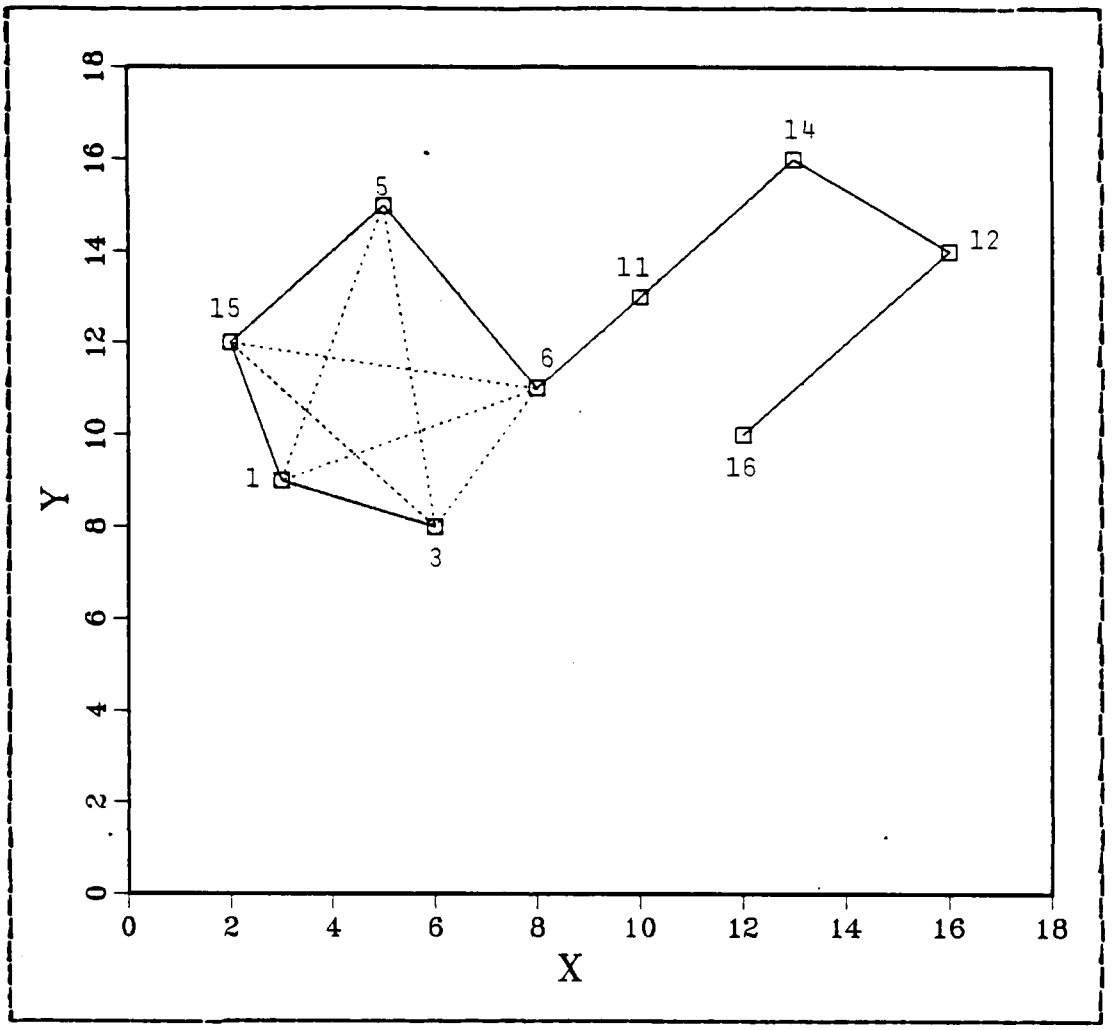


Figure 3.5 Intermediate Subtour in SCCO Procedure.

this case there is no such node. Then we select the next smallest upper bound (node 14), add it to the tour, look for nodes to insert before it, and continue in this manner. Now we have formed the partial tour 16, 12, 14, 11, 6, 3 as shown in Figure 3.4.

As shown in Figure 3.5, we can insert 3 time free nodes between node 6 and node 3. These insertions are made according to the cheapest insertion and cheapest selection

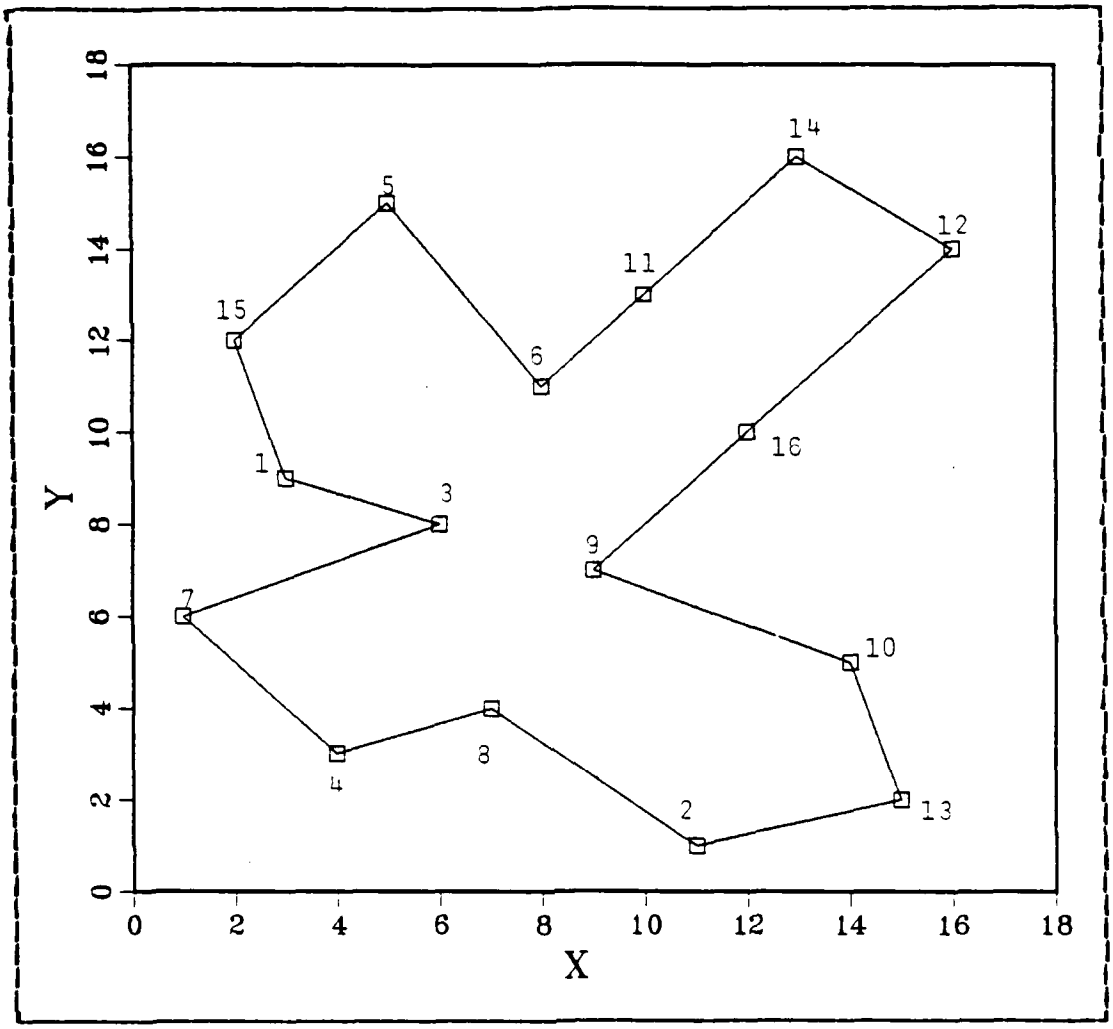


Figure 3.6 Final Subtour for SCCO.

criteria. We do not make any further insertions because they would cause a time window violation at node 3.

Figure 3.6 shows the final tour for the SCCO heuristic. This tour is passed to a Modified Oropt, but in this case it will find no improvement.

3. SCAO

This heuristic is identical tc SCCO except for the use of the greatest angle selection criterion for the time-free nodes, instead of cheapest selection.

Algorithm : Successive Cheapest Angle Oropt (SCAO)

Input : Number of nodes, x and y co-ordinates of all nodes, time windows for all nodes.

Output : Ordered list of tour, total travel time.

Step 1 . (Initialization)
Start at the depot.
Let i=depot, $N' = \{i\}$.

Step 2 . Set $k = \text{argmin } u_j$ s.t. $j \in N-N'$.
If k is time free node, set k = depot.

Step 3 . Calculate $ARRVT_k$.
$ARRVT_k = \max \{ ARRVT_i, l_i \} + d_i + c_{ik}$.

Step 4 . If $ARRVT_k \leq u_k$, then go to step 5.
Otherwise, select time free node $m \in N'$ which results in the greatest time saved for deletion. Delete node m from N', go to step 3.

Step 5 . Add node k to the subtour N'.

Step 6 . Insert time free node $j \in N-N'$ between nodes i and k by cheapest insertion and greatest angle selection (same as CCAO) until $ARRVT_K$ does not exceed u_k .
If $ARRVT_k < l_k$, then set $ARRVT_k = l_k$.

Step 7 . Let i = k.
If N'=N, then go to next step.
Otherwise, go to step 2.

Step 8 . Apply the Modified Oropt procedure to the current tour. Stop when no further improvement can be found.

End of algorithm SCAO

This algorithm is same as SCCO except greatest angle selection is used instead of cheapest selection, as in SCCO.

4. SLACK

This heuristic was suggested by Professor Rosenthal. It is designed for the case when the widths between the upper and lower bounds of the time windows are relatively large.

In this heuristic, the SLACK is the most important concept. The SLACK(i) can be defined as the maximum amount of time by which arrival at node i can be delayed without causing an upper bound to be violated for a node currently on the tour.

The SLACK function can be defined as a recursive function as follows.

$$SLACK(L(i)) = \min \{ u(L(i)) - ARRVT(L(i)) , SLACK(L(i+1)) + WAIT(L(i)) \}$$

where

$$WAIT(L(i)) = \max \{ 0, l(L(i)) - ARRVT(L(i)) \}$$

The first element of this recursive function is the difference between the upper bound and arrival time at node L(i). The second one is the sum of next node's SLACK and waiting time of node L(i). The minimum of these two elements is a possible delay time of the arrival time at node L(i) without violating the upper bound of all nodes after L(i) in the current tour.

The advantage of this recursive function is that it is easy to calculate a possible delay time without calculating new arrival times for all nodes after L(i). The algorithm is summarized as follows.

Algorithm : SLACK

Input : Number of nodes, x and y co-ordinates of all nodes, time windows for all nodes.

Output : Ordered list of tour, total travel time.

Step 1 . (Initialization)
Start at the depot. Let N' = {depot}.

Step 2 . Sort the upper time windows.
$U = (u_1, u_2, \ldots, u_n)$
s.t. $u_1 \leq u_2 \leq \ldots \leq u_n$.

Step 3 . Set $k = \text{argmin}_i \, u_i$ s.t. $i \in N-N'$.

Step 4 . Find a node L(ISTAR) after which node k can be inserted in the current sequence, if such a node exists. Go to step 7.
(The criteria by which we determine if an insertion can be made are given below.)

Step 5 . If there is no such place to insert node k, then try to find a node L(ISWAP) in the current sequence such that k can replace L(ISWAP) and L(ISWAP) has a good chance of being reinserted somewhere else.
Select ISWAP which has the largest time window width among candidates for ISWAP.
If there is no candidate, then stop.
(' no feasible solution ')

Step 6 . Do swap (add k to N', and delete L(ISWAP) from N', and set k = L(ISWAP)),
then update slack and arrival times.
Go to step 3.

Step 7 . Select the node which results in the minimum additional travel, i .e, the node k which minimizes the following quantity.
$c(L(I),k) + c(k,L(I+1)) - c(L(I),L(I+1))$.

Step 8 . Insert k after L(ISTAR), and add k to N', and update slack and arrival times.

Step 9 . If N'=N, then stop.
Otherwise, go to step 3.

End of algorithm SLACK

This procedure starts with sorting an array $u_1, u_2, \ldots, u_n$ into ascending order using a heapsort [Ref. 20]. This u array is used to select a node k in ascending order for insertion. Since the upper bound cannot be violated, this step is performed. Then find a node L(I) after which node k can be inserted in the current sequence, if a such a node exists.

There are two tests which must be administered to determine if k can be inserted after L(I). First, the arrival time at node k if k succeeds L(I), which is called TEST1 must not be greater than the upper bound u . Second, if k precedes L(I+1), then the resulting delay in arrival at L(I+1), which is called TEST2 , must not greater than SLACK(L(I+1)). We can calculate TEST1, TEST2 as follows.

TEST1 = Arrival time at node k if k succeed L(i).

= max {ARRVT(L(i)), l(I(i))} + d(L(i)) + c(L(i),k).

TEST2 = Delay in arrival at L(i+1) if k precedes L(i+1).

= max { TEST1, l(k) } + d(k) + c(k,L(i+1)).

If there exists more than one node L(I) after which node k can be inserted, we select L(I) according to the criterion of least additional travel time. This additional travel time, called TEST3, is given by

$$TEST3 = c(L(I),k) + c(k,L(I+1)) + C(L(I),L(I+1)).$$

When we insert node k after L(I), we update the arrival times and SLACKs. In the updating process, we compute updated values of SLACK only for the nodes whose SIACK actually changes as a result of the insertion.

If there is no place to insert node k, we call a subroutine called 'TSWAP'. TSWAP tries to find a node L(ISWAP) in the current sequence such that k can replace L(ISWAP) and L(ISWAP) has a good chance of being reinserted somewhere else. TSWAP uses IEST1 ans IEST2 to find a candidate for ISWAP and then uses a largest time window width to select ISWAP. If there exists such a ISWAP, then we do the swap and update SLACKs and arrival times and try to insert again.

C. EXACT SOLUTION TECHNIQUES FOR HARD TIME WINDOWS

1. State-Space Relaxation Procedure

A dynamic programming model of the time-constrained TSP has been developed by Christofides et al. [Ref. 5]. We applied their approach to compute bounding information within a branch and bound algorithm.

Consider the TSP defined on the graph G = {N,A} with the time windcw constraints, where N is a set of all nodes of G, and A is a set of arcs. Let R(j) be the set of nodes from which it is possible to go directly to node j. We can initially set $R(j) = N - \{ i \mid l_i + \bar{d}_i + c_{ij} > u_j \}$, because it is impossible to go directly from ncde i to node j if the earlist possible arrival time at node j exceeds the upper time window cf node j.

Let $f(S,j)$ be the duration of the least time path starting at node 1 passing through every node of $S \subseteq S' = N-\{1\}$ and finishing at node j. For a given S and j, we can calculate a minimum arrival time in node j as

$$T(S,j) = \min_{i \in (S-j \cap R(i))} [\, f(S-j,i) + d_i + c_{ij} \,]. \qquad (3.1)$$

Then,

$$\begin{aligned} f(S,j) &= T(S,j), \text{ if } l_j \le T(S,j) \le u_j \\ &= l_j, \text{ if } T(S,j) \le l_j \\ &= \infty, \text{ if } T(S,j) > u_j \end{aligned}$$

with the initialization:

$$\begin{aligned} f(\{j\},j) &= c(1,j), \text{ if } l_j \le c_{1j} \le u_j \\ &= l_j, \text{ if } c_{1j} \le l_j \\ &= \infty, \text{ if } c_{1j} > u_j \end{aligned}$$

In equation (3.1) the minimum arrival time in node j passing through the nodes in the set S can be described as the sum of three terms : the first is the duration of the least time path passing through the nodes in the set $S-\{j\}$ and ending in node i, the second is the time required to spend in node i, and the third is the travel time from node i to node j.

The $f(S,j)$ can be calculated for all subsets S of S' and for all nodes j by using equation (3.1) recursively. Finally, the optimum solution can be calculated as

$$\min_{i \in S'} [\, f(S',i) + d_i + c_{i1} \,].$$

Since the computer storage requirements increase exponentially with the size of the problem, this method is limited to small problems. The total number of $f(S,j)$, when S contains k nodes, is $k\binom{n-1}{k}$, since $f(S,j)$ must be calculated for all subsets S of S', and since each node in S must

be considered as a possible end-node j. Therefore the storage requirement for f(S,j) in a n node problem, is given by [Ref. 21].

$$\sum_{k=1}^{n-1} k \binom{n-1}{k} = (n-1)\ 2^{n-2} \qquad (3.2)$$

The storage requirements to solve a 22 node problem exceed 22,020,096. For relaxing this limitation, Christofides et al. [Ref. 5] proposed a state space relaxation procedure which is analogous to Lagrangean relaxation [Ref. 22] in integer programming. The state space associated with a given dynamic programming recursion is relaxed in such a way that the solution to the relaxed recursion provides a bound which could be embedded in a general branch and bound method [Ref. 23]. We describe Christofides et al's method for doing this below.

Consider the dynamic programming formulation (3.1) The state variable in that formulation is (S,j), and the stage is the cardinality of S. Let g(S) be a mapping from the domain of (S,j) to some other vector space (g(S),j). Let:

$$H(g(S),j) = \{ (g(S-j),i) \mid i \in (S-j \cap R(j)) \} \qquad (3.3)$$

Since we are interested in lower bounds to the TSP with time constraints, H(g(S),j) in (3.3) may be replaced by any larger set that is easier to compute. Thus, H(g(S),j) can be defined by the following equation:

$$H(g(S),j) = \{ (g(S-j),i) \mid i \in E(g(S),j) \} \qquad (3.4)$$

where $(S-j \cap R(j)) \subseteq E(g(S),j)$.

For calculating the lower bound of the problem, equation (3.1) can be changed to the following equation:

$$T(g(S),j) = \min_{(g(S-j),i) \in H(g(S),j)} [f(g(S-j),i) + d_i + c_{ij}] \qquad (3.5)$$

$$= \min_{i \in E(g(S),j)} [f(g(S-j),i) + d_i + c_{ij}] \quad (3.6)$$

This gives us:

$$f(g(S),j) = T(g(S),j), \text{ if } l_j \le T(g(S),j) \le u_j$$
$$= l_j, \text{ if } T(g(S),j) \le l_j$$
$$= \infty^j, \text{ if } T(g(S),j) > u_j.$$

With the initialization:

$$f(g(j),j) = c_{1j}, \text{ if } l_j \le c_{1j} \le u_j$$
$$= l_j, \text{ if } c_{1j} \le l_j$$
$$= \infty^j, \text{ if } c_{1j} > u_j$$

Finally, the optimum solution can be calculated as

$$\min_{i \in E(g(N),1)} [f(g(S'),i) + d_i + c_{i1}].$$

The mapping can be selected from any separable function. Christofides et al. used the following mapping function.

$$g(S) = |S|. \quad (3.7)$$

Then equation (3.6) becomes :

$$T(k,j) = \min_{i \in E(k,j)} [f(k-1,i) + d_i + c_{ij}] \quad (3.8)$$

where $k = |S| > 1$.

This gives us:

$$f(k,j) = T(k,j), \text{ if } l_j \le T(k,j) \le u_j$$
$$= l_j, \text{ if } T(k,j) \le l_j$$
$$= \infty^j, \text{ if } T(k,j) > u_j.$$

With the initialization:

$$f(1,j) = c(1,j), \text{ if } l_j \le c_{1j} \le u_j$$
$$= l_j, \text{ if } c_{1j} \le l_j$$
$$= \infty^j, \text{ if } c_{1j} > u_j$$

Finally, the optimum solution can be calculated as

$$\min_{i \in E(|N|,1)} [\, f(|S'|,i) + d_i + c_{i1} \,].$$

2. Additional Condition

In the previous section, we discussed Christofides et al.'s state space relaxation procedure which provides a lower bound on the TSP by reducing a state space in dynamic programming. This lower bound is effective in branch and bound only if it is a tight bound. This is similar to the case in integer programming where the effectiveness of Lagrangean relaxation in producing bounds is relative to the integer programming formulation. A redundant state-space condition can be helpful to get a better bound. For this purpose, an additional condition was used by Christofides et al. to avoid loops formed by three consecutive nodes [Ref. 5]. This can be done in the following way.

Let $k = |S|$. Let $f(k,j,1)$ be the duration of the least time path from the initial state to state (k,j) without loops formed by three consecutive nodes. Let $f(k,j,2)$ be the duration of the second least time path from the initial state to state (k,j) without loops formed by three consecutive nodes. Let $p(k,j,m)$ be the predecessor of j on the path corresponding to $f(k,j,m)$. With the above definition, recursion (3.8) becomes:

$$T(k,j,1) = \min_{i \in E(k,j)} [\, f(k-1,i,m) + d_i + c_{ij} \,], \qquad (3.9)$$

where $m = 1$, if $p(k-1,i,1) \neq j$

$= 2$, otherwise.

This gives us:

$$f(k,j,1) = I(k,j,1), \quad \text{if } l_j \leq T(k,j,1) \leq u_j \qquad (3.10)$$

$$= l_j, \quad \text{if } T(k,j,1) \leq l_j$$

$$= \infty \quad , \text{ if } T(k,j,1) > u_j .$$

Recursion for $f(k,j,2)$ can be written in the following way.

Let:

$$T(k,j,2) = \min_{\substack{i \in E(k,j) \\ i \neq p(k,j,1)}} [\, f(k-1,i,m) + d_i + c_{ij} \,], \qquad (3.11)$$

where $m = 1$, if $p(k-1,i,1) \neq j$

$= 2$, otherwise.

This gives us:

$$f(k,j,2) = T(k,j,2), \quad \text{if } l_j \leq T(k,j,2) \leq u_j \qquad (3.12)$$
$$= l_j \quad , \quad \text{if } T(k,j,2) \leq l_j$$
$$= \infty \quad , \quad \text{if } T(k,j,2) > u_j .$$

The initialization is

$$f(1,i,1) = c(1,i) \quad , \text{ if } l_i \leq c_{1i} \leq u_i \qquad (3.13)$$
$$= l_i \quad , \text{ if } c_{1i} \leq l_i$$
$$= \infty \quad , \text{ if } c_{1i} > u_i$$

and

$$f(1,i,2) = \infty \qquad (3.14)$$

Finally, the optimum solution can be calculated as

$$\min_{i \in E(|N|,1)} [\, f(|S'|,i,1) + d_i + c_{i1} \,]. \qquad (3.15)$$

Since the additional condition can avoid consideration of a useful lower bound, we considered $f(k-1,i,2)$ in recursion (3.9) and (3.11) only when the predecessor of i on the path corresponding to $f(k-1,i,1)$ is j. If we do not consider the second least time path in case of $p(k-1,i,1)=j$, then $f(g(S),j)$ does not guarantee the lower bound of $f(S,j)$.

For this example, let's consider a 4 node TSP with time constraints. Node A is the starting node. D is the time free node. The lower bound of node B is 9, the upper bound of node B is 11, the lower bound of node C is 19, and

the upper bound of node C is 21. Suppose service time at each node is zero. Figure 3.7 shows an optimal route for this problem.

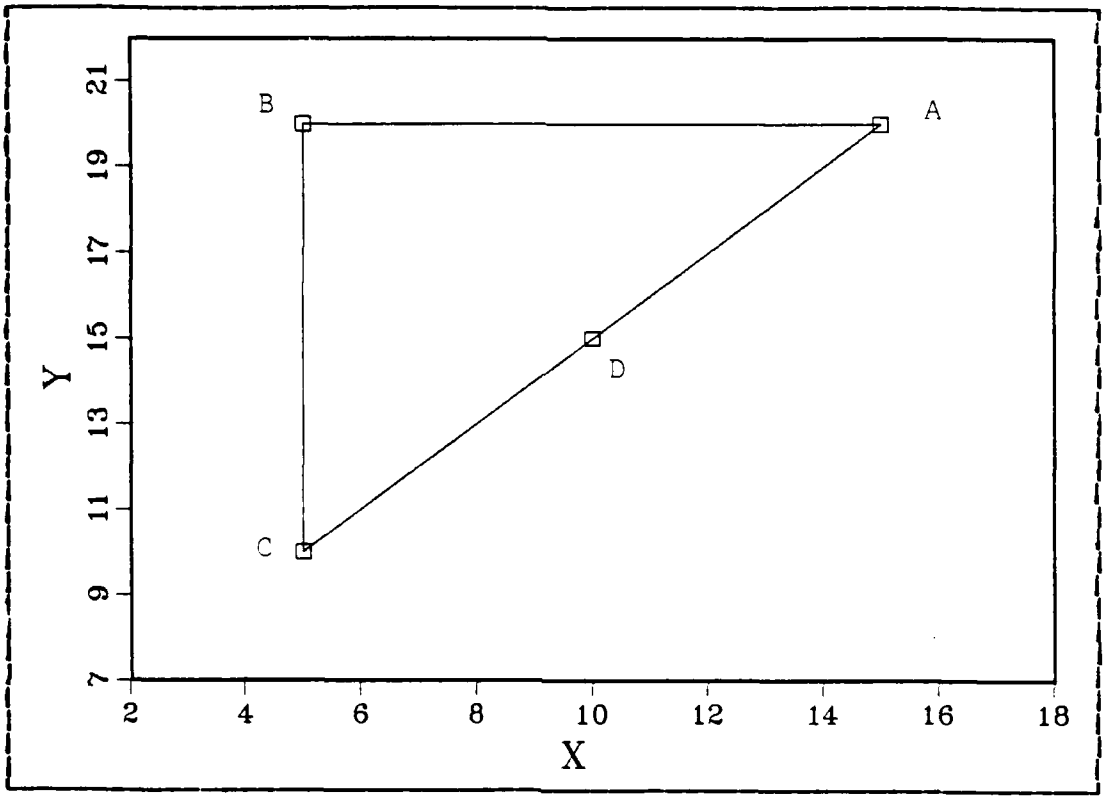


Figure 3.7 Optimal Route of Four Nodes Problem.

From equation (3.13) we can get:

$f(1,B,1) = 10,$

$f(1,C,1) = 19,$

$f(1,D,1) = 7.07$

Now applying equation (3.9) recursively with i=1, for k=2 we can get:

$f(2,B,1) = \infty$,

$f(2,C,1) = 19$,

$f(2,D,1) = 17.07$

Similarly, for k=3

$f(3,B,1) = \infty$,

$f(3,C,1) = \infty$,

$f(3,D,1) = \infty$.

We can see easily that $f(3,D,1)$ is not a lower bound of $f(\{B,C,D\},D)$.

3. Branch and Bound Procedure

In this section we introduce branch and bound enumeration which is used to eliminate subtours in the solution of the state space relaxation procedure. Since the state space relaxation procedure is a relaxation of the TSP with time constraints, the solution to the state space relaxation prccedure provides a lower bound on the optimal value of the TSP with time constraints. Any heuristic solution can provide an upper bound. We denote some notation to explain this algorithm as follows.

FLBD = The lower bound, which is the optimal solution to the state space relaxation procedure, on the optimal solution to the TSP with time constraints given restrictions at the current node.

Z = Current upper bound.

STACK = Array which represent decision tree. It contains arc lists which have the same head in optimal tour to the state space relaxation

procedure given restrictions at the current node.

$[c'_{ij}]$ = Travel time matrix given restrictions at the current node.

There are two types of tree search. One is depth-first search, the other is breadth-first search [Ref. 24]. We used depth-first search since breadth-first search required substantially more storage. Depth-first search simply means that when a separation is defined, one of the nodes created by the separation is immediately selected to be the next subproblem, and when a node is fathomed, the enumeration always backtracks to the most recently created live node.

One of the most important requirements of any branch and bound algorithm is tight bounds. The closer the bounds are to the optimal solution, the fewer nodes must be enumerated. We used the SCCO heuristic, which was described in section B.2, as an initial upper bound. The lower bound is obtained from equation (3.15).

To save computing time we need a criterion to decide whether or not the branching should be continued. If FLBD is greater than Z, then the node is fathomed since explicit enumeration need not be extended below the current node. For branching we consider the arcs which have the same head node in the directed graph since each arc must have a different head in the TSP solution. If there is no such arc, then that solution is a feasible solution. After all nodes of the tree are fathomed, a feasible solution which has the same value as the upper bound is an exact solution to the TSP with time window constraints.

The following branch and bound algorithm is used in the programs written for exact solution.

Algorithm : Branch and Bound Procedure

Input : Total travel time of heuristic, travel time.
Output: Ordered list of tour, total travel time.

Step 1 . (Initialization)
Let Z = the optimal solution of SCCO.
STACK = empty.
$[c'_{ij}] = [c_{ij}]$

Step 2 . Compute FLBD given restrictions defined by $[c'_{ij}]$. If FLBD > Z, go to step 5.

Step 3 . (Construct the tree)
Put all arc(i,j) which have the same head j in directed graph on STACK.
If there is no such arc, save feasible route and update Z = FLBD then go to step 5.

Step 4 . Let travel time of arc(i,j) which is in the top of STACK be infinite, then go to step 2. (i.e., $c'_{ij} = \infty$.)

Step 5 . (Backtrack)
If STACK = empty, go tc step 7.

Step 6 . If travel time of arc(i,j) which is in the top of STACK is finite, let travel time of that arc(i,j) be infinite, then go to step 2. (i.e., $c'_{ij} = \infty$.)
Otherwise, let travel time of that arc(i,j) be original travel time of that arc(i,j) and remove that arc(i,j) from top of the STACK. Go to step 5. (i.e., $c'_{ij} = c_{ij}$)

Step 7 . (termination)
If there is a feasible route, then the optimal travel time = Z.
Otherwise, there is no feasible solution.

End of algorithm Branch and Bound Procedure

We present the results of our computational experience with the algorithms of this Chapter in Chapter V.

IV. THE TSP WITH SOFT TIME WINDOW CONSTRAINTS

A. INTRODUCTION

The second time-constrainted TSP we consider is the case in which both late and early arrivals are allowed by paying a penalty cost. The penalties are allowed to be different for early and late arrivals. The penalty cost is calculated as follows.

Upper penalty cost = max [0, upper penalty constant x (arrival time - upper bound)].

Lower penalty cost = max [0, lower penalty constant x (lower bound - arrival time)].

In fact, the upper penalty constant is greater than the lower penalty constant in most cases. Figure 4.1 may be helpful to understand this case.

This approach makes every problem feasible, no matter what the time windows are, i.e, even if it is infeasible in the hard time window case. This reflects a practical point of view, especially when it is possible to save a great deal of mileage by allowing a small amount of time window violation.

In this Chapter, we considered one unit of cost to be the same as one unit of time. In real world problems, it is possible to get a cost by multiplying traveling time by some constant.

We use the notation lp_k and up_k for the lower and upper penalty cost at node k.

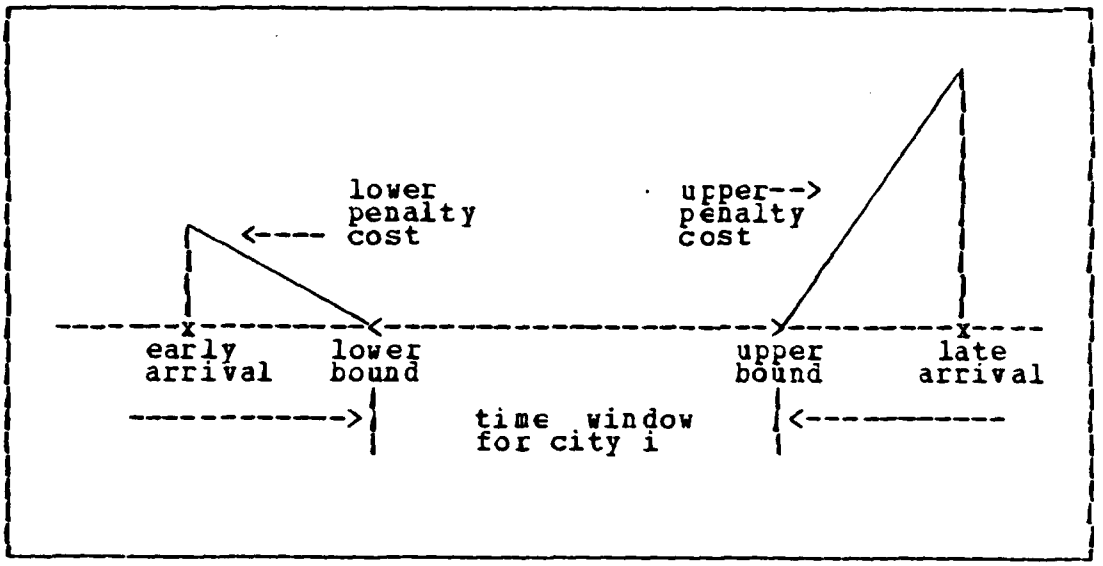


Figure 4.1 Diagram for Soft Time Window Case.

B. HEURISTIC SOLUTION TECHNIQUES FOR SOFT TIME WINDOWS

1. Nearest Neighbor

This heuristic is similar to the hard time windows except it takes into account any penalty cost that might be necessary.

Algorithm : Nearest Neighbor

Input : Number of nodes, x and y co-ordinates of all nodes, time windows for all nodes.

Output : Ordered list of tour, total cost.

Step 1 . (Initialization)
Start at the depot.
Let i=depot, $N' = \{i\}$, $cost_i = 0$.

Step 2 . Compute ARRVT for all nodes $k \in N-N'$

$$ARRVT_k = ARRVT_i + d_i + c_{ik}.$$

$$cost_k = cost_i + d_i + c_{ik}.$$

Step 3 . If $ARRVT_k < l_k$, then $cost_k = cost_k + lp_k$.

If $ARRVT_k > u_k$, then $cost_k = cost_k + up_k$.

Step 4 . (Nearest Neighbor Selection)
Select the node $k \in N-N'$ such that $cost_k$ is a minimum. I.e. , find
$k = \text{argmin } cost_j$ s.t. $j \in N-N'$.

Step 5 . (Insertion)
Insert k after i, add k to subtour N', and let i = k .

Step 6 . If N' = N , then go to next step.
Otherwise, go to step 2.

Step 7 . Compute total cost, then stop.
Total cost = $cost_k + d_k + c_{k,depot}$.

End of algorithm Nearest Neighbor

This solution was constructed by starting at the depot and moving to the nearest neighboring customer that has not yet been visited. The term "nearest" is modified in the sense that we add a penalty cost to the travel time if the time window for city i is violated.

2. SCCO

This algorithm is also designed for the case when there is a combination of tight time window nodes and time free nodes. The strict observance of the upper bound in the hard time windows is replaced by a penalty cost.

Algorithm : SCCO

Input : Number of nodes, x and y co-ordinates of all nodes, time windows for all nodes.

Output : Ordered list of tour, total cost.

Step 1 . (Initialization)

Start at the depot.

Let i=depot, $N' = \{i\}$, $cost_i = 0$.

Step 2. Set $k = \text{argmin } u_j$ s.t. $j \in N-N'$.

If k is time free node, then set k = depot.

Step 3. Insert node k in the subtour N'.

Compute $ARRVT_k$

$$ARRVT_k = ARRVT_i + d_i + c_{ik}.$$

Step 4. Insert time free node $j \in N-N'$ between nodes i and k by cheapest insertion and cheapest selection (same as CCCO) until $ARRVT_k$ does not exceed u_k.

Step 5. Update $cost_k$.

$$cost_k = cost_i + d_i + c_{ik}.$$

If $ARRVT_k < l_k$, then $cost_k = cost_k + lp_k$.

If $ARRVT_k > u_k$, then $cost_k = cost_k + up_k$.

Step 6. Let i = k.

If N' = N, then go to next step.

Otherwise, go to step 2.

Step 7. Apply the Modified Oropt procedure to the current tour. Stop when no further improvements can be found.

End of algorithm SCCO

This procedure is also similar to the cheapest selection, cheapest insertion method for the unconstrained TSP, except that the nodes with time windows are treated differently from the time free nodes. The nodes with time

windows are inserted in order of increasing upper time window bounds. The time free nodes are inserted between those nodes by cheapest selection and cheapest insertion, for as long as the upper bound of the time windows will allow.

In the end, a Modified Oropt is used to improve the solution. This procedure consider only those exchanges that would result in a node being inserted between two other nodes in the current tour.

3. SCAO

This algorithm is also designed for the time window set which is composed of some tight time windows and some time free nodes.

Algorithm : SCAO

Input : Number of nodes, x and y co-ordinates of all nodes, time windows for all nodes.

Output : Ordered list of tour, total cost.

Step 1 . (Initialization)
Start at the depot.
Let i=depot, $N' = \{i\}$, $cost_i = 0$.

Step 2. Set $k = \text{argmin } u_j$ s.t. $j \in N-N'$.

If k is time free node, set k = depot.

Step 3 . Insert node k in the subtour N'.
Compute $ARRVT_k$

$ARRVT_k = ARRVT_i + d_i + c_{ik}$.

Step 4 . Insert time free node $j \in N-N'$ between nodes i and k by cheapest insertion and greatest angle (same as CCAO) until $ARRVT_k$ does not exceed u_k.

Step 5 . Update $cost_k$.

$cost_k = cost_i + d_i + c_{ik}$.

If $ARRVT_k < l_k$, then $cost_k = cost_k + lp_k$.

If $ARRVT_k > u_k$, then $cost_k = cost_k + up_k$.

Step 6 . Let i = k.

If N' = N, then go to next stop.

Otherwise, go to step 2.

Step 7 . Apply the Modified-Oropt procedure to the current tour. Stop when no further improvements can be fcund.

End of algorithm SCAO

This algorithm is same as SCCO except a greatest angle selection in stead of a cheapest selection in SCCO.

C. EXACT SOLUTION TECHNIQUES FOR SOFT TIME WINDOWS

1. State-Space Relaxatien Procedure

In this section we describe a state space relaxation procedure, which is adapted from Christofides et al. [Ref. 5], for soft time windows. They only ccnsidered the TSP with hard time windows and without time windows. The differences are as follows. Tne waiting cost is replaced by a penalty cost to be paid in the early arrival case. Late arrival is allowed, but a penalty cost has to be paid. So we have to calculate tne duration and the penalty cost on each possible path to decide the least cost path in each stage. We denote the penalty ccst on each possible path as PC in this section.

Consider the TSP defined on the graph G = (N,A) with scft time window constraints. Let S' be a set of all nodes except starting node. Let S be a subset of S'. Let f(S,j)

be the cost of the least cost path starting at node 1 passing through every node of S and finishing at node j. Let T(S,j) be the total duration of a path corresponding to f(S,j). Let p(S,j) be the predecessor of j on the path corresponding to f(S,j). Let lp(t) be the early arrival penalty cost function and up(t) be the late arrival penalty cost function. For a given S and j, total duration of a path can be calculated as

$$T(S,j) = [\ T(S-j,i) + d_i + c_{ij}\ ]. \qquad (4.1)$$

where $p(S,j) = i$.

In equation (4.1) total duration of the least cost path passing through the nodes in the set S and ending in node j can be described as the sum of three terms: the first is total duration of the least cost path passing through the nodes in the set S-{j} and ending in node i, the second is the time required to spend in node i, and the third is the travel time from node i to node j. The dynamic programming recursion to determine the least cost path may then be stated as

$$f(S,j) = \min_{i \in S-j} [\ f(S-j,i) + d_i + c_{ij} + PC\ ] \qquad (4.2)$$

where $T1 = [\ T(S-j,i) + d_i + c_{ij}\ ]$.

$$PC = 0 \quad , \text{if } l_j \le T1 \le u_j$$
$$= lp(l_j - T1) \ , \text{if } T1 < l_j$$
$$= up(T1 - u_j) \ , \text{if } T1 > u_j$$

with the initialization:

$$f(\{j\},j) = c_{1j} \quad , \text{if } l_j \le c_{1j} \le u_j$$
$$= c_{1j} + lp(l_j - c_{1j}) \ , \text{if } c_{1j} < l_j$$
$$= c_{1j} + up(c_{1j} - u_j) \ , \text{if } c_{1j} > u_j$$

Finally, the optimum solution can be calculated as

$$\min_{i \in S'} [\ f(S',i) + d_i + c_{i1}\ ].$$

Since the computer storage requirements are increased exponentially with the size of the problem, this method is limited to small problems. For relaxing this limitation, a state space relaxation procedure can be used same as Chapter III.

Consider the dynamic programming formulation (4.2) The state variable in that formulation is (S,j), and the stage is the cardinality of S. Let g(S) be a mapping from the domain of (S,j) to some other vector space (g(S),j). Let:

$$H(g(S),j) = \{ (g(S-j),i) \mid i \in S-j \} \tag{4.3}$$

Since we are interested in lower bounds to the TSP with time constraints, H(g(S),j) in (4.3) may be replaced by any larger set that is easier to compute. Thus, H(g(S),j) can be defined by the following equation:

$$H(g(S),j) = \{ (g(S-j),i) \mid i \in E(g(S),j) \} \tag{4.4}$$

where $S-j \subseteq E(g(S),j)$.

For calculating the lower bound of the problem, recursion (4.1) can be changed to the following equation:

$$T(g(S),j) = [\ T(g(S-j),i) + d_i + c_{ij}\ ] \tag{4.5}$$

where $p(g(S),j) = i$.

Recursion (4.2) may be stated as

$$f(g(S),j) = \min_{(g(S-j),i) \in H(g(S),j)} [f(g(S-j),i) + d_i + c_{ij} + PC] \tag{4.6}$$

$$= \min_{i \in E(g(S),j)} [f(g(S-j),i) + d_i + c_{ij} + PC] \tag{4.7}$$

where $T1 = [\ T(g(S-j),i) + d_i + c_{ij}\ ]$,

$PC = 0$, if $l_j \leq T1 \leq u_j$

$= lp(l_j - T1)$, if $T1 \leq l_j$

$= up(T1 - u_j)$, if $T1 > u_j$

with the initialization:

$$f(g(j),j) = c_{1j} \quad , \text{if } l_j \le c_{1j} \le u_j$$
$$= c_{1j} + lp(l_j - c_{1j}) \quad , \text{if } c_{1j} \le l_j$$
$$= c_{1j} + up(c_{1j} - u_j) \quad , \text{if } c_{1j} > u_j$$

Finally, the optimum solution can be calculated as

$$\min_{i \in E(g(N),1)} [\, f(g(S'),i) + d_i + c_{i1} \,].$$

The mapping can be selected from any separable function. We used a mapping function (3.7), which is proposed by Christofides et al., same as Chapter III. Then equation (4.5) becomes:

$$T(|S|,j) = [\, T(|S|-1,i) + d_i + c_{ij} \,]. \qquad (4.8)$$

where $p(|S|,j) = i$

Recursion (4.7) may be stated as:

$$f(|S|,j) = \min_{i \in E(|S|,j)} [f(|S|-1,i) + d_i + c_{ij} + PC] \qquad (4.9)$$

where $T1 = [\, T(|S|-1,i) + d_i + c_{ij} \,]$,

$$PC = 0 \quad , \text{if } l_j \le T1 \le u_j$$
$$= lp(l_j - T1) \quad , \text{if } T1 \le l_j$$
$$= up(T1 - u_j) \quad , \text{if } T1 > u_j$$

with the initialization:

$$f(1,j) = c_{1j} \quad , \text{if } l_j \le c_{1j} \le u_j$$
$$= c_{1j} + lp(l_j - c_{1j}) \quad , \text{if } c_{1j} \le l_j$$
$$= c_{1j} + up(c_{1j} - u_j) \quad , \text{if } c_{1j} > u_j$$

Finally, the optimum solution can be calculated as

$$\min_{i \in E(|N|,1)} [\, f(|S'|,i) + d_i + c_{i1} \,].$$

2. Additional Condition

In the previous section, we discussed a state space relaxation procedure which is adapted from Christofides et al.[Ref. 5]. That procedure provides a lower bound on the TSP with soft time window constraints. The additional condition to avoid loops formed by three consecutive nodes was used to get a better bound [Ref. 5]. This can be done in the following way.

Let $k = |S|$. Let $f(k,j,1)$ be the cost of the least cost path from the initial state to state (k,j) without loops formed by three consecutive nodes. Let $f(k,j,2)$ be the cost of the second least cost path from the initial state to state (k,j) without loops formed by three consecutive nodes. Let $p(k,j,m)$ be the predecessor of j on the path corresponding to $f(k,j,m)$. With the above definition, equation (4.8) becomes:

$$T(k,j,m') = [\ T(k-1,i,m) + d_i + c_{ij}\ ],\ m'=1,2 \qquad (4.10)$$

where $p(k,j,m') = i$

$m = 1$, if $p(k-1,i,1) \neq j$

$= 2$, otherwise.

With the initialization:

$$T(1,j,1) = c_{1j}$$

and

$$T(1,j,2) = \infty .$$

Recursion for $f(k,j,1)$ can be calculated in the following way. Let:

$$T'(k,j,m) = [\ T(k-1,i,m) + d_i + c_{ij}\ ],\ m=1,2$$

This gives us:

$$f(k,j,1) = \min_{i \in E(k,j)} [f(k-1,i,m) + d_i + c_{ij} + PC] \quad (4.11)$$

where $PC = 0$, if $l_j \leq T'(k,j,m) \leq u_j$

$= lp(l_j - T'(k,j,m))$, if $T'(k,j,m) \leq l_j$

$= up(T'(k,j,m) - u_j)$, if $T'(k,j,m) > u_j$

$m = 1$, if $p(k-1,i,1) \neq j$

$= 2$, otherwise.

With the initialization:

$$f(1,i,1) = c_{1i} \text{ , if } l_i \leq c_{1i} \leq u_i \quad (4.12)$$
$$= c_{1i} + lp(l_i - c_{1i}) \text{ , if } c_{1i} \leq l_i$$
$$= c_{1i} + up(c_{1i} - u_i) \text{ , if } c_{1i} > u_i$$

Recursion for $f(k,j,2)$ can be written in the following way:

$$f(k,j,2) = \min_{\substack{i \in E(k,j) \\ i \neq p(k,j,1)}} [f(k-1,i,m) + d_i + c_{ij} + PC] \quad (4.13)$$

where $PC = 0$, if $l_j \leq T'(k,j,m) \leq u_j$

$= lp(l_j - T'(k,j,m))$, if $T'(k,j,m) \leq l_j$

$= up(T'(k,j,m) - u_j)$, if $T'(k,j,m) > u_j$

$m = 1$, if $p(k-1,i,1) \neq j$

$= 2$, otherwise.

With the initialization:

$$f(1,i,2) = \infty \quad (4.14)$$

Finally, the optimum solution can be calculated as

$$\min_{i \in E(|N|,1)} [f(|S'|,i,1) + d_i + c_{i1}]. \quad (4.15)$$

Since the additional condition can avoid consideration of a useful lower bound, we considered $f(k-1,i,2)$ in recursion (4.11) and (4.13) only when the predecessor of i on the path corresponding to $f(k-1,i,1)$ is j. If we do not consider the second least cost path in case of $p(k-1,i,1)=j$, then $f(g(S),j)$ does not guarantee the lower bound of $f(S,j)$

For this example, let's consider 4 node TSP with time constraints. Node A is the starting node. D is the time free node. The lower bound of node B is 9, the upper bound of node B is 11, the lower bound of node C is 19, and the upper bound of node C is 21. Suppose service time at each node is zero, lp(t)=t, and up(t)=5t. Figure 3.7 shows an optimal route for this problem. From equation (4.10) and (4.12) we can get:

$$f(1,B,1) = 10,\ T(1,B,1) = 10,\ p(1,B,1) = A;$$

$$f(1,C,1) = 19,\ T(1,C,1) = 14.14,\ p(1,C,1) = A;$$

$$f(1,D,1) = 7.07,\ T(1,D,1) = 7.07,\ p(1,D,1) = A.$$

Now applying equation (4.10) and (4.11) recursively with i=1, for k=2 we can get:

$$f(2,B,1) = \min_{i \in \{C,D\}} [\ 94.7,\ 29.84\ ] = 29.84,$$

$$T(2,B,1) = 14.14,\ p(2,B,1) = D.$$

Similarly,

$$f(2,C,1) = 19,\ T(2,C,1) = 14.14,\ p(2,C,1) = D;$$

$$f(2,D,1) = 17.07,\ T(2,D,1) = 17.07,\ p(2,D,1) = B.$$

For k = 3,

$$f(3,B,1) = \min_{i \in \{C\}} [\ 94.7\ ] = 94.7,$$

$$T(3,B,1) = 24.14,\ p(3,B,1) = C.$$

Similarly,

$$f(3,C,1) = 39.84,\ T(3,C,1) = 24.14,\ p(3,C,1) = D;$$

$$f(3,D,1) = \infty .$$

We can see easily that f(3,D,1) is not a lower bound of f({B,C,D},D).

3. Branch and Bound Procedure

We used the same branch and bound procedure used to eliminate subtours in the solution of the state space relaxation procedure in Chapter III.C.3. The SCCO heuristic, which was described in section B.2, was used as an initial upper bound, and the lower bound was obtained from equation (4.15).

We present the results of our computational experience with the algorithms of this Chapter in the next Chapter.

V. COMPUTATIONAL EXPERIENCE

A. TEST PROBLEMS

Four sets of test data are used in this thesis. Test problem number [1] is taken from Sedgewick [Ref.19: p.309]. The other problems, numbered [2], [3] and [4], are from Appendix 9.1 of Eilon et al.'s text [Ref. 21]. These test problems are shown in Appendices A,B,C respectively. These published problems contain node and depot locations, but they do not include time windows.

We constructed time windows for test problems [1],[2],[3] by first using the CCAO heuristic on the unconstrained TSP. Time windows were then placed about each node such that the CCAO route was feasible. The idea for generating time windows in this way comes from Baker [Ref. 25], who used the unconstrained Nearest Neighbor heuristic as his starting point instead of CCAO.

The time window widths were set to varying sizes ranging from 3 to 14. Some of the time windows were fairly tight while others overlapped. This is in contrast to Baker's work, where all the time windows have width equal 2 units.

The last problem number [4] is the same as test problem [3], except that the time windows were constructed from a Nearest Neighbor solution to the unconstrained traveling salesman problem, as in Baker [Ref. 25]. Figure 5.1 displays the CCAO solution for test problem [3] and Figure 5.2 illustrates the unconstrained Nearest Neighbor solution for the test problem [4]. We found a small error in Baker's TSP solution for the Nearest Neighbor [Ref. 25], in that the nearest node from node 16 is node 17, not node 13. The resulting cost is actually higher, it is 312.09, not 310.22.

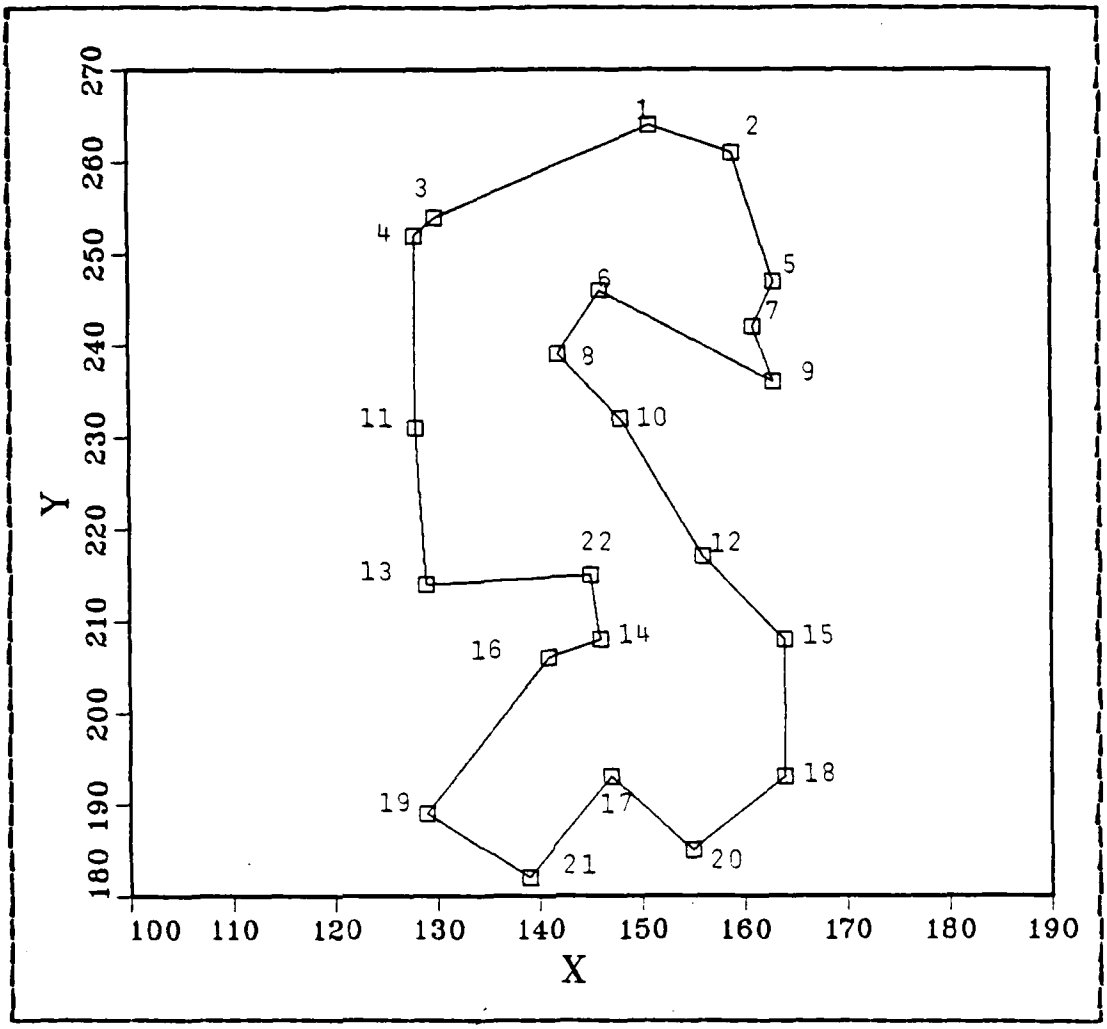


Figure 5.1 Unconstrained Solution Obtained by CCAO.

Each of the four sets of test data was used to create four test problems. The separate instances differed in the percentages of time window constraints that were chosen to be in effect. The four cases were 100%, 90%, 75%, and 50%. We refer to this percentage as the "time-window percentage".

A random number generator was used to decide which nodes would have time windows. Test problems for the time window constrained TSP are shown in Appendices G through V. The

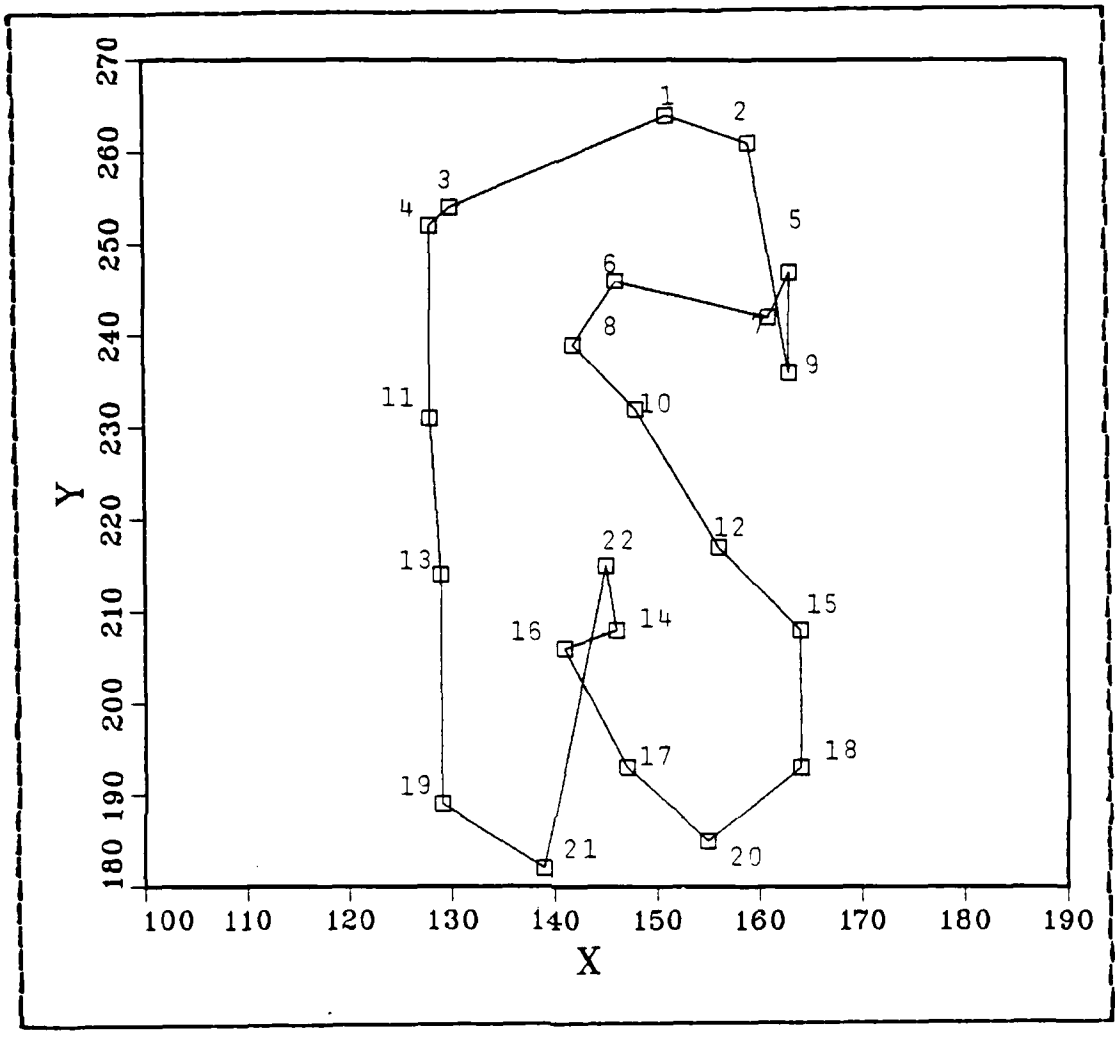


Figure 5.2 Unconstrained Solution Obtained by Nearest Neighbor Heuristic.

penalty cost factors can be varied depending on real world problems. We used 2 and 5 as the lower and upper penalty cost factor. Also we set the service time at each node to 0 to make it easy to construct the time windows.

The computational results are presented in Tables II and III. The figures reported represent results of our test runs for each test case.

B. COMPUTATIONAL RESULTS

1. Hard Time Windows

As noted in section 3, the Nearest Neighbor heuristic often cannot solve the problem, because the arrival time of the nearest node frequently violates the upper bound. However in test problem [4], the results of the Nearest Neighbor are the same as in the unconstrained problem, because this problem itself was constructed by a Nearest Neighbor heuristic.

Generally, the SCCO and SCAO heuristic can be easily applied to the hard time window TSP. According to our experiments, if the time window width becomes large relative to the travel time between nodes on the optimal unconstrained TSP route, then the lower percentage time window problems become more difficult to satisfy. This phenomenon can be seen in test problem [1]. That is because the other nodes in the optimal route for the unconstrained TSP problem could be inserted without causing violation of the upper bound.

The SIACK heuristic takes slightly more time than the other heuristics. It achieved lower accuracy in test problems [1] and [3] in the 50 percent time window case. The exact algorithm can find the exact answer in most problems, but when there are fewer windows in effect, it takes more computation time. It cannot solve the 50 percent time window problems [2] and [4] within 180 seconds.

TABLE II

COMPUTATIONAL RESULTS OF THE HARD TIME WINDOWS

Problem	Number of nodes	Time Windows in Effect	Nearest Neighbor		SCCO		SCAO		SLACK		Exact	
			cost	CPU	cost	CPU	cost	CPU	cost	CPU	cost	CPU
{1}	16	15	69.766	.017	66.604	.017	66.604	.017	66.604	.020	66.604	.017
	16	14	69.766	.007	66.604	.007	66.604	.017	66.604	.017	66.604	.027
	16	12	-		66.604	.010	66.604	.013	66.604	.017	66.604	.070
	16	8	-		89.095	.020	80.995	.013	79.048	.010	66.604	.043
{2}	22	21	469.03	.013	469.03	.017	469.03	.007	469.03	.017	469.03	.057
	22	19	-		469.03	.010	469.03	.013	469.03	.020	469.03	.186
	22	15	-		469.03	.013	469.03	.017	469.03	.013	469.03	33.5
	22	11	-		469.03	.017	469.03	.013	469.03	.020	-	
{3}	22	21	278.44	.017	278.44	.013	278.44	.013	278.44	.023	278.44	.030
	22	19	-		278.44	.010	278.44	.017	278.44	.023	278.44	.067
	22	15	-		278.44	.020	278.44	.017	278.44	.020	278.44	.070
	22	11	-		278.44	.013	278.44	.013	337.29	.023	278.44	.626
{4}	22	21	312.09	.017	312.09	.020	312.09	.017	312.09	.013	312.09	.063
	22	19	312.09	.017	312.09	.017	312.09	.017	312.09	.020	312.09	.063
	22	15	312.09	.010	308.32	.013	308.32	.010	308.32	.020	308.32	25.3
	22	11	312.09	.020	312.09	.013	312.09	.010	312.09	.017	-	

* CPU times in seconds on IBM 3033.

2. Soft Time Windows

All of the methods tested for soft time windows were able to find some answer to every problem within reasonable computing time, except for two instances with the exact algorithm. With the Nearest Neighbor heuristic, the quality of solution is not desirable. In general, the lower time window percentage problems have lower solution quality. As in the hard window case, on test problem [4], the results of the Nearest Neighbor heuristic coincide with the unconstrained TSP heuristic, because this problem itself was constructed by a Nearest Neighbor.

As in the hard time window problem, SCCO and SCAO generally find an optimal solution except for one problem with 50 percent time windows. In test problem [1] with 50 percent time windows, the SCCO and SCAO values were 215.686, 165.544 respectively. The exact algorithm could not solve the two test problems with 50 percent time windows within 180 seconds. The reason is that the solutions of the state space relaxations have many subtours and it takes a long time to eliminate these subtours.

With both hard and soft time windows, the results are sensitive to the percentage, width and position of the time windows. In most problems, the fewer time windows there are, the lower the accuracy of the heuristics.

TABLE III

COMPUTATIONAL RESULTS OF THE SOFT TIME WINDOWS

Problem	Number of nodes	Time Windows in Effect	Nearest Neighbor		SCCO		SCAO		Exact	
			cost	CPU	cost	CPU	cost	CPU	cost	CPU
{1}	16	15	69.766	.007	66.604	.007	66.604	.017	66.604	.063
	16	14	69.766	.017	66.604	.007	66.604	.007	66.604	.040
	16	12	318.758	.013	66.604	.017	66.604	.017	66.604	.176
	16	8	202.150	.017	215.686	.013	165.544	.020	66.604	.060
{2}	22	21	469.029	.010	469.029	.007	469.029	.007	469.029	.183
	22	19	947.088	.020	469.029	.020	469.029	.013	469.029	.170
	22	15	995.020	.020	469.029	.013	469.029	.013	469.029	66.2
	22	11	1199.196	.020	469.029	.020	469.029	.020	-	
{3}	22	21	278.437	.013	278.437	.017	278.437	.007	278.437	.190
	22	19	614.950	.007	278.437	.007	278.437	.010	278.437	.200
	22	15	1044.347	.007	278.437	.020	278.437	.017	278.437	.190
	22	11	1406.414	.023	278.437	.007	278.437	.007	278.437	1.67
{4}	22	21	312.089	.013	312.089	.007	312.089	.020	312.089	.180
	22	19	312.089	.020	312.089	.020	312.089	.023	312.089	.183
	22	15	312.089	.010	308.321	.017	308.321	.020	308.321	58.4
	22	11	312.089	.010	312.089	.017	312.089	.017	-	

* CPU times in seconds on IBM 3033.

VI. CONCLUSIONS AND RECOMMENDATIONS

This thesis has presented some heuristics and exact algorithms for the solution of traveling salesman problem with time window constraints. We considered two different kinds of time window constraints : hard time windows and soft time windows. Hard time windows are inviolable, whereas soft windows may be violated at a cost.

For both hard time windows and soft time windows, we developed some new heuristics, SCCC and SCAO, which are modifications of Stewart's unconstrained TSP heuristics [Ref. 6] CCCO and CCAO. Also for the hard time window only, we developed the SLACK heuristic. We also developed an exact algorithm for both hard and soft window using state space relaxation dynamic programming and branch and bound as proposed by Christofides et al. [Ref. 5].

The procedures were shown to be effective on some moderately small sized problems. A Nearest Neighbor heuristic was also developed, but it was often unable to solve the problem with hard time windows, and it found very low quality solutions with soft time windows. This experience is consistent with the findings of others [Ref. 7] who determined that the Nearest Neighbor heuristic does not perform well on the unconstrained TSP.

The SCCO and SCAO are generally effective on most of the small sized problems we tested, except for the problems in which less than half the nodes have time windows. Further research is needed in order to satisfactorily solve these problems. Another problem difficulty that may require more research is dealing with wider time windows.

The SLACK heuristic which is used only with hard time windows is slightly slower than the other heuristics.

Particularly, in the lower time window percentage problems, the accuracy becomes lower.

The exact algorithm succeeded in solving 14 of the 16 test problems to optimality, but it was too slow to use in most of the lower time window percentage problems. This algorithm's performance also depends upon the quality of the upper bound which is obtained from the heuristic. Additional research is needed to reduce computation time, but a working program for at least some problems has resulted from this effort.

APPENDIX A

TEST PROBLEM [1]

node	x	y	node	x	y
1	3	9	11	10	13
2	11	1	12	16	14
3	6	8	13	15	2
4	4	3	14	13	16
6	8	11	16	12	10
7	6	4			
8	7	4			
9	9	7			
10	14	5			

Depot cc-ordinates : (12,10)

problem source : Sedgewick [Ref. 19]

APPENDIX B
TEST PROBLEM [2]

node	x	y	node	x	y
1	295	272	12	267	242
2	301	258	13	259	265
3	309	260	14	315	233
4	217	274	15	329	252
5	218	267	16	318	252
6	282	267	17	329	224
7	242	249	18	267	213
8	230	262	19	275	192
9	249	268	20	303	201
10	256	267	21	208	217
11	265	257	22	326	181

Depot co-ordinates : (326,181)

problem source : Eilon et al. [Ref. 21]

APPENDIX C
TEST PROBLEM [3]

node	x	y	node	x	y
1	151	264	12	156	217
2	159	261	13	129	214
3	130	254	14	146	208
4	128	252	15	164	208
5	163	247	16	141	206
6	146	246	17	147	193
7	161	242	18	164	193
8	142	239	19	129	189
9	163	236	20	155	185
10	148	232	21	139	182
11	128	231	22	145	215

Depot co-ordinates : (145,215)

problem source : Eilon et al. [Ref. 21].

APPENDIX D
TEST PROBLEM [5]

node	x	y	node	x	y	node	x	y	node	x	y
1	37	52	14	12	42	27	30	48	40	5	6
2	49	49	15	36	16	28	43	67	41	10	17
3	52	64	16	52	41	29	58	48	42	21	10
4	20	26	17	27	23	30	58	27	43	5	64
5	40	30	18	17	33	31	37	69	44	30	15
6	21	47	19	13	13	32	38	46	45	30	10
7	17	63	20	57	58	33	46	10	46	32	39
8	31	62	21	62	42	34	61	33	47	25	32
9	52	33	22	42	57	35	62	63	48	25	55
10	51	21	23	16	57	36	63	69	49	48	28
11	42	41	24	8	52	37	32	22	50	56	37
12	31	32	25	7	38	38	45	35			
13	5	25	26	27	68	39	59	15			

Depot co-ordinates : (30,40)

problem source : Eilon et al. [Ref .21].

APPENDIX E

TEST PROBLEM [6]

node	x	y	node	x	y	node	x	y	node	x	y
1	22	22	20	66	14	39	30	60	58	40	60
2	36	26	21	44	13	40	30	50	59	70	64
3	21	45	22	26	13	41	12	17	60	64	4
4	45	35	23	11	28	42	15	14	61	36	6
5	55	20	24	7	43	43	16	19	62	30	20
6	33	34	25	17	64	44	21	48	63	20	30
7	50	50	26	41	46	45	50	30	64	15	5
8	55	45	27	55	34	46	51	42	65	50	70
9	26	59	28	35	16	47	50	15	66	57	72
10	40	66	29	52	26	48	48	21	67	45	42
11	55	65	30	43	26	49	12	38	68	38	33
12	35	51	31	31	76	50	15	56	69	50	4
13	62	35	32	22	53	51	29	39	70	66	8
14	62	57	33	26	29	52	54	38	71	59	5
15	62	34	34	50	40	53	55	57	72	35	60
16	21	36	35	55	50	54	67	41	73	27	24
17	33	44	36	54	10	55	10	70	74	40	20
18	9	56	37	60	15	56	6	25	75	40	37
19	62	48	38	47	66	57	65	27			

Depot co-ordinates : (40,40)

problem source : Eilon et al. [Ref .21].

APPENDIX F

TEST PROBLEM FOR THE SCCO

node	x	y	time window l(i)	time window u(i)	node	x	y	time window l(i)	time window u(i)
1	3	9	-	-	11	10	13	10	17
2	11	1	-	-	12	16	14	2	9
3	6	8	27	36	13	15	2	-	-
4	4	3	37	45	14	13	16	5	13
5	5	15	-	-	15	2	12	-	-
6	8	11	12	23	16	12	10	-	-
7	6	4	35	43					
8	7	4	42	49					
9	9	7	58	68					
10	14	5	53	64					

Depot co-ordinates : (12,10)

problem source

node locations : Sedgewick [Ref. 19]

time windows : see Chapter V.

APPENDIX G
TEST PROBLEM [1-1]

node	x	y	time window l(i)	time window u(i)	node	x	y	time window l(i)	time window u(i)
1	3	9	25	32	11	10	13	10	17
2	11	1	46	53	12	16	14	2	9
3	6	8	27	36	13	15	2	51	59
4	4	3	37	45	14	13	16	5	13
5	5	15	18	28	15	2	12	22	30
6	8	11	14	23	16	12	10	-	-
7	6	4	35	43					
8	7	4	42	49					
9	9	7	58	68					
10	14	5	53	64					

Depot co-ordinates : (12,10)

CL = 2.0, CU = 5.0

problem source

node locations : Sedgewick [Ref. 19]

time windows : see Chapter V.

APPENDIX H

TEST PROBLEM [1-2]

node	x	y	time window l(i)	time window u(i)	node	x	y	time window l(i)	time window u(i)
1	3	9	25	32	11	10	13	10	17
2	11	1	46	53	12	16	14	2	9
3	6	8	27	36	13	15	2	51	59
4	4	3	-	-	14	13	16	5	13
5	5	15	18	28	15	2	12	22	30
6	8	11	14	23	16	12	10	-	-
7	6	4	35	43					
8	7	4	42	49					
9	9	7	58	68					
10	14	5	53	64					

Depot co-ordinates : (12,10)

CL = 2.0, CU = 5.0

problem source

node locations : Sedgewick [Ref. 19]

time windows : see Chapter V.

APPENDIX I

TEST PROBLEM [1-3]

node	x	y	time window l(i)	time window u(i)	node	x	y	time window l(i)	time window u(i)
1	3	9	-	-	11	10	13	10	17
2	11	1	46	53	12	16	14	2	9
3	6	8	27	36	13	15	2	51	59
4	4	3	37	45	14	13	16	5	13
5	5	15	18	28	15	2	12	22	30
6	8	11	14	23	16	12	10	-	-
7	6	4	35	43					
8	7	4	42	49					
9	9	7	-	-					
10	14	5	-	-					

Depot co-ordinates : (12,10)

CL = 2.0, CU = 5.0

problem source

node locations : Sedgewick [Ref. 19]

time windows : see Chapter V.

APPENDIX J

TEST PROBLEM [1-4]

node	x	y	time window l(i)	time window u(i)	node	x	y	time window l(i)	time window u(i)
1	3	9	25	32	11	10	13	10	17
2	11	1	46	53	12	16	14	-	-
3	[illegible]	8	-	-	13	15	2	51	59
4	4	3	37	45	14	13	16	5	13
5	5	15	-	-	15	2	12	-	-
6	8	11	14	23	16	12	10	-	-
7	6	4	-	-					
8	7	4	-	-					
9	9	7	58	68					
10	14	5	-	-					

Depot Co-ordinates : (12,10)

CL = 2.0, CU = 5.0

Problem Source

node locations : Sedgewick [Ref. 19]

time windows : see Chapter V.

APPENDIX K
TEST PROBLEM [2-1]

node	x	y	time window l(i)	time window u(i)	node	x	y	time window l(i)	time window u(i)
1	295	272	125	135	12	267	242	170	179
2	301	258	110	118	13	259	265	193	202
3	309	260	102	110	14	315	233	57	67
4	217	274	242	250	15	329	252	81	89
5	218	278	239	246	16	318	252	90	98
6	282	267	141	149	17	329	224	40	49
7	242	249	279	286	18	267	213	382	393
8	230	262	261	271	19	275	192	404	413
9	249	268	206	215	20	303	201	432	442
10	256	267	200	208	21	208	217	323	332
11	265	257	183	193	22	326	181	-	-

Depot co-ordinates : (326,181)

CL = 2.0, CU = 5.0

problem source

node locations : Eilon et al. [Ref. 21].

time windows : see Chapter V.

APPENDIX L
TEST PROBLEM [2-2]

node	x	y	time window l(i)	time window u(i)	node	x	y	time window l(i)	time window u(i)
1	295	272	125	135	12	267	242	170	179
2	301	258	110	118	13	259	265	-	-
3	309	260	102	110	14	315	233	57	67
4	217	274	242	250	15	329	252	81	89
5	218	278	239	246	16	318	252	90	98
6	282	267	141	149	17	329	224	40	49
7	242	249	279	286	18	267	213	382	393
8	230	262	261	271	19	275	192	404	413
9	249	268	206	215	20	303	201	432	442
10	256	267	200	208	21	208	217	-	-
11	265	257	183	193	22	326	181	-	-

Depot co-ordinates : (326,181)

CL = 2.0, CU = 5.0

problem source

node locations : Eilon et al. [Ref. 21]

time windows : see Chapter V .

APPENDIX H

TEST PROBLEM [2-3]

node	x	y	time window l(i)	time window u(i)	node	x	y	time window l(i)	time window u(i)
1	295	272	125	135	12	267	242	170	179
2	301	258	-	-	13	259	265	-	-
3	309	260	-	-	14	315	233	57	67
4	217	274	242	250	15	329	252	81	89
5	218	278	239	246	16	318	252	90	98
6	282	267	141	149	17	329	224	40	49
7	242	249	-	-	18	267	213	382	393
8	230	262	-	-	19	275	192	-	-
9	249	268	206	215	20	303	201	432	442
10	256	267	200	208	21	208	217	323	332
11	265	257	183	193	22	326	181	-	-

Depot co-ordinates : (326,181)

CL = 2.0, CU = 5.0

problem source

node locations : Eilon et al. [Ref. 21].

time windows : see Chapter V.

APPENDIX N
TEST PROBLEM [2-4]

node	x	y	time window l(i)	time window u(i)	node	x	y	time window l(i)	time window u(i)
1	295	272	-	-	12	267	242	170	179
2	301	258	110	118	13	259	265	-	-
3	309	260	-	-	14	315	233	57	67
4	217	274	242	250	15	329	252	-	-
5	218	278	239	246	16	318	252	90	98
6	282	267	141	149	17	329	224	-	-
7	242	249	279	286	18	267	213	-	-
8	230	262	-	-	19	275	192	404	413
9	249	268	206	215	20	303	201	432	442
10	256	267	-	-	21	208	217	-	-
11	265	257	-	-	22	326	181	-	-

Depot co-ordinates : (326,181)

CL = 2.0, CU = 5.0

problem source

node locations : Eilon et al. [Ref. 21].

time windows : see Chapter V.

APPENDIX Q

TEST PROBLEM [3-1]

node	x	y	time window l(i)	time window u(i)	node	x	y	time window l(i)	time window u(i)
1	151	264	196	204	12	156	217	105	118
2	159	261	185	193	13	129	214	259	271
3	130	254	217	225	14	146	208	2	10
4	128	252	222	234	15	164	208	92	105
5	163	247	174	185	16	141	206	10	19
6	146	246	142	154	17	147	193	54	68
7	161	242	166	173	18	164	193	79	89
8	142	239	131	142	19	129	189	30	38
9	163	236	159	165	20	155	185	67	75
10	148	232	123	131	21	139	182	40	53
11	128	231	242	253	22	145	215	-	-

Depot co-ordinates : (145,215)

CL = 2.0 , CU = 5.0

problem source

node locations : Eilon et al. [Ref. 21].

time windows : see Chapter V.

APPENDIX P
TEST PROBLEM [3-2]

node	x	y	time window l(i)	time window u(i)	node	x	y	time window l(i)	time window u(i)
1	151	264	196	204	12	156	217	105	118
2	159	261	185	193	13	129	214	-	-
3	130	254	217	225	14	146	208	2	10
4	128	252	222	234	15	164	208	92	105
5	163	247	174	185	16	141	206	10	19
6	146	246	142	154	17	147	193	54	68
7	161	242	166	173	18	164	193	79	89
8	142	239	131	142	19	129	189	30	38
9	163	236	159	165	20	155	185	67	75
10	148	232	123	131	21	139	182	-	-
11	128	231	242	253	22	145	215	-	-

Depot co-ordinates : (145,215)

CL = 2.0 , CU = 5.0

problem source

node locations : Eilon et al [Ref. 21].

time windows : see Chapter V.

APPENDIX Q
TEST PROBLEM [3-3]

node	x	y	time window l(i)	time window u(i)	node	x	y	time window l(i)	time window u(i)
1	151	264	196	204	12	156	217	105	118
2	159	261	-	-	13	129	214	-	-
3	130	254	-	-	14	146	208	2	10
4	128	252	222	234	15	164	208	92	105
5	163	247	174	185	16	141	206	10	19
6	146	246	142	154	17	147	193	54	68
7	161	242	-	-	18	164	193	79	89
8	142	239	-	-	19	129	189	-	-
9	163	236	159	165	20	155	185	67	75
10	148	232	123	131	21	139	182	40	53
11	128	231	242	253	22	145	215	-	-

Depot co-ordinates : (145,215)

CL = 2.0 , CU = 5.0

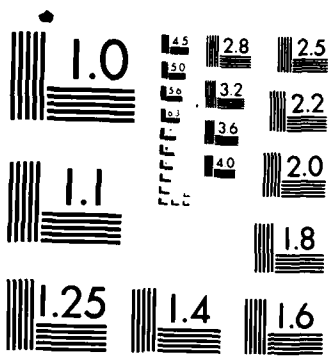
problem source

node locations : Eilon et al. [Ref. 21].

time windows : see Chapter V.

AD-A162 118 ALGORITHMS AND HEURISTICS FOR TIME-WINDOW-CONSTRAINED TRAVELING SALESMAN PROBLEMS(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA B J CHUN ET AL SEP 85 2/2

UNCLASSIFIED F/G 12/1 NL



APPENDIX B

TEST PROBLEM [3-4]

node	x	y	time window l(i)	time window u(i)	node	x	y	time window l(i)	time window u(i)
1	151	264	-	-	12	156	217	105	118
2	159	261	185	193	13	129	214	-	-
3	130	254	-	-	14	146	208	2	10
4	128	252	222	234	15	164	208	-	-
5	163	247	174	185	16	141	206	10	19
6	146	246	142	154	17	147	193	-	-
7	161	242	166	173	18	164	193	-	-
8	142	239	-	-	19	129	189	30	38
9	163	236	159	165	20	155	185	67	75
10	148	232	-	-	21	139	182	-	-
11	128	231	-	-	22	145	215	-	-

Depot co-ordinates : (145,215)

CL = 2.0 , CU = 5.0

problem source

node locations : Eilon et al. [Ref. 21].

time windows : see Chapter V.

APPENDIX S
TEST PROBLEM [4-1]

node	x	y	time window l(i)	time window u(i)	node	x	y	time window l(i)	time window u(i)
1	151	264	171	179	12	156	217	72	79
2	159	261	162	170	13	129	214	237	245
3	130	254	196	203	14	146	208	5	9
4	128	252	198	206	15	164	208	61	67
5	163	247	128	136	16	141	206	10	14
6	146	246	106	113	17	147	193	22	28
7	161	242	122	130	18	164	193	48	53
8	142	239	97	105	19	129	189	261	269
9	163	236	138	146	20	155	185	35	40
10	148	232	89	96	21	139	182	273	280
11	128	231	220	227	22	145	215	-	-

Depot co-ordinates : (145,215)

CL = 2.0, CU = 5.0

problem source

node locations : Eilon et al [Ref. 21].

time windows : see Chapter V.

APPENDIX T
TEST PROBLEM [4-2]

node	x	y	time window l(i)	time window u(i)	node	x	y	time window l(i)	time window u(i)
1	151	264	171	179	12	156	217	72	79
2	159	261	162	170	13	129	214	-	-
3	130	254	196	203	14	146	208	5	9
4	128	252	198	206	15	164	208	61	67
5	163	247	128	136	16	141	206	10	14
6	146	246	106	113	17	147	193	22	28
7	161	242	122	130	18	164	193	48	53
8	142	239	97	105	19	129	189	261	269
9	163	236	138	146	20	155	185	35	40
10	148	232	89	96	21	139	182	-	-
11	128	231	220	227	22	145	215	-	-

Depot co-ordinates : (145,215)

CL = 2.0, CU = 5.0

problem source

node locations : Eilon et al. [Ref. 21].

time windows : see Chapter V.

APPENDIX U

TEST PROBLEM [4-3]

node	x	y	time window l(i)	time window u(i)	node	x	y	time window l(i)	time window u(i)
1	151	264	171	179	12	156	217	72	79
2	159	261	-	-	13	129	214	-	-
3	130	254	-	-	14	146	208	5	9
4	128	252	198	206	15	164	208	61	67
5	163	247	128	136	16	141	206	10	14
6	146	246	106	113	17	147	193	22	28
7	161	242	-	-	18	164	193	48	53
8	142	239	-	-	19	129	189	-	-
9	163	236	138	146	20	155	185	35	40
10	148	232	89	96	21	139	182	273	280
11	128	231	220	227	22	145	215	-	-

Depot co-ordinates : (145,215)

CL = 2.0, CU = 5.0

problem source

node locations : Eilon et al. [Ref. 21]

time windows : see Chapter V.

APPENDIX V

TEST PROBLEM [4-4]

node	x	y	time window l(i)	time window u(i)	node	x	y	time window l(i)	time window u(i)
1	151	264	-	-	12	156	217	72	79
2	159	261	162	170	13	129	214	-	-
3	130	254	-	-	14	146	208	5	9
4	128	252	198	206	15	164	208	-	-
5	163	247	128	136	16	141	206	10	14
6	146	246	106	113	17	147	193	-	-
7	161	242	122	130	18	164	193	-	-
8	142	239	-	-	19	129	189	261	269
9	163	236	138	146	20	155	185	35	40
10	148	232	-	-	21	139	182	-	-
11	128	231	-	-	22	145	215	-	-

Depot co-ordinates : (145,215)

CL = 2.0, CU = 5.0

problem source

node locations : Eilon et al. [Ref. 21].

time windows : see Chapter V.

LIST OF REFERENCES

1. Garey, M. R., Graham, R. L., and Johnson, P. S., "Some NP-complete Geometric Problems," Proce. 8th ACM Symp. on Theory of Computing, 1976.

2. Lenstra, J. K. and Rinnooy Kan, A. H. G., " Complexity of Vehicle Routing and Scheduling Problems," Networks, Vol. 11, pp. 221-227, 1981.

3. Psaraftis, H. N., "A Dynamic programming Solution to the Single Vehicle Many-to-Many Immediate Request Dial-A-Ride Problem," Transportation Science, Vol. 14, No. 2, pp 130-154, 1980.

4. Baker, E. K., "An Exact Algorithm for the Time-Constrained Traveling Salesman Problem", Operations Research, Vol. 31, No. 5, pp. 938-945, September-October, 1983.

5. Christofides, N., Mingozzi, A., and Toth, P., "State-Space Relaxation Procedures for the Computation of Bounds to Routing Problems," Networks, Vol. 11, No. 2, pp. 145-164, 1981.

6. Stewart, W. R., New Algorithms for Deterministic and Stochastic Vehicle Routing Problems, Doctorial Dissertation, College of Business and Management, University of Maryland, 1981.

7. Golden, B., Bodin, L., Doyle, T., and Stewart, W. R., " Approximate Traveling Salesman Problem," Operations Research, Vol. 28, pp. 694-711, 1980.

8. Rosenkrantz, D. J., Stearns, R. E., and Lewis, P. M., "Approximate Algorithms for the Traveling Salesman Problem," SIAM Journal on Computing, V.6, pp. 563-581, 1977.

9. Clarke, G. and Wright, S. W., " Scheduling of Vehicles from A Central Depot to A Number of Delivery Points," Operations Research, Vol. 12, pp. 568-581, 1964

10. Wiorkowski, J. and McElvain, K., " A Rapid Heuristic Algorithm for the Approximate Solution of the Traveling Salesman Problem," Trans Research, Vol. 9, pp. 181-185 , 1975.

11. Or, I., Traveling Salesman-Type Combinatorial Problems and Their Relation to the Logistics of Blood Banking, PH.D. Thesis, Dept.of Industrial Engineering and Management Sciences, Northwestern University, 1976.

12. Stewart, W. R., "A Computationally Efficient Heuristic for the Traveling Salesman Problem," Proceedings Thirteenth Annual Meeting of Southeastern TIMS, Myrtle Beach, S.C., pp. 75-85, 1977.

13. Norback, J. P. and Love, R. F., "Heuristic for the Hamiltonian Path Problem in Euclidean Two Space," Operations Research V. 30, pp. 363-368, 1979.

14. Hardgrave, W. W. and Nemhauser, G. L., "On The Relating Between The Traveling Salesman Problem and The Longest Path problem," Operations Research, V.10, pp. 647-657, 1962.

15. Lin, S., " Computer Solutions of the Traveling Salesman Problem," Bell Syst. Tech, J. 44, pp. 2245-2269, 1965.

16. Lin, S. and Kernighan, B., "An Effective Heuristic Algorithm for the Traveling Salesman Problem," Operations Research, Vol. 21, pp. 498-516, 1973.

17. Golden, B., "A Statistical Approach to TSP," Networks, Vol. 7, pp. 209-225, 1977.

18. Norback, J. P. and Love, R. F., "Geometric Approaches to Solving The Traveling Salesman Problem," Management Science, V. 23, pp. 1208-1223, 1977.

19. Sedgewick, R., Algorithms, Addison-Wesley, Menlo Park, Califonia, pp. 307-332, 1983.

20. Aho, A. V., Hopcroft, J. E. and Ullman, J. D., The Design and Analysis of Computer Algorithms, Addison-Wesley, Menlo park, Califonia, pp. 87-92, Jun, 1974.

21. Eilon, S., Watson-Gandy, C. and Christofides, N., Distribution Management, Griffin Press, London, pp. 113-149, 1971.

22. Fisher, M. L., " The Lagrangean Relaxation Method for Solving Integer Programming Problems," Management Science, Vol. 27, No 1, pp. 1-17, Jan, 1981.

23. Garfinkel, R. S. and Nemhauser, G. L., Integer Programming, John wiley, New York, pp. 108-152, pp. 354-360, 1972.

24. Christofides, N., Graph Theory, Academic Press, New York, pp. 390-395, pp. 236-281, 1975.

25. Baker, E. K., "Vehicle Routing with Time Window Constraints," The Logistics and Transportation Review, Vol. 18, number 4, pp. 385-401, 1982.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, CA 93943-5100	2
3.	Department Chairman, Code 55 Department of Operations Research Naval Postgraduate School Monterey, CA 93943-5100	1
4.	Professor Richard E. Rosenthal Code 55Rl Naval Postgraduate School Department of Operations Research Monterey, CA 93943-5100	2
5.	Professor James K. Hartman Code 55Hl Department of Operations Research Naval Postgraduate School Monterey, CA 93943-5100	1
6.	Library, P.O.Box 77 Gong Neung Dong, Dobong-ku Seoul 130-09, Korea	1
7.	Library Air Force Academy Dae Bang Dong, Dongjak-ku Seoul 151-01, Korea	1
8.	Air Force Library P.O.Box 6 Sin Dae Bang Dong, Dongjak-ku Seoul 151-01, Korea	1
9.	Major. Chun, Bock Jin 382-01 Sun Hwa 1 Dong, Chung-Ku Dae Jeon, Choong Nam 300-00, Korea	7
10.	Major. Lee, Sang Heon 248-16, 19 Tong 2 Ban Kaneung-1 Dong, Euijeongbu-si Kycungki 130-30, Seoul Korea	7
11.	Chow Kay Cheong Apt Block 291A Jurong East Street 21 # 12-583, Singapore (0140)	1
12.	Major. Min, Byung Ho 1086 Hamilton Ave. #B Seaside, CA 93955	1

END

FILMED

1-86

DTIC