UCRL-LR--105145

DE91 004943

The Cost-Constrained Traveling Salesman Problem

Padmini R. Sokkappa
(Ph.D. Thesis)

Manuscript date: October 1990

LAWRENCE LIVERMORE NATIONAL LABORATORY
University of California • Livermore, California • 94551

ABSTRACT

The Cost-Constrained Traveling Salesman Problem (CCTSP) is a variant of the well-known Traveling Salesman Problem (TSP). In the TSP, the goal is to find a tour of a given set of cities such that the total cost of the tour is minimized. In the CCTSP, each city is given a value, and a fixed cost-constraint is specified. The objective is to find a subtour of the cities that achieves maximum value without exceeding the cost-constraint. Thus, unlike the TSP, the CCTSP requires both selection and sequencing. As a consequence, most results for the TSP cannot be extended to the CCTSP. We show that the CCTSP is NP-hard and that no K-approximation algorithm or fully polynomial approximation scheme exists, unless P = NP. We also show that several special cases are polynomially solvable.

Algorithms for the CCTSP, which outperform previous methods, were developed in three areas: upper bounding methods, exact algorithms, and heuristics. Extensive computational studies were undertaken to evaluate and compare algorithms. These computational studies also examined the sensitivity of performance to problem characteristics. We found that a bounding strategy based on the knapsack problem performs better, both in speed and in the quality of the bounds, than methods based on the assignment problem. Likewise, we found that a branch-and-bound approach using the knapsack bound was superior to a method based on a common branch-and-bound method for the TSP. In our study of heuristic algorithms, we found that, when selecting nodes for inclusion in the subtour, it is important to consider the "neighborhood" of the nodes. A node with low value that brings the subtour near many other nodes may be more desirable than an isolated node of high value. We found two types of repetition to be desirable: repetitions based on randomization in the subtour building process, and repetitions encouraging the inclusion of different subsets of the nodes. By varying the number and type of repetitions, we can adjust the computation time required by our method to obtain algorithms that outperform previous methods in both speed and solution quality.

ACKNOWLEDGMENTS

Now that the many years of work are over, I wish to thank all those who provided me with support and assistance. Unfortunately, there are far too many to acknowledge each by name. I would, however, like to name those who made the largest contributions.

First, let me thank my dissertation reading committee. Professor Frederick Hillier, my dissertation advisor, provided invaluable guidance, comments, and encouragement. The other committee members — Professors Richard Cottle and George Dantzig — provided many useful comments, which greatly improved the final product. Special thanks go to Professor Dantzig, who was my first research advisor. He remained a firm believer and supporter, even during my worst periods of nonproductivity.

My dissertation research was funded by the Electrical Engineering Department at Lawrence Livermore National Laboratory. It could not have been completed without their support. I appreciate the efforts of all who helped me obtain this funding: in particular, Rokaya Al-Ayat, Stein Weissenberger, and David Pehrson. Also, I have not forgotten ATT-Bell Laboratories, who were responsible for sending me to Stanford in the first place and funded my first year.

I would like to express my sincere appreciation to several individuals at Lawrence Livermore National Laboratory without whose support I never would have finished this dissertation. Rokaya Al-Ayat provided great assistance and encouragement in finding my dissertation topic and seeing it through. I am indebted to Scott Strait for his continuous encouragement and understanding and for the great motivation he provided by setting a fine example and proving that it could be done. Without the assistance of Cherie Jo Patenaude and Amanda Goldner, I probably would still be sitting at my computer, trying to get my first program to run. Alan Sicherman, who always got excited about the slightest result, provided me with special inspiration.

I would also like to acknowledge Dr. Margaret Schaefer, who was my undergraduate thesis advisor at the College of William and Mary. The experience I obtained doing that thesis was of great value in this endeavor.

Finally, thanks to all my friends and colleagues (Bruce Judd, Debbie Halpern, Curtis Eaves, and countless others) who supplied encouragement, support, and unwavering belief in my capabilities, and who talked me through many difficult encounters with "the thesis blues".

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1
INTRODUCTION

This research addresses the "Cost-Constrained Traveling Salesman Problem," named for its similarity to the well-known Traveling Salesman Problem. In its general form, the problem is: given a set of tasks which require varying amounts of a limited resource, select a subset of the tasks and a sequence of performing this subset such that maximum value is obtained without exceeding the resource limit. The resource may be of any nature, but typically is time or money. Similarly, the tasks may be of any nature. The key characteristics are: the resource is limited; each task has a fixed value; the tasks may be viewed as sequential; completion of a task requires some amount of the limited resource; and the resource requirement for a task may depend on the previous task.

Although the Cost-Constrained Traveling Salesman Problem is very similar in nature to the Traveling Salesman Problem, there is a fundamental difference. The Cost-Constrained Traveling Salesman Problem requires both selection and sequencing of tasks, while the Traveling Salesman Problem requires sequencing only. In the Traveling Salesman Problem, the goal is not to select and sequence tasks to make optimal use of a limited resource. Rather, it is to sequence a fixed set of tasks in order to minimize use of an unlimited resource. The traveling salesman wishes to tour a fixed set of cities and the cost of this tour depends on the order in which the cities are visited. The goal is to find a *tour* — an ordering of the cities — that minimizes the total cost. In the Cost-Constrained Traveling Salesman Problem, the traveling salesman is given a fixed cost-constraint, or budget. The goal is to find a maximal subsequence of cities — a *subtour* — to visit without exceeding the cost-constraint. The problem may be further complicated if some cities have different values than others. In another version of the Traveling Salesman Problem, the problem is worded as "Can the set of tasks be completed given the resource constraint?" The answer is simply "yes" or "no." If the answer is "no," no attention is

given to optimizing the number or value of tasks which can be completed within the given constraint. Henceforth, we will refer to the tasks as *nodes*, the resource requirements as *costs*, and the resource limit as the *budget*.

While an abundance of literature is available on the Traveling Salesman Problem, very little work has been done on the cost-constrained version. We make use of previous work on both the Traveling Salesman Problem and the Cost-Constrained Traveling Salesman Problem as much as possible. However, the departure of the cost-constrained version from the Traveling Salesman Problem is significant enough to limit severely the applicability of Traveling Salesman Problem results. This is particularly true for theoretical results pertaining to approximation algorithms.

A point of interest is that the Cost-Constrained Traveling Salesman Problem was, in fact, the original version of the Traveling Salesman Problem. The earliest known reference to the Traveling Salesman Problem is a book published in Germany in 1831 by B. F. Voigt, *The Traveling Salesman, how he should be and what he should do to get Commissions and to be Successful in his Business. By a veteran Traveling Salesman* **[Vo]**. In this book, the author does not state that the objective is to minimize the cost of visiting all of the cities, but rather, "The most important aspect is to cover as many locations as possible . . ." **[HW]**. In subsequent work on the Traveling Salesman Problem, which doesn't really appear until the mid-1900's, the problem is changed to the current Traveling Salesman Problem formulation.

The first four chapters of this dissertation are introductory in nature. Chapter 2 provides a brief review of basic concepts in combinatorial optimization. Its main intention is to introduce terminology that is used in later chapters. This is followed by a formal description of the Cost-Constrained Traveling Salesman Problem, including alternate formulations, extensions, and applications. The theoretical complexity of the problem is also discussed. Chapter 4 contains a synopsis of relevant previous work.

The remaining chapters focus on algorithms for the Cost-Constrained Traveling Salesman Problem. Chapter 5 presents several special cases that can be solved with very efficient polynomial algorithms. This is followed by a discussion of the evaluation

framework that was used for computational evaluation of the algorithms presented in subsequent chapters. Chapters 7 through 9 address upper bounding methods, exact algorithms, and heuristic algorithms, respectively. Previous methods are discussed as well as new. Extensive computational experiments are used for evaluation and comparison of methods.

Finally, we conclude with a summary of results, as well as a discussion of open questions and promising areas for future research. In addition to the reference list, a bibliography is included. The reference list contains only those works which are specifically discussed in this dissertation. The bibliography contains additional citations of relevant work, including the textbooks that were the basis for Chapter 2.

Chapter 2
REVIEW OF COMBINATORIAL OPTIMIZATION

In combinatorial analysis, one is often concerned with the existence of a particular type of arrangement of a finite number of objects. Combinatorial optimization looks for the *best* arrangement of these objects. This is analogous to the distinction in combinatorics between *recognition* problems and *optimization* problems. In recognition problems, the existence of a particular type of arrangement is questioned. In optimization problems, the optimal arrangement is sought. For example, the recognition version of the Traveling Salesman Problem is:

> **TSP(recognition)**: Given a set of nodes $\{1,2,\ldots,n\}$ and a non-negative cost matrix $C = [c_{ij}]$, does there exist a tour, starting and ending at node 1, with total cost B or less?
>
> In other words, is there a permutation $\pi_n = \langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ with $\pi(1) = 1$ and
>
> $$\sum_{i=1}^{n-1} c_{\pi(i),\pi(i+1)} + c_{\pi(n),\pi(1)} \leq B \ ?$$

The optimization version is:

> **TSP(optimization)**: Given a set of nodes $\{1,2,\ldots,n\}$ and a non-negative cost matrix $C = [c_{ij}]$ find a tour, starting and ending at node 1, with minimum cost.
>
> In other words, find a permutation $\pi_n = \langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ with $\pi(1) = 1$ that minimizes
>
> $$\sum_{i=1}^{n-1} c_{\pi(i),\pi(i+1)} + c_{\pi(n),\pi(1)}.$$

Henceforth, the notation "TSP" refers to TSP(optimization).

As in the above definitions of TSP, we will denote an arrangement of a set of n objects by a permutation $\pi_n = \langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ — a one-to-one mapping of the set

$\{1,2,\ldots,n\}$ onto itself — where $\pi(i) = j$ indicates that the object with label j is in the ith position. Using this same interpretation, we denote an arrangement of a subset of a set of n objects by a partial permutation $\pi_n^m = \langle \pi(1), \pi(2), \ldots, \pi(m) \rangle$ — a one-to-one mapping of the set $\{1,2,\ldots,m\}$, where $m \leq n$, into the set $\{1,2,\ldots,n\}$.

For any finite set of objects, there is only a finite number of possible arrangements. Thus, solving combinatorial problems requires consideration of only a finite number of possibilities. This number, however, is usually prohibitively large, making total enumeration impractical. For example, the number of ways to arrange 25 objects is 25!, or approximately 1.5×10^{25}. Examining each arrangement using a nanosecond computer would take approximately 5×10^{8} years.

A combinatorial problem is defined by a general description of its parameters and a statement of the properties the solution is required to satisfy. An *instance* of a problem is a particular set of values for the problem parameters. A recognition problem has two types of instances, *yes* instances and *no* instances. "Yes" instances are those for which a solution satisfying the specified conditions exists. "No" instances are those for which such a solution does not exist. Solving a recognition problem means determining whether it is a "yes" instance or a "no" instance.

We can represent most combinatorial problems by a digraph $G = (N, A)$ and a matrix $C = [c_{ij}]$. In this representation, N is the set of nodes and A is a set of arcs or ordered pairs of nodes. Often, $A = N \times N$, i.e., there is an arc from every node to every other node. In this case, we say G is *complete*. A sequence or permutation of nodes is called a *path* and is represented by a subset of A where arc (i, j) is in the subset if and only if node j follows node i in the sequence. The cost associated with having node j follow node i is called the *length* or *arc length* of arc (i, j) and is represented by c_{ij}. In most cases, a digraph G which is not complete is equivalent to a complete digraph in which $c_{ij} = \infty$ for the arcs (i, j) not in G. The total cost of a sequence or subsequence of nodes is called the *path length*. In some cases, there is also a vector V of weights or values on the nodes.

We will now look at some characterizations of algorithms, but first we must define the "size" of a problem. The *size* of a problem is the number of bits required to represent

the data of the problem. The size of a graph is characterized by the number of vertices, the number of arcs, and the logarithm of the maximum arc length, which is proportional to the number of bits required to encode the data in a computer.

One characterization of an algorithm is the maximum time required to solve a problem of given size n, usually evaluated in terms of elementary operations (addition, multiplication, comparison, etc.). This gives a measure of the "worst case" behavior of the algorithm. This maximum time is a function $f(n)$ of the size of the problem. Since the measure of time depends on the types of operations, the relative times needed for these operations, the type of computer, etc., one generally considers the growth rate, or asymptotic order, of the function $f(n)$. We say an algorithm is of order g(n), or $O(g(n))$, if $f(n)/g(n)$ tends to a constant as $n \to \infty$. Algorithms that are $O(n)$ are called *linear*; those that are $O(n^p)$ are called *polynomial of order p*; those that are $O(2^n)$ are called *exponential*; and those that are $O(n!)$ are called *factorial*. An algorithm for which the computation time depends polynomially on numerical data not encompassed by tthe size of the problem is called *pseudo-polynomial*. Algorithms that are exponential or factorial are, in many cases, computationally infeasible for large problems. A common criterion for an "efficient" algorithm is that it be polynomial. This criterion is based on the assumption that the worst case behavior of an algorithm is typical of problems encountered in practice, and, therefore, should not be taken as gospel. In practice, many non-polynomial algorithms are very efficient, and, in some cases, more efficient than polynomial algorithms.

Another characterization of an algorithm is the "average" time required to solve a problem of given size n. This is important since, in many cases, the average time required to solve a problem is much better than the worst case. The Simplex Method is a classic example. Its worst case growth rate for solving linear programs is exponential, while its average growth rate, based both on problems encountered in practice and on randomly generated problems, appears to be little more than linear.

The storage space required to execute an algorithm may also be of interest. In some cases, the size of problems that can be solved is limited more by storage requirements than by computation time. We note that, as new generations of computers and storage devices

are developed, the restrictions imposed by storage requirements and computation time become less and less stringent. However, doubling the computation speed and storage capacity would not mean that we could solve problems twice as large, unless the algorithm being used was linear.

Recognition problems generally fall into two classes. The first, called P, is the class of recognition problems for which polynomial algorithms exist. A classic example of a problem in this class is the Assignment Problem (AP):

> **AP(recognition)**: Given a set of nodes $V = \{v_1, v_2, \ldots, v_n\}$, a set of nodes $U = \{u_1, u_2, \ldots, u_n\}$, and a cost matrix $C = [c_{ij}]$ where c_{ij} is the cost of assigning node v_i to node u_j, is there an assignment of V to U with cost B or less?
>
> In other words, is there a permutation $\pi_n = \langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ for which
>
> $$\sum_{i=1}^{n} c_{i,\pi(i)} \leq B,$$
>
> where $\pi(i) = j$ indicates that node v_i is assigned to node u_j?

Before we define the second class, we require some additional definitions.

The class called *NP* is a larger class of recognition problems that includes P. For a problem to be in NP, we do not require that every instance can be solved in polynomial time by some algorithm. We require only that for every "yes" instance of the problem, there exists a *certificate* — a proof that it is a "yes" instance — that can be checked for validity in polynomial time. This certificate is usually a solution that satisfies the specified criteria of the problem. For example, a certificate for a "yes" instance of TSP(recognition) is a tour of length B or less. (Note that the existence of a certificate implies nothing about the existence of an algorithm for finding the certificate.)

We say a problem A_1 *reduces in polynomial time* to a problem A_2 if, assuming there exists a polynomial algorithm for A_2, there exists a polynomial algorithm for A_1 that uses as a subroutine the algorithm for A_2. We say a recognition problem A_1 *polynomially transforms* to a recognition problem A_2, if, given any instance x of A_1, we can construct

within polynomial time (in the size of x) an instance y of A_2 such that x is a "yes" instance of A_1 if and only if y is a "yes" instance of A_2.

We can now define the second class of problems referred to above, those that are "NP-complete". A recognition problem A is *NP-complete* if A is in NP and all problems in NP polynomially transform to A. TSP(recognition) is a classic example of an NP-complete problem. A problem A is *NP-hard* if all problems in NP reduce in polynomial time to A, but A is not necessarily in NP. TSP(optimization) is an NP-hard problem. If there exists a polynomial algorithm for any NP-hard problem, then, by the definition of polynomial reducibility, there exists a polynomial algorithm for all problems in NP. This would mean that P = NP. The classes P, NP, NP-complete, and NP-hard are referred to as complexity classes. The question of whether P = NP is a long-standing open question in the field of combinatorics.

There are two types of algorithms for optimization problems, *exact* and *heuristic*. An *exact algorithm* is one that is proven to find the true solution to an optimization problem, that is, a solution whose value is the true optimum. *Heuristic algorithms* find solutions that satisfy the constraints of the problem but cannot be guaranteed to be optimal. Heuristic algorithms are designed to find solutions whose values are, at least, near to the optimal value. Since there are no known polynomial exact algorithms for NP-hard problems, heuristics are, in many cases, of great importance. For some problems, there exist heuristic algorithms for which it is possible to prove that a *performance guarantee* exists. These performance guarantees have several forms. For example, a performance guarantee might state that a given heuristic algorithm for a minimization problem always finds a solution whose value is not more than twice the optimal value. In general, however, performance guarantees cannot be obtained. In fact, for some problems (e.g. TSP), it is possible to prove that a heuristic with a performance guarantee cannot exist, unless P = NP. Furthermore, in cases where a performance guarantee does exist, heuristics generally perform better than their guarantee. For these reasons, heuristics are usually evaluated on the basis of their empirical performance.

Chapter 3
PROBLEM DESCRIPTION

In this chapter, we give a more precise definition of the problem under consideration and discuss several variations. We also look at the complexity of solving the problem, both exactly and approximately. Finally, we present some extensions and applications.

3.1 Basic Problem Formulation

Like the Traveling Salesman Problem, the Cost-Constrained Traveling Salesman Problem can be formulated as either an optimization problem or as a recognition problem. We define the optimization problem as follows:

> **CCTSP(optimization):** Given a set of nodes $\{1,2,\ldots,n\}$, a non-negative cost matrix $C = [c_{ij}]$, positive values $\{v_1, v_2, \ldots, v_n\}$, and a budget B, find a subtour of maximum value, starting and ending at node 1, whose total cost does not exceed B.
>
> In other words, find a partial permutation $\pi_n^m = \langle \pi(1), \pi(2), \ldots, \pi(m) \rangle$ with $m \leq n$, $\pi(1) = 1$, and
>
> $$\sum_{i=1}^{m-1} c_{\pi(i),\pi(i+1)} + c_{\pi(m),\pi(1)} \leq B$$
>
> that maximizes
>
> $$\sum_{i=1}^{m} v_{\pi(i)}.$$

The corresponding recognition problem is:

> **CCTSP(recognition):** Given a set of nodes $\{1,2,\ldots,n\}$, a non-negative cost matrix $C = [c_{ij}]$, positive values $\{v_1, v_2, \ldots, v_n\}$, and a budget B, does there exist a subtour, starting and ending at node 1, whose total cost does not exceed B and whose value is Q or greater?

In other words, for some $m \le n$, is there a partial permutation $\pi_n^m = \langle \pi(1), \pi(2), \ldots, \pi(m) \rangle$ with $\pi(1) = 1$,

$$\sum_{i=1}^{m-1} c_{\pi(i),\pi(i+1)} + c_{\pi(m),\pi(1)} \le B,$$

and

$$\sum_{i=1}^{m} v_{\pi(i)} \ge Q?$$

In this basic formulation of the Cost-Constrained Traveling Salesman Problem, we require the subtour to be a closed loop starting and ending at a specified node. Each node may be visited at most once and the nodes may have different values. Henceforth, the notation "CCTSP" refers to CCTSP(optimization), as defined above.

CCTSP can also be formulated as a 0-1 integer programming problem. In the integer programming formulation, we have $x_{ij} = 1$ if node j follows node i in the subtour and $x_{ij} = 0$ otherwise. The value of a node is accrued only if the node is contained in the subtour. The integer programming formulation can be written as

$$\max \sum_{i=1}^{n} \sum_{j=1}^{n} v_i x_{ij} \qquad (IP_1)$$

$$\text{subject to: } \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \le B$$

$$\sum_{j=2}^{n} x_{1j} = 1$$

$$\sum_{j=1}^{n} x_{ij} \le 1 \qquad \text{for } i = 1,2,\ldots,n$$

$$\sum_{i=1}^{n} x_{ij} - \sum_{k=1}^{n} x_{jk} = 0 \qquad \text{for } j = 1,2,\ldots,n$$

$$x_{ij} \in \{0,1\} \qquad \text{for all } i \text{ and all } j$$

$$\sum_{i \in S} \sum_{j \in S} x_{ij} \le |S| - 1 \qquad \text{for all } S \subset \{2,3,\ldots,n\},$$

where $|S|$ is the cardinality of S. We refer to the last 2^{n-1} inequalities as the *subtour elimination constraints* (also referred to in TSP literature as "loop" conditions). Note that if a solution contains a subtour that does not include node 1, then, letting S be the set of nodes in this subtour, the last constraint is violated. This prevents solutions containing two or more disjoint subtours and also prevents solutions with $x_{jj} = 1$ for some $j \neq 1$. Furthermore when node i is not included in the subtour, we have

$$\sum_{j=1}^{n} v_i x_{ij} = 0.$$

As an alternative integer programming formulation, we can define $c_{jj} = 0$ for all j and let $x_{jj} = 0$ if node j is included in the subtour and $x_{jj} = 1$ if node j is not included in the subtour. This leads to the formulation

$$\max \sum_{i=1}^{n} v_i (1 - x_{ii}) \qquad (IP_2)$$

$$\text{subject to: } \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij} x_{ij} \leq B$$

$$x_{11} = 0$$

$$\sum_{i=1}^{n} x_{ij} = 1 \qquad \text{for } j = 1,2,\ldots,n$$

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad \text{for } i = 1,2,\ldots,n$$

$$x_{ij} \in \{0,1\} \qquad \text{for all } i \text{ and all } j$$

$$\sum_{i \in S}\sum_{\substack{j \in S \\ j \neq i}} x_{ij} \leq |S| - 1 \qquad \text{for all } S \subset \{2,3,\ldots,n\}\ .$$

3.2 Alternate Formulations

Several alternatives to the basic formulation of CCTSP exist. These include cases where the starting node and/or ending node are not specified, a closed loop is not required, all nodes have equal value, and/or nodes may be visited more than once. We now discuss these alternate formulations and their relationship to the basic formulation.

Suppose that rather than a subtour starting and ending at node 1, we desire a path starting at node 1 and ending at an unspecified node or at a specified node other than node 1. We call this problem *CCTSP-path*. In the case where the ending node is unspecified, CCTSP-path can be reduced to CCTSP by setting the costs $c'_{i1} = 0$ for all i and requiring a subtour starting and ending at node 1. If we require the path to end at a specified node, m, then we reduce the problem to CCTSP by setting $c'_{m1} = 0$ and $c'_{i1} = \infty$ for all $i \neq m$. The conversions can also go the other way. CCTSP can be reduced to CCTSP-path by creating an artificial node $1'$ and setting $c_{i1'} = c_{i1}$ and $c_{1'i} = \infty$ for all i. Note that if node $1'$ occurs in the path, it will be the endpoint. We either specify node $1'$ as the ending node or, if the ending node is unspecified, assign a sufficiently large value to $v_{1'}$ assuring that node $1'$ will be the endpoint of the path. Similar transformations can be applied for the case where the ending node is specified but the starting node is not.

In the case where a closed subtour is required but the starting node is not specified, we reduce the problem to CCTSP by creating n instances, each one specifying a different starting node. The instance with the highest solution value gives the solution to the original problem. If the starting node is not specified and we desire a path rather than a closed subtour, we can transform the problem to CCTSP by adding an artificial node. We add node 0 with value $v_0 = 0$ and let $c_{i0} = c_{0i} = 0$ for $i = 1,\ldots,n$. We then solve CCTSP, requiring that the subtour start and end at node 0. To reduce CCTSP to the case where a closed subtour is required but the starting node is not specified, we assign a sufficiently high value to node 1. This assures node 1 will be in the optimal solution, thus giving a subtour that starts and ends at node 1.

If all nodes have equal value, then the objective reduces to maximizing the number of nodes in the subtour. No transformation is required. We can simply set $v_i = 1$ for all i. When the node values are integer and are not equal, CCTSP is transformed to a problem where the objective is to maximize the number of nodes in the subtour by creating v_i replicas of node i for each i, resulting in the set of nodes

$$\{1_1, 1_2, \ldots, 1_{v_1}, 2_1, 2_2, \ldots 2_{v_2}, \ldots n_1, n_2, \ldots, n_{v_n}\}$$

and the cost matrix C', where

$$c'_{i_k j_l} = c_{ij} \quad \text{for all } i,j,k,l \text{ with } i \neq j$$

and

$$c'_{i_k i_l} = 0 \quad \text{for all } i,k,l.$$

It is important to note that this is a pseudo-polynomial transformation rather than a polynomial transformation, as it increases the number of nodes from n to

$$\sum_{i=1}^{n} v_i.$$

Suppose we have a problem that is like CCTSP, except that nodes may be visited more than once. We assume that a node's value is acquired if the node occurs at least once in the subtour and that no additional value is acquired for multiple occurrences of the same node. Thus, the only reason to revisit a node is if doing so results in a cheaper path between two nodes than the direct path. We note that if the triangle inequality is satisfied, that is, if

$$c_{ik} \leq c_{ij} + c_{jk} \quad \text{for all } i,j,k,$$

then there is never anything to be gained by multiple visits to a node, since an indirect path is never cheaper than a direct one. Thus, in this case, an optimal solution exists in which no node is revisited, and solving CCTSP will give an optimal solution. If we replace c_{ij} with c'_{ij} where c'_{ij} is the length of a shortest path from node i to node j, then the triangle inequality will always be satisfied. Thus we can replace C with C', assume that each node may be visited at most once, and solve CCTSP. If (i_p, i_q) is an arc in the optimal solution under C' and $(i_p, i_s, i_t, \ldots, i_q)$ is a shortest route from i_p to i_q under C, we replace the arc (i_p, i_q) in the optimal solution with $(i_p, i_s, i_t, \ldots, i_q)$.

We have not found a transformation from CCTSP to the variant where nodes may be revisited. However, we can make a transformation to a similar problem. Let M be a sufficiently large number (e.g., $M > B$) and replace c_{ij} with $c'_{ij} = c_{ij} + M$ for each i,j. We allow nodes to be revisited but require that the subtour have cost less than $B' = B + mM$,

where m is the number of distinct nodes in the subtour. Although revisits are allowed the cost constraint assures that, in any feasible solution, no nodes will be revisited. Hence, any feasible solution is also feasible for CCTSP. This is not a precise transformation because B' is a function of m, an unspecified variable. In the case where we wish to maximize the number of nodes rather than the value, we can make a precise polynomial transformation between CCTSP(recognition) and the corresponding recognition problem that does allow nodes to be revisited. Suppose we desire to know whether there is a subtour containing m nodes and having cost less than B. We replace c_{ij} with $c'_{ij} = c_{ij} + M$ and ask whether there is a subtour containing m distinct nodes and having cost less than $B' = B + mM$. Although we allow nodes to be visited more than once, the cost constraint assures that any feasible subtour contains at most m arcs and, thus, if it contains m distinct nodes, no nodes are revisited.

3.3 Complexity

In this section, we show first that CCTSP(recognition) is NP-complete. We then show that, for certain types of approximations, a polynomial algorithm cannot exist unless P = NP.

Theorem 3.1: CCTSP(recognition) is NP-complete.

Proof: In order to show that CCTSP(recognition) is NP-complete, we must show (a) that the problem is in NP and (b) that all other problems in NP polynomially transform to CCTSP(recognition). To show (b), it suffices to show that a problem known to be NP-complete polynomially transforms to CCTSP(recognition). We will use TSP for this purpose.

CCTSP(recognition) is in NP if there exists a certificate that can be checked for validity in polynomial time for every "yes" instance of the problem. By definition, for every "yes" instance of CCTSP(recognition), there exists a partial permutation $\pi_n^m = \langle \pi(1), \pi(2), \ldots, \pi(m) \rangle$ with $m \le n$, $\pi(1) = 1$,

$$\sum_{i=1}^{m-1} c_{\pi(i),\pi(i+1)} + c_{\pi(m),\pi(1)} \le B,$$

and

$$\sum_{i=1}^{m} v_{\pi(i)} \geq Q.$$

Thus, π_n^m is a certificate and it can be validated in polynomial time by verifying that the above two constraints are satisfied. Hence, CCTSP(recognition) belongs to the class NP.

To conclude our proof, we show that TSP(recognition) polynomially transforms to CCTSP(recognition). Let $v_i = 1$ for all i, and $Q = n$. We then ask:

Is there a partial permutation $\pi_n^m = \langle \pi(1), \pi(2), \ldots, \pi(m) \rangle$ with $m \leq n$, $\pi(1) = 1$,

$$\sum_{i=1}^{m-1} c_{\pi(i),\pi(i+1)} + c_{\pi(m),\pi(1)} \leq B,$$

and

$$\sum_{i=1}^{m} v_{\pi(i)} \geq n?$$

A permutation π_n^m that satisfies these constraints must contain $m = n$ nodes and thus defines a Traveling Salesman tour of the n nodes with total cost B or less. Likewise, any Traveling Salesman tour with cost B or less defines a permutation π_n that satisfies the above conditions. Thus, there exists a polynomial transformation from TSP(recognition) to CCTSP(recognition), completing our proof that CCTSP(recognition) is NP-complete. □

Note that this proof also shows that the special case of CCTSP(recognition) where all nodes have equal value is NP-complete.

Corollary 3.1: CCTSP(optimization) is NP-hard.

Consider now the problem of finding an approximate solution to CCTSP with a value within K of the optimal solution. We will call an algorithm that finds such a solution a K-approximation algorithm.

Theorem 3.2: No K-approximation algorithm exists for CCTSP, unless P = NP.

Proof: Suppose algorithm A is a K-approximation algorithm for CCTSP and we wish to solve an instance of TSP(recognition). Let $v_i = K+1$ for $i = 1,\ldots,n$. We then apply algorithm A to the problem:

(CCTSP) Find a partial permutation $\pi_n^m = \langle \pi(1), \pi(2), \ldots, \pi(m) \rangle$ with $m \le n$, $\pi(1) = 1$, and

$$\sum_{i=1}^{m-1} c_{\pi(i),\pi(i+1)} + c_{\pi(m),\pi(1)} \le B$$

that maximizes

$$\sum_{i=1}^{m} v_{\pi(i)}.$$

If there exists a Traveling Salesman tour with cost B or less, then there exists a feasible solution to CCTSP containing n nodes and, thus, having value $n(K+1)$. Note that this is the maximum value possible. Hence, if the instance of TSP is a "yes" instance, the optimal value for CCTSP is $n(K+1)$. By definition, algorithm A will find an approximate solution to CCTSP with value $n(K+1)-K = (n-1)(K+1)+1$ or greater. Thus, the solution must contain all n nodes and have value $n(K+1)$. If the instance of TSP is a "no" instance, there does not exist a Traveling Salesman tour with cost B or less. Any feasible solution to CCTSP must contain at most $n-1$ nodes and have value $(n-1)(K+1)$ or less. We can answer TSP(recognition) by applying algorithm A to CCTSP. The answer to TSP is "yes" if and only if the value of the solution found by algorithm A is $n(K+1)$.. If algorithm A is polynomial, then we have found a polynomial algorithm which solves TSP(recognition), an NP-complete problem. Thus, a polynomial K-approximation algorithm cannot exist unless P = NP. □

A *fully polynomial approximation scheme* for a problem Π is an algorithm A such that for any $\varepsilon > 0$ (the accuracy requirement), A_ε is polynomial in the size of Π and $1/\varepsilon$ and

$$\frac{|\mathrm{OPT}(\Pi) - A_\varepsilon(\Pi)|}{\mathrm{OPT}(\Pi)} < \varepsilon.$$

Theorem 3.3: There is no fully polynomial approximation scheme for CCTSP unless P = NP.

Proof: Suppose A is a fully polynomial approximation scheme for CCTSP. Consider an instance I of CCTSP where all nodes have a value of 1 and there are n nodes. Let $\varepsilon = 1/n$. Then A_ε is polynomial in the size of I and n. If A_ε does not return the optimal value for an instance I of CCTSP then $\mathrm{OPT}(I) - A(I) \geq 1$. Also, $\mathrm{OPT}(I) \leq n$. So,

$$\frac{|\mathrm{OPT}(\Pi) - A_\varepsilon(\Pi)|}{\mathrm{OPT}(\Pi)} \geq 1/n = \varepsilon.$$

Thus, A_ε must return the optimal value and is a polynomial algorithm for the case of CCTSP where all nodes have equal value. Since CCTSP where all nodes have equal value is an NP-hard problem, the existence of A implies P = NP. □

Several complexity questions still remain open. Our primary interest in in the following: Is there *any* polynomial approximation algorithm for CCTSP with a performance guarantee? That is, is there any algorithm A and number r such that

$$\frac{|\mathrm{OPT}(I) - A(I)|}{\mathrm{OPT}(I)} \leq r$$

for all instances I of CCTSP?

3.4 Extensions

Two extensions of CCTSP are of particular interest. These are (1) the case where there are time windows on the nodes and (2) the time varying problem. Both extensions have practical applications, for example, in the area of battle management for strategic defense systems.

In the first extension, the case of time windows, the limited resource is time. Rather than a budget, a time window, defined by a start and stop time, is placed on each node and a starting time for the subtour is specified. A node may be visited only during its time window. In some cases, it may be desirable to wait some period of time at a node before continuing the subtour and this is allowed. Baker **[Ba]** presents a branch and bound algorithm for a limited version of this problem. Rather than seeking a feasible

subtour of maximum value, he seeks a complete tour satisfying the time window constraints. If such a tour does not exist, the problem is infeasible. CCTSP is the special case of this extension where all the nodes have identical time windows.

Another extension of CCTSP is the time varying problem. Again, the limited resource is time and a start time for the subtour is specified. The node values and the cost matrix may change over time and are defined as functions of time, $v(t)$ and $C(t)$. There is no explicit budget constraint, but an implicit budget constraint may be defined by the time at which the node values go to zero. CCTSP is the special case where the cost matrix is constant and the value function is:

$$v_j = \begin{cases} v_j & \text{for } t \le B \\ 0 & \text{for } t > B. \end{cases}$$

3.5 Applications

The Cost-Constrained Traveling Salesman Problem is applicable to a wide variety of problems. Many problems traditionally treated with TSP are better handled with CCTSP. This discussion focuses on several applications that have arisen at Lawrence Livermore National Laboratory (LLNL), and one, more light-hearted, application.

Many military applications of CCTSP have arisen at LLNL. One example was a project involving battle management for a Free-Electron Laser Strategic Defense System. In this problem, the "traveling salesman" was a laser beam focused by a space mirror, the tasks were destroying missiles, the resource was time, and the budget was the length of the window of vulnerability. The time required to destroy a missile depended on the missile's type and its angular distance from the previous missile destroyed. The goal was to find a target sequence which resulted in as many missiles being destroyed as possible. This is a challenging problem since the loss due to a target sequence that is even slightly suboptimal could be significant. Furthermore, the time required to compute the target sequence was critical since the problem data would be arriving in real time and time spent computing the sequence would be time not spent destroying targets. Applications occurring in naval tactics include task sequencing for mine sweepers and surveillance ships, and target

prioritization problems arising in naval air defense. Other military applications are countless.

In addition to military applications, many operational applications occur at LLNL. For example, hazardous waste management requires many waste processing tasks with sequence dependent set-up times to be scheduled at a single facility. Any tasks which cannot be completed within a specified time interval must be contracted to an outside facility at substantial cost. Thus, it is desirable to select and sequence the tasks to be completed in-house such that the cost of contracting out the remaining tasks is minimized.

On the lighter side, another application is the "time-constrained shopping spree" — the event where an individual wins a shopping spree of a specified time-length at a particular store. If the layout of the store and both the location and value of items is known ahead of time, the problem of computing an optimal "shopping strategy" is equivalent to CCTSP.

Chapter 4
REVIEW OF PREVIOUS WORK

While a great deal of work has been done on the Traveling Salesman Problem, relatively little previous work exists for CCTSP. Furthermore, the majority of the results for TSP cannot be extended to CCTSP. This is particularly true for approximation algorithms since, for TSP, the *cost* of an optimal solution is being approximated, while for CCTSP, it is the *value* of an optimal solution that is being approximated. We begin this review of previous work by discussing some common algorithms for TSP that are alluded to in later sections. This is followed by a discussion of selected work on TSP and related problems. Finally, we give a brief discussion of previous work on CCTSP. More detailed discussions of previous algorithms for CCTSP are presented in subsequent chapters.

Four common heuristic procedures for TSP are referred to in subsequent chapters. These are: the nearest neighbor algorithm, the cheapest insertion algorithm, the farthest insertion algorithm, and the two-opt procedure. The first three of these are tour building procedures developed in the 1960's and are difficult to attribute to any particular individuals, while the fourth is a tour improvement procedure developed by Shen Lin in 1965 **[Lin]**.

The nearest neighbor algorithm is a completely myopic procedure. It begins with a path consisting of a single node, usually the "home base." At each step, the node not yet in the path that is closest to the node at the end of the path is added to the end of the path. When all nodes have been added to the path, the last node is connected to the starting node to form a tour. Rosenkrantz, Stearns, and Lewis **[RSL]** proved that, for instances of TSP that satisfy the triangle inequality, the nearest neighbor algorithm always produces tours with lengths not greater than

$$\left(\frac{1}{2}\lceil \log_2 n \rceil + \frac{1}{2}\right) \times \text{optimal tour length.}$$

Both the cheapest insertion and farthest insertion methods begin with a subtour consisting of a single node, usually the "home base," and iteratively insert nodes into the tour. At each step in the cheapest insertion algorithm, the cheapest insertion point and associated insertion cost is determined for each node not yet in the tour. The cheapest insertion point for a node is the place in the current subtour where inserting the node results in the minimum cost increase, where the cost of inserting a node p between nodes i and j is

$$c_{ip} + c_{pj} - c_{ij}.$$

The node with the smallest insertion cost is then selected and inserted at its cheapest insertion point. In the farthest insertion algorithm, for each node not yet in the subtour, its distance from the current subtour is determined. This distance is the minimum of the distances from each node in the current subtour. The node which is farthest from the current subtour is selected and inserted in the subtour at its cheapest insertion point. In both algorithms, the insertion process continues until a tour including all nodes has been generated. Rosenkrantz, Stearns, and Lewis **[RSL]** have shown that, for instances of TSP satisfying the triangle inequality, the cheapest insertion algorithm produces tours whose lengths, or costs, are not greater than twice the length of an optimal solution, while for the farthest insertion algorithm they are only able to show that it produces tours whose lengths are no greater than

$$\left(\lceil \log_2 n \rceil + 1\right) \times \text{optimal tour length.}$$

In spite of this, their computational experiments show that, in practice, the average performance of the farthest insertion algorithm is at least as good as that of the cheapest insertion and nearest neighbor algorithms.

A tour is said to be λ-optimal (or λ-opt) if it is impossible to obtain a tour with smaller cost by replacing any λ of its arcs with any other set of λ arcs. Making such a replacement when $\lambda = 2$ is equivalent to inverting, or reversing, the order of a set of neighboring nodes in the tour. Thus, a tour that is 2-opt is optimal relative to inversion. A two-opt routine is a procedure that takes a tour and makes it two-optimal by iteratively performing profitable inversions, until no further profitable inversions are possible.

One of the early exact algorithms for TSP is a branch-and-bound method that uses the assignment problem for bounding and subtour elimination constraints for branching [Li]. The assignment problem is a straightforward relaxation of TSP obtained by dropping the constraints that require the solution to be a single tour. An assignment is a union of directed cycles, hence, either a tour or a collection of subtours. By successively adding subtour elimination constraints, a single tour is eventually obtained. Garfinkel [Ga73] developed a branching rule, using subtour elimination, that produces more tightly constrained subproblems than previous subtour elimination schemes. At each node in the branch-and-bound tree, an assignment problem is solved. If the resulting assignment is not a single tour, a subtour is selected for elimination. Let $\{a_1, a_2, \ldots, a_m\}$ denote the sequence of arcs in this subtour. The problem is then partitioned into m subproblems where the ith subproblem includes additional constraints excluding arc a_i from the assignment and requiring the assignment to include arcs $a_1, \ldots, a_{i-1}$.

As we have already seen, several results pertaining to approximation algorithms have been obtained for TSP. For the special case of TSP where the cost matrix satisfies the triangle inequality, a number of algorithms with performance guarantees exist. The most notable of these is Christofides' algorithm [Ch] which first solves a minimum spanning tree problem and then turns the tree into a tour by solving a bipartite matching algorithm. Christofides' algorithm produces tours whose lengths are not greater than 1.5 times the optimal tour length. Fortunately, in most cases, these approximation algorithms perform much better than their guarantees. Another notable theoretical result pertaining to approximation algorithms is that, for the general TSP (where the triangle inequality is not necessarily satisfied), unless P = NP, there is no polynomial algorithm A with a performance guarantee of the form

$$length_A \leq r \times length_{opt},$$

where r is a finite constant [SG]. This is proven by showing that such an algorithm could be used to solve the Hamiltonian Cycle Problem, another NP-complete problem.

Crowder and Padberg [CP] have developed a method of solving TSP to optimality that has been applied successfully to very large TSP instances. The method, which they

call branch-and-cut, uses a cutting plane approach coupled with branch-and-bound. The cutting planes they use are problem specific and are based on previous theoretical results regarding facets of the traveling salesman polytope. Because of this, their algorithm cannot be applied to CCTSP. If similar theoretical results regarding the CCTSP polytope could be derived, a similar approach might be developed. Padberg and Rinaldi **[PR]** applied the branch-and-cut approach to a variation of the traveling salesman problem having several side constraints, including a cost-constraint. Their computation times, which range from 100 to 500 seconds for 11-city problems, raise doubts about how promising this approach might be for CCTSP. However, they were applying the approach to a problem significantly more complicated than CCTSP and, thus, the results may not be indicative of what would occur for CCTSP.

A problem closely related to CCTSP is the Prize Collecting Traveling Salesman Problem (PCTSP) **[FT]**, which might be described as the converse of CCTSP. In fact, CCTSP(recognition) and the recognition version of the PCTSP are identical. In the Prize Collecting Traveling Salesman Problem, the objective is to find a tour or subtour of minimum cost, subject to the requirement that at least a specified value be obtained. TSP is a special case of the Prize Collecting Traveling Salesman Problem. The same observation does not hold for CCTSP. Furthermore, some of the results for TSP, which cannot be extended to CCTSP, can be extended to the PCTSP. The proof that a polynomial algorithm with a performance guarantee does not exist unless P = NP is one example. If one had an exact algorithm for the PCTSP, it could be applied to CCTSP by "parametrically" solving a PCTSP formulation. This is done by repetitively solving the PCTSP, varying the total value requirement each time. CCTSP is solved when a value requirement V is found such that the cost of the optimal solution to the PCTSP does not exceed B, but the cost of an optimal solution to the PCTSP when the value requirement is $V+1$ does exceed B. Likewise, a heuristic algorithm for the PCTSP could be applied parametrically to obtain an approximate solution to CCTSP. However, if there existed a performance guarantee for the PCTSP algorithm, it would not imply a performance

guarantee when applied to CCTSP. The converse of these last three observations is also true.

Another related problem is the Minimal Cost-to-Time Ratio Cycle Problem. In this problem, each arc is assigned a profit (analogous to the node vlaues in CCTSP) and a travel time (analogous to the costs in CCTSP). To obtain a minimization problem, the profits are multipled by -1 and called costs. The objective is to find a subtour, or cycle, for which the ratio of total cost to total travel time is minimized. Dantzig, Blattner, and Rao **[DBR]** showed that the problem reduces to finding negative cycles within an iterative framework and can be solved in $O(n^3 \log \tau)$ time, where n is the number of nodes and τ is the maximum entry in the profit and time matrices. We note that when the travel times are non-negative and the travel time and cost matrices are symmetric, an optimal solution is the cycle formed by the arcs $(\hat{i},\hat{j})$ and $(\hat{j},\hat{i})$, where $(\hat{i},\hat{j}) = \text{argmin}\, c_{ij}/t_{ij}$. To prove this, consider any cycle C. The cost-to-time ratio of C is

$$\frac{\sum_{(i,j)\in C} c_{ij}}{\sum_{(i,j)\in C} t_{ij}} = \frac{\sum_{(i,j)\in C} \frac{c_{ij}}{t_{ij}} t_{ij}}{\sum_{(i,j)\in C} t_{ij}} \geq \frac{\sum_{(i,j)\in C} \frac{c_{\hat{i}\hat{j}}}{t_{\hat{i}\hat{j}}} t_{ij}}{\sum_{(i,j)\in C} t_{ij}} = \frac{\frac{c_{\hat{i}\hat{j}}}{t_{\hat{i}\hat{j}}} \sum_{(i,j)\in C} t_{ij}}{\sum_{(i,j)\in C} t_{ij}} = \frac{c_{\hat{i}\hat{j}}}{t_{\hat{i}\hat{j}}}.$$

Prior to this dissertation, relatively little research had been done on CCTSP. Golden, Levy, and Dahl **[GLD]** published a heuristic algorithm in 1981 for a generalization of TSP for which CCTSP is a special case. Their algorithm is based on the cheapest insertion algorithm but uses a linear combination of node value and insertion cost, rather than insertion cost alone, to select nodes for insertion. No computational experiments were conducted to determine the quality of the algorithm.

In 1984, Tsiligirides **[Tsi]** addressed the sport of orienteering and formulated the problem faced by orienteering competitors as what we called "CCTSP-path" in the previous chapter. He developed several variations of two heuristic algorithms and compared them using three test problems and a number of budgets for each. The favored method, called "Tsiligirides' Stochastic Algorithm" in later papers, is similar to the nearest neighbor algorithm. Rather than using distance alone, nodes are selected for addition to the path

based on the ratio of their value to their distance from the last node in the path. Also, there is randomization in the node selection process. Thus, the algorithm can be repeated a number of times and the best solution chosen.

Golden, Levy, and Dahl **[GLD]** presented another heuristic algorithm for CCTSP in 1987. Their algorithm was based on an idea which they called "center of gravity." The algorithm did not include any randomization. However, the algorithm generated a number of solutions bearing a deterministic relationship to each other. The algorithm compared favorably with Tsiligirides' stochastic algorithm for the three test problems used by Tsiligirides. Further computational experiments were not conducted.

Later in 1987, Golden, Wang, and Liu **[GLW]** developed a more complicated heuristic for CCTSP. Their algorithm was less myopic than the previous algorithms. When selecting a node for insertion, they took into consideration how the insertion of that node might affect the future progress of the algorithm. Their algorithm utilized randomization but also had a deterministic component in the repetition process. Again, computational experiments were done using the three test problems presented by Tsiligirides, and their algorithm compared favorably to the previous two in terms of solution quality. It required substantially more computation time than the center of gravity algorithm. No computational experiments were done to determine how close any of these heuristics came to optimality. This algorithm, as well as the other heuristics for CCTSP, are presented in greater detail in Chapter 9.

Two exact algorithms for CCTSP, utilizing different branch-and-bound schemes, were developed in 1988. One, by Laporte and Martello **[LM]**, uses a very simple branching rule and an upper bounding method based on the knapsack problem. The other, by Kataoka and Morito **[KM]**, uses an approach similar to the branch-and-bound method described above for TSP. Bounding is done using a variant of the assignment problem, and branching is based on subtour elimination. Since the two algorithms were developed at approximately the same time, no comparison of the two methods was made. These two methods are discussed in further detail in Chapter 8.

Chapter 5
SPECIAL CASES

In this section, we present some special cases of CCTSP that can be solved in polynomial time. Many special cases of TSP have been shown to be solvable with efficient polynomial algorithms. The special cases discussed here are a subset of those presented for TSP in a survey by Gilmore, Lawler, and Schmoys **[GLS]**. Each one is defined by placing restrictions on the cost matrix C and, in some cases, requiring the nodes to have equal value.

5.1 Outer-Sum Matrices

Our definition of an outer-sum matrix is inspired by the constant TSP. A constant TSP is one for which all possible tours have the same cost. Berenguer **[Be]** has shown that the only cost matrices C for which all traveling salesman tours have the same cost are those of the form

$$c_{ij} = a_i + b_j \quad \text{for all } i, j.$$

We will call matrices of this form *outer-sum matrices*. An interpretation of this form is that each node has associated with it a fixed cost for entering that node and a fixed cost for leaving it. The cost of traversing an arc is the sum of the cost of leaving its origin and the cost of entering its destination.

Theorem 5.1: For CCTSP where C is an outer-sum matrix, the cost of a subtour depends only on the subset of nodes included in the subtour and not on the order in which these nodes are visited. If S is the subset of nodes contained in a subtour, the cost of the subtour is

$$\sum_{i \in S} (a_i + b_i).$$

Proof: We can divide the cost of each arc (i, j) into the cost of starting at node i (a_i) and the cost of ending at node j (b_j). We know that there are exactly $|S|$ arcs in the subtour and that each node in S is the starting point of exactly one arc and the endpoint of exactly one arc. Thus, the cost of the subtour is

$$\sum_{i \in S} (a_i + b_i). \quad \square$$

Theorem 5.2: For CCTSP where C is an outer-sum matrix and all nodes have equal value, an optimal solution is to cycle through the first m nodes, where the nodes are labeled such that

$$a_2 + b_2 \leq a_3 + b_3 \leq \ldots \leq a_n + b_n.$$

and m satisfies

$$\sum_{i=1}^{m} (a_i + b_i) \leq B < \sum_{i=1}^{m+1} (a_i + b_i).$$

Proof: By Theorem 5.1 and the definition of m, the cost of a subtour containing nodes $1,2,\ldots,m$ is less than or equal to B. Thus, the proposed solution is feasible. Furthermore, any subtour with a greater value contains node 1 and at least m other nodes. By Theorem 5.1 and the definition of m, the cost of such a tour must be greater than B. Thus, there are no feasible tours with a greater value. $\square$

In the general case, since the cost of a subtour depends only on the subset of nodes included in the subtour and not on the order in which they are visited, the problem reduces to

$$\begin{aligned} & \max \sum_{i \in S} v_i \\ \text{subject to: } & \sum_{i \in S} (a_i + b_i) \leq B - (a_1 + b_1) \\ & S \subseteq \{2,3,\ldots,n\}. \end{aligned}$$

This is equivalent to the knapsack problem,

$$\text{(KP)} \qquad \max \sum_{i=1}^{n} v_i x_i$$

$$\text{subject to:} \quad \sum_{i=1}^{n} c_i x_i \le B$$

$$x_i = 0 \text{ or } 1 \quad \text{for all } i,$$

which is known to be NP-hard **[PS]**. When the costs c_i and the budget B are integer, KP can be solved in $O(nB)$ time using a dynamic programming algorithm **[Dan]**. However, since this algorithm is pseudopolynomial — it depends on the magitude of B — its computational efficiency is highly dependent on the scale of the problem.

5.2 Small Matrices

A matrix C is called *small* if there exist n-dimensional vectors a and b such that $c_{ij} = \min\{a_i, b_j\}$. These matrices have the property that, for each node, there is a cost associated with entering that node and a cost associated with leaving it. When traversing an arc, one chooses whether to incur the cost of leaving its origin or to incur the cost of entering its destination, rather than incurring both. We will assume that all of the elements of a and b are distinct and define d_i as the ith smallest of the $2n$ distinct values of a and b. Thus, $d_1 < d_2 < \ldots < d_{2n}$. Note that $\{d_1, d_2, \ldots, d_{2n}\} = \{a_1, a_2, \ldots, a_{2n}\} \cup \{b_1, b_2, \ldots, b_{2n}\}$. We will show that CCTSP where C is a small matrix and all nodes have equal value can be solved in $O(n^2)$ time.

Theorem 5.3: For CCTSP with a small matrix C, suppose $D \subseteq \{d_1, d_2, \ldots, d_{2n}\}$ is the set of arc lengths for the arcs that comprise a subtour containing node 1. Then either

(i) For some node i, both $a_i \in D$ and $b_i \in D$,

or (ii) $D \subseteq \{a_1, a_2, \ldots, a_n\}$ and $a_1 \in D$,

or (iii) $D \subseteq \{b_1, b_2, \ldots, b_n\}$ and $b_1 \in D$.

Proof: Suppose $D \subseteq \{a_1, a_2, \ldots, a_n\}$. If node 1 is in the subtour, then the subtour uses some arc $(1,k)$ with arc length $c_{1k} = \min\{a_1, b_k\}$. This means that $a_1 \in D$.

Suppose $D \subseteq \{b_1, b_2, \ldots, b_n\}$. If node 1 is in the subtour, then the subtour uses some arc $(k,1)$ with arc length $c_{k1} = \min\{a_k, b_1\}$. This means that $b_1 \in D$.

Suppose $D \not\subseteq \{a_1, a_2, \ldots, a_n\}$ and $D \not\subseteq \{b_1, b_2, \ldots, b_n\}$. Then, viewing the subtour as a continuous loop, at some point in the subtour, an arc with cost b_i must be followed by one with cost a_j for some i and j. But this means that arc (k,i) is followed by arc (j,l), which means $i = j$. Thus, both $a_i \in D$ and $b_i \in D$. □

Theorem 5.4: Let D^* be the D with maximum cardinality that satisfies the conditions of Theorem 5.3 and satisfies

$$C(D^*) = \sum_{i \in D^*} d_i \leq B.$$

Then there exists a feasible subtour containing $|D^*|$ nodes, and there do not exist any feasible subtours containing more than $|D^*|$ nodes.

Proof: First let us show that there exists a feasible subtour containing $|D^*|$ nodes.

Suppose D^* satisfies condition (i). Let D_0 be the set of nodes with neither a_i or b_i in D^*, D_a be the set of nodes with only a_i in D^*, D_b be the set of nodes with only b_i in D^*, and D_2 be the set of nodes with both a_i and b_i in D^*. Note that $|D_a| + |D_b| + 2|D_2| = |D^*| \leq n$ and $|D_0| + |D_a| + |D_b| + |D_2| = n$. Thus, $|D_0| \geq |D_2|$. Construct a subtour as follows: start at any node in D_2, visit the nodes in D_a in any order, go to a node in D_0 (choose node 1 if $1 \in D_0$), visit the nodes in D_b in any order, and complete the tour by alternating between nodes in D_2 and D_0 until the nodes in D_2 are exhausted, finally, returning to the starting node. This subtour contains node 1 and has a cost no greater than $C(D^*)$.

Suppose D^* satisfies condition (ii). Let D_a be the set of nodes with a_i in D^*. Visit the nodes in D_a in any order. This subtour contains node 1 and has a cost no greater than $C(D^*)$.

Finally, suppose D^* satisfies condition (iii). Let D_b be the set of nodes with b_i in D^*. Visit the nodes in D_b in any order. This subtour contains node 1 and has a cost no greater than $C(D^*)$.

Now let us show that there is no feasible subtour containing more than $|D^*|$ nodes. Let D be the set of arclengths for the arcs that comprise a subtour containing node 1 and $|D| > |D^*|$. By Theorem 5.3 and the definition of $|D^*|$, the cost of the subtour is greater than B. □

Thus, in order to solve CCTSP where C is a small matrix and all nodes have equal value, we need only find D^* as defined in Theorem 5.4. Let

$$k_i = \underset{k}{\operatorname{argmax}} \sum_{\substack{j=1 \\ d_j \neq a_i \\ d_j \neq b_i}}^{k} d_j \leq B - (a_i + b_i)$$

$$k_a = \underset{k}{\operatorname{argmax}} \sum_{d_j \in \{a_2, a_3, \ldots, a_n\}}^{k} d_j \leq B - a_1$$

$$k_b = \underset{k}{\operatorname{argmax}} \sum_{d_j \in \{b_2, b_3, \ldots, b_n\}}^{k} d_j \leq B - b_1$$

and

$$D_i = \{d_1, d_2, \ldots, d_{k_i}\} \cup \{a_i, b_i\} \quad \text{for all } i \in \{1, 2, \ldots, n\}$$

$$D_a = (\{d_1, d_2, \ldots, d_{k_a}\} \cap \{a_1, a_2, \ldots, a_n\}) \cup \{a_1\}$$

$$D_b = (\{d_1, d_2, \ldots, d_{k_b}\} \cap \{b_1, b_2, \ldots, b_n\}) \cup \{b_1\}.$$

Then, $D^* = \operatorname{argmax}\{|D_a|, |D_b|, |D_1|, |D_2|, \ldots, |D_n|\}$. These computations can be made in $O(n^2)$ time.

5.3 Circulant Matrices

In this section, we show that the problem CCTSP-path where all nodes have equal value can be solved using the nearest neighbor rule (add the nearest (cheapest) unvisited node to the end of the path) when C is a circulant matrix. A circulant matrix is a matrix of the form

$$C = \begin{bmatrix} c_0 & c_1 & c_2 & \cdots & c_{n-1} \\ c_{n-1} & c_0 & c_1 & \cdots & c_{n-2} \\ c_{n-2} & c_{n-1} & c_0 & \cdots & c_{n-3} \\ \vdots & \vdots & \vdots & & \vdots \\ c_1 & c_2 & c_3 & \cdots & c_0 \end{bmatrix}.$$

The cells (i,j) such that $(j-1) = k(\text{mod}\, n)$ all have the same value c_k. We call these cells the kth stripe of C. Garfinkel **[Ga77]** has shown that the assignment given by the kth stripe yields $\gcd(k,n)$ subtours each containing $n/\gcd(k,n)$ nodes.

Define $k(0)$, $k(1),\ldots,$ $k(n-1)$ such that $c_{k(0)} \le c_{k(1)} \le \ldots \le c_{k(n-1)}$ and let

$$g_0 = \gcd(k(0), n)$$
$$g_{i+1} = \gcd(k(i+1), g_i).$$

The arcs from stripes $k(0)$, $k(1),\ldots,$ $k(i)$ yield a subgraph with g_i connected components, each containing n/g_i nodes (see Gilmore, Lawler, and Schmoys **[GLS]**). Suppose we desire a connected component containing m nodes. If we use only the arcs from stripes $k(0)$, $k(1),\ldots, k(\text{i})$, we know that we can only obtain connected components with at most n/g_i nodes in them. Thus, at best, we can produce $\left\lceil \frac{m}{n} g_i \right\rceil$ disjoint components of which $\left\lfloor \frac{m}{n} g_i \right\rfloor$ contain n/g_i nodes and one contains $m - \left\lfloor \frac{m}{n} g_i \right\rfloor n/g_i$ nodes.

Theorem 5.5: If C is a circulant matrix, a lower bound on the cost of connecting m nodes is

$$\left(m - \left\lceil \frac{m}{n} g_0 \right\rceil\right) c_{k(0))} + \left(\left\lceil \frac{m}{n} g_0 \right\rceil - \left\lceil \frac{m}{n} g_1 \right\rceil\right) c_{k(1))} + \ldots + \left(\left\lceil \frac{m}{n} g_{n-2} \right\rceil - \left\lceil \frac{m}{n} g_{n-1} \right\rceil\right) c_{k(n-1)}.$$

Proof: Connecting m nodes requires a minimum of $m-1$ arcs. To obtain a lower bound, we assume exactly $m-1$ arcs are used. As previously stated, using only arcs from stripes $k(0)$, $k(1),\ldots,$ $k(i)$ results, at best, in $\left\lceil \frac{m}{n} g_i \right\rceil$ disjoint components. Thus, at least $\left\lceil \frac{m}{n} g_i \right\rceil - 1$ arcs must come from stripes $k(i+1),\ldots,$ $k(n-1)$, leaving $m - \left\lceil \frac{m}{n} g_i \right\rceil$ arcs that can come from stripes $k(0)$, $k(1),\ldots,$ $k(i)$. Applying these bounds iteratively results in the lower bound given above. □

Corollary 5.1: If C is a circulant matrix and all nodes are of equal value, an upper bound on the optimal value of CCTSP-path is $\hat{m}$ where $\hat{m}$ solves

$$\max_{1 \le m \le n} \quad m$$

subject to

$$\left(m - \left\lceil \frac{m}{n} g_0 \right\rceil\right) c_{k(0)} + \left(\left\lceil \frac{m}{n} g_0 \right\rceil - \left\lceil \frac{m}{n} g_1 \right\rceil\right) c_{k(1)} + \ldots + \left(\left\lceil \frac{m}{n} g_{n-2} \right\rceil - \left\lceil \frac{m}{n} g_{n-1} \right\rceil\right) c_{k(n-1)} \le B.$$

Theorem 5.6: If C is a circulant matrix and all nodes are of equal value, the nearest neighbor rule yields an optimal solution to CCTSP-path.

Proof: Starting at node 1 and applying the nearest neighbor rule results in a sequence as follows, where a always refers to the last node in the current path:

0) Repeatedly add node $a + k(0) \pmod n$ to the path until the next addition will result in a cycle or will exceed the budget.

1) Add node $a + k(i_1) \pmod n$ to the path where $i_1 = \min\left(i > 0 \mid g_{i_1} \ne g_0\right)$. Repeat from Step 0 until the next addition will result in a cycle or will exceed the budget.

⋮

j) Add node $a + k\left(i_j\right) \pmod n$ to the path where $i_j = \min\left(i > 0 \mid g_{i_j} \ne g_{i_{j-1}}\right)$. Repeat from Step 0 until the next addition will result in a cycle or will exceed the budget.

⋮

The end result of this sequence is a path containing $\hat{m}$ nodes and having cost

$$\left(\hat{m} - \left\lceil \frac{\hat{m}}{n} g_0 \right\rceil\right) c_{k(0)} + \left(\left\lceil \frac{\hat{m}}{n} g_0 \right\rceil - \left\lceil \frac{\hat{m}}{n} g_1 \right\rceil\right) c_{k(1)} + \ldots + \left(\left\lceil \frac{\hat{m}}{n} g_{n-2} \right\rceil - \left\lceil \frac{\hat{m}}{n} g_{n-1} \right\rceil\right) c_{k(n-1)}$$

where $\hat{m}$ is as defined in Corollary 1. □

5.4 Upper Triangular Matrices

A matrix C is *upper triangular* if $i \geq j$ implies $c_{ij} = 0$. We will show that solving CCTSP, where C is an upper triangular matrix, is as easy as computing shortest paths.

Theorem 5.7: For CCTSP where C is an upper triangular matrix, the cost, or length, of a subtour containing node m is at least as great as the length of the shortest path from node 1 to node m.

Proof: Since a subtour containing node m must contain a path from node 1 to node m, and the cost matrix C is non-negative, the cost of a subtour containing node m must be at least as great as the length of the shortest path from node 1 to node m. □

Theorem 5.8 For CCTSP where C is an upper triangular matrix, if there exists a path π from node 1 to node m with cost less than or equal to B, then there exists a feasible subtour π' containing nodes 1 through m.

Proof: Let π' start at node 1 and follow the same path as π until node m is reached. From node m, visit the remaining nodes in the set $\{1,2,3,...,m\}$ in order of decreasing index, returning to node 1 at the end. The portion of π' from node 1 to node m has a cost no greater than that of π. The remaining portion of π' has a cost of 0. Thus, π' is a feasible subtour. □

Theorem 5.9: An optimal solution to CCTSP, where C is an upper triangular matrix, is to follow the shortest path from node 1 to node m, and then visit the remaining nodes in the set $\{1,2,...,m\}$ in order of decreasing index, returning to node 1 at the end, where m is the maximum index for which the shortest path from node 1 to node m has length less than or equal to B.

Proof: By Theorem 5.8 and the definition of m, the proposed solution is feasible. Any solution having a greater value must contain a node with index greater than $m+1$, but by Theorem 5.7 and the definition of m, such a solution cannot be feasible. □

Theorem 5.10: If C is an upper triangular matrix, then the length of the shortest path from node 1 to node j is less than or equal to the length of the shortest path from node 1 to node $j+1$.

Proof: Because of the special structure of C, the path obtained by taking the shortest path from node 1 to node j and then visiting node $j+1$ has a length equal to the shortest path from node 1 to node $j+1$. □

Corollary 5.2: Solving CCTSP, where C is an upper triangular matrix, requires computing at most $O(\log n)$ shortest paths.

Chapter 6
EVALUATION FRAMEWORK

In this chapter, we describe the evaluation framework used in computational evaluation of the algorithms presented in the following three chapters. The performance measures used are defined first, and include both speed and solution quality. This is followed by a description of the types of test problems used in the evaluations. The test problems used encompass many different problem characteristics. This was done as an effort to uncover sensitivities of an algorithm's performance to problem characteristics.

6.1 Performance Measures

Algorithms are evaluated based on two performance measures: solution quality and computation speed. Exact algorithms are evaluated on computation time alone. Statistical tests are used in evaluating the difference between two algorithms.

The quality of a solution generated by an upper bounding method or a heuristic algorithm is measured in terms of its closeness to optimality. For upper bounding methods, the measure used is

$$\% \text{ error} = 100 \times \frac{UB - V_{\text{opt}}}{V_{\text{opt}}}$$

where UB is the upper bound obtained by the algorithm and V_{opt} is the value of an optimal solution obtained by an exact algorithm. For heuristic algorithms, the solution quality is measured by

$$\% \text{ error} = 100 \times \frac{V_{\text{opt}} - V_{\text{heur}}}{V_{\text{opt}}}$$

where V_{heur} is the value of the solution obtained by the heuristic. For problems which are too large to obtain an optimal solution with one of our exact algorithms (the test problems with 50 or more nodes), V_{opt} is replaced with the value of the best known solution,

obtained by applying several heuristics to the problem. The solution quality of an algorithm is generally reported as an average over a number of test problems sampled from the same population.

In comparing two algorithms, the Wilcoxon Signed Rank Test **[BJ]**, a nonparametric statistical test, is used to check whether there is a significant difference in the quality of solutions they produce. The null hypothesis is that there is no difference in the performance of the two algorithms. The alternate hypothesis is that one algorithm produces solutions with a smaller error than the other. The null hypothesis is rejected if the probability, under the null hypothesis, of observing differences at least as large as the differences obtained in the computational experiments is less than 5%.

The computation speed of an algorithm is the amount of CPU time (reported in seconds) required to execute the algorithm. Input and output are not included in the CPU time. As with solution quality, CPU times are generally reported as an average over a number of test problems sampled from the same population. Computational experiments were conducted on a SUN 4/330 workstation, which has a 25 MHz SPARC processor and is rated at 16 MIPS and 2.5 MFLOPS. The resolution of the CPU clock on this machine is 16.67 milliseconds.

6.2 Test Problems

In our computational experiments, we desired test problems which would stress an algorithm, as well as ones that might represent an average case. The key factors which might affect an algorithm's performance are the structure of the cost matrix, the relative values on the nodes, and the percentage of nodes in an optimum solution. Computational experiments were conducted using test problems representing 18 different combinations of problems characteristics. These characteristics, which we discuss below, are: *class*, *distribution*, *node values*, and *budget*. In our experiments, we specifically looked for sensitivities to problem characteristics, both in the performance of a specific algorithm and in the comparison between two or more algorithms.

Our test problems can be divided into two classes: Euclidean and non-Euclidean. In Euclidean problems, the nodes correspond to points in a plane and can be represented by

x-y coordinates. The cost matrix is the matrix of Euclidean distances between the nodes and, therefore, is symmetric and satisfies the triangle inequality. *Non-Euclidean* problems do not necessarily have a geometric representation. The cost matrix may be any $n \times n$ non-negative matrix. However, in all the non-Euclidean problems we generated, the cost matrices were symmetric. Furthermore, we applied a shortest-path algorithm to the cost matrices (treating costs as distances) and replaced them with the matrices of shortest-path costs. This results in matrices that satisfy the triangle inequality. Doing this is analogous to allowing indirect paths to be taken (which may result in multiple visits to a node) when it is advantageous to do so.

For both Euclidean and non-Euclidean problems, test problems are generated using three different *distributions*: uniform, clusters, and outliers. For Euclidean problems, the distribution refers to the distribution of nodes in the x-y plane. Cost matrices for *uniform* problems are generated by distributing nodes uniformly in a circle of radius 100, and calculating the resulting distance matrix. (We chose to use a circle rather than a square or rectangle because it seemed better suited to the following two types of problems.) *Clusters* refers to a problem where uniform clusters of nodes are uniformly distributed. The problems are generated by first generating cluster points and corresponding cluster sizes. The cluster points are uniformly distributed in a circle of radius 100. The cluster sizes, that is, the number of nodes in a cluster, are uniformly distributed between 1 and $0.4n-1$ for problems with less than 50 nodes and between 1 and $0.2n-1$ for problems with 50 or more nodes. The cost matrix is then generated by distributing the appropriate number of nodes uniformly in a circle of radius 20 centered at each cluster point, and calculating the resulting distance matrix. In *outlier* problems, 80 percent of the nodes are uniformly distributed within a circle of radius 100. The remaining nodes are uniformly distributed in the ring formed by this circle and a concentric circle of radius 200. Again, the cost matrix is the resulting distance matrix. For all problems, entries in the distance matrices are rounded up to integer values.

In non-Euclidean problems, the entries in the cost matrix are directly generated. This is done in such a way as to be analogous to the Euclidean distributions. For uniform

problems, symmetric entries in the cost matrix are integers uniformly distributed between 1 and 200. The matrix is then replaced by the matrix of shortest paths. For clusters, first cluster sizes are generated as defined above. A matrix of distances between cluster points is then generated with symmetric integer entries uniformly distributed between 1 and 200. Then, for each cluster, an $(m+1)\times(m+1)$ sub-matrix is generated with integer entries uniformly distributed between 1 and 40, where m is the cluster size. These distances are combined in a single matrix. An example where there are two clusters of two is

$$\begin{bmatrix} \begin{bmatrix} 0 & 25 & 15 \\ 25 & 0 & 10 \\ 15 & 10 & 0 \end{bmatrix} & \begin{matrix} 70 & - & - \\ - & - & - \\ - & - & - \end{matrix} \\ \begin{matrix} 70 & - & - \\ - & - & - \\ - & - & - \end{matrix} & \begin{bmatrix} 0 & 10 & 30 \\ 10 & 0 & 35 \\ 30 & 35 & 0 \end{bmatrix} \end{bmatrix}$$

The matrix is then replaced by the matrix of shortest paths, and the rows and columns corresponding to the cluster points are thrown out. For the above matrix, this results in

$$C = \begin{bmatrix} 0 & 10 & 105 & 125 \\ 10 & 0 & 95 & 115 \\ 105 & 95 & 0 & 35 \\ 125 & 115 & 35 & 0 \end{bmatrix}$$

For problems with outliers, first a $(0.8n+1)\times(0.8n+1)$ matrix is generated with integer entries uniformly distributed between 1 and 200. This matrix is then combined with a $(0.2n+1)\times(0.2n+1)$ matrix with integer entries uniformly distributed between 201 and 400, as shown below for $n = 5$.

$$\begin{bmatrix} 0 & 30 & 105 & 175 & 65 & - \\ 30 & 0 & 45 & 60 & 120 & - \\ 105 & 45 & 0 & 110 & 140 & - \\ 175 & 60 & 110 & 0 & 20 & - \\ 65 & 120 & 140 & 20 & 0 & 310 \\ - & - & - & - & 310 & 0 \end{bmatrix}$$

This matrix is then replaced by the matrix of shortest paths, and the row and column where the two original matrices overlapped are thrown away.

For each cost matrix, two different sets of node values are used: one where all nodes have equal values, and one where the node values are integers uniformly distributed between 1 and 10. For uniform problems, three additional sets of values are used: integers uniformly distributed between 1 and 100, integers uniformly distributed between 1 and 3, and integers obtained by rounding down a variable that is exponentially distributed with a mean of 5 and then adding 1 $\left(v_i = 1 + \lfloor x \rfloor, \text{ where } x \sim \exp(1/5)\right)$.

Each problem (defined by a cost matrix and node values) is solved using several budgets. The budgets are defined as a fraction of the cost of a complete tour, obtained using a heuristic algorithm for TSP. The farthest-insertion algorithm in combination with a two-opt routine was used to obtain an approximate TSP solution. Generally, the budgets used were 0.25, 0.50 and 0.75 times the cost of the approximate TSP solution.

Due to the large number and variety of test problems used, in the following three chapters, only highlights of the computational results will be presented. Detailed results are available in Appendix C.

Chapter 7
UPPER BOUNDS

Bounding methods are methods which can be used to establish a range within which the optimal value for a problem must lie. For a minimization problem, *lower bounding methods* are used to generate *lower bounds* on the value of any feasible solution. Thus, if a feasible solution is found with a value equal to a lower bound, it must be optimal. The value of a solution found by a heuristic algorithm is used as an *upper bound.* Similarly, for maximization problems, *upper bounding methods* are used to generate upper bounds on the value of any feasible solution. A heuristic algorithm is used to find a lower bound. Bounding methods generate an optimistic estimates of the optimal value. A good bounding method generates a bound which is close to the optimal value.

Bounding methods generally work by calculating the optimal solution to a relaxation of the original problem. For example, if the problem is

$$\max z(x), \qquad \text{subject to } x \in S, \tag{1}$$

an upper bound may be calculated by solving the relaxation

$$\max z(x), \qquad \text{subject to } x \in T, \text{ where } S \subset T. \tag{2}$$

Since $S \subset T$, the solution to (2) must be greater than or equal to the solution to (1). Bounds that are close to the optimal value are called *tight*. Given two upper bounding methods F_1 and F_2, if, for any problem instance, the bound generated by F_1 is never greater than the bound generated by F_2, then we say that method F_1 *dominates* F_2. Suppose that the problem to be solved is (1), bounding method F_1 is

$$\max z(x), \qquad \text{subject to } x \in T_1, \text{ where } S \subset T_1,$$

and bounding method F_2 is

$$\max z(x), \qquad \text{subject to } x \in T_2, \text{ where } S \subset T_2.$$

If $T_1 \subset T$, then F_1 dominates F_2.

Bounding methods and the bounds generated by them have several uses. One type of heuristic algorithm is one that enumerates increasingly good feasible solutions, stopping when a solution is found with a value within some specified percentage of the bound. This type of algorithm can be dangerous, unless we can guarantee that a feasible solution exists within that percentage of the upper bound. Often, as with Christofides' algorithm [**Ch**] for TSP, heuristic algorithms are based on bounding methods. This is especially true for those heuristics which have performance guarantees. As we will discuss later, branch-and-bound algorithms are often used for combinatorial optimization problems, and their success is highly dependent on the use of good bounding methods. Another use for bounds is in the evaluation of heuristics. In order to empirically evaluate a heuristic, based on the closeness of its solution value to the optimal value, we must know the optimal value for the problem. However, heuristic algorithms are of most interest in cases where the true optimum cannot be obtained. In these cases, we may choose to evaluate heuristics by comparing their solution values to bounds on the optimal solution.

We present several methods for obtaining upper bounds for CCTSP. Some of these methods can be shown to dominate others. However, methods which generate looser bounds may still be of interest. Generally, computing a tighter upper bound is more difficult and requires greater time. It is sometimes desirable to generate an upper bound quickly and easily rather than make it as tight as possible.

7.1 Knapsack Bounds

One method of computing an upper bound for CCTSP is to solve the following relaxation of formulation IP_1 (defined in Section 3.1):

$$\max \sum_{i=1}^{n}\sum_{j=1}^{n} v_i x_{ij} \qquad (F_1)$$

$$\text{subject to:} \quad \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij} x_{ij} \le B$$

$$\sum_{j=1}^{n} x_{ij} \le 1 \qquad i = 2,3,\ldots,n$$

$$\sum_{j=2}^{n} x_{1j} = 1$$

$$x_{ii} = 0 \qquad \text{for all } i$$

$$x_{ij} \in \{0,1\} \qquad \text{for all } i \text{ and all } j$$

An upper bound on F_1, and hence on CCTSP, can be found by solving the linear programming relaxation of F_1 using a greedy algorithm. For each node i, let

$$w_i = \min_{j \neq i}\{c_{ij}\}$$

and relabel the nodes such that

$$v_2/w_2 \ge v_3/w_3 \ge \ldots \ge v_n/w_n.$$

Let

$$m = \max_{k}\left\{k \mid \sum_{i=1}^{k} w_i \le B\right\}.$$

Then an optimal solution to the linear programming relaxation of F_1 is

$$x_{ij} = 1 \quad \text{for } i \le m \text{ and } j = \operatorname{argmin}_{j \neq i}\{c_{ij}\}$$

$$x_{m+1,j} = c_{m+1,j} \Big/ \left(B - \sum_{i=1}^{m} w_i\right) \quad \text{for } j = \operatorname{argmin}_{j \neq m+1}\{c_{m+1,j}\}$$

and $x_{ij} = 0$ otherwise.

Note that this is equivalent to solving the linear programming relaxation of the following knapsack problem, where the weights w_i are as defined above:

$$\max \quad v_1 + \sum_{i=2}^{n} v_i y_i \qquad \text{(KP)}$$

$$\text{subject to:} \quad \sum_{i=2}^{n} w_i y_i \le B - w_1$$

$$y_i \in \{0,1\} \quad \text{for all } i.$$

In the case where all nodes have equal value, with the possible exception of node 1, the problems F_1 and KP can be solved exactly. The optimal solution is as described above, except $x_{m+1,j} = 0$ for all j.

Laporte and Martello **[LM]** describe a more general upper bounding method based on the knapsack problem. They show that the problem KP where

$$w_j = \alpha \min_{i \neq j}\{c_{ij}\} + (1-\alpha)\min_{k \neq j}\{c_{jk}\} \quad \text{for all } j$$

provides an upper bound for CCTSP for any specified value of α with $0 \le \alpha \le 1$. Computing this bound is equivalent to solving the following relaxation, F_2, of IP_1, which is derived by replacing each x_{ij} in IP_1 with $\alpha x_{ij}^1 + (1-\alpha)x_{ij}^2$ where $x_{ij}^1 = x_{ij}^2$ and then dropping the constraint $x_{ij}^1 = x_{ij}^2$ as well as the subtour elimination constraints.

$$\max \sum_{i=1}^{n}\sum_{j=1}^{n} v_i x_{ij}^1 \qquad (F_2)$$

$$\text{subject to:} \quad \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}\left((1-\alpha)x_{ij}^1 + \alpha x_{ij}^2\right) \le B$$

$$\sum_{j=1}^{n} x_{ij}^1 \le 1 \qquad i = 2,3,\ldots,n$$

$$\sum_{j=2}^{n} x_{1j}^1 = 1$$

$$\sum_{i=1}^{n} x_{ij}^2 - \sum_{k=1}^{n} x_{jk}^1 = 0 \qquad j = 1,2,\ldots,n$$

$$x_{ii}^k = 0 \qquad \text{for all } i, k$$

$$x_{ij}^k \in \{0,1\} \qquad \text{for all } i, j, k$$

We note that the value of α that results in the tightest bound is problem specific. For example, let $v = (1,1,\ldots,1)$, $B = 6$, and

$$C_1 = \begin{bmatrix} \infty & 2 & 2 & 2 & 2 \\ 1 & \infty & 2 & 2 & 2 \\ 1 & 2 & \infty & 2 & 2 \\ 1 & 2 & 2 & \infty & 2 \\ 1 & 2 & 2 & 2 & \infty \end{bmatrix} \text{ and } C_2 = \begin{bmatrix} \infty & 1 & 1 & 1 & 1 \\ 2 & \infty & 2 & 2 & 2 \\ 2 & 2 & \infty & 2 & 2 \\ 2 & 2 & 2 & \infty & 2 \\ 2 & 2 & 2 & 2 & \infty \end{bmatrix}.$$

Using values of 0 and 1/2 for α gives upper bounds of 3.5 and 4 respectively for matrix C_1, and bounds of 5 and 4 respectively for matrix C_2. In both cases, the optimal solution to CCTSP has a value of 3. Experiments by Laporte and Martello indicate that, on the average, setting $\alpha = 1/2$ provides the best bounds.

Since KP is, itself, an NP-hard problem, Laporte and Martello obtain an upper bound for CCTSP by computing an upper bound to KP using the method of Martello and Toth **[MT]**. To compute this bound, assume the nodes are labeled such that

$$v_2/w_2 \geq v_3/w_3 \geq \ldots \geq v_n/w_n$$

and that

$$\sum_{j=1}^{l} w_j \leq B < \sum_{j=1}^{l+1} w_j.$$

Node $l+1$ is either included in the optimal subtour or not included. This gives bounds UB_1 and UB_2 respectively, where

$$UB_1 = \sum_{j=1}^{l+1} v_j - \left(\sum_{j=1}^{l+1} w_j - B \right) \frac{v_l}{w_l}$$

$$UB_2 = \sum_{j=1}^{l} v_j + \left(B - \sum_{j=1}^{l} w_j \right) \frac{v_{l+2}}{w_{l+2}}.$$

An upper bound on KP is then

$$UB = \max\{UB_1, UB_2\}.$$

Henceforth, we will refer to the upper bounding method defined by Laporte and Martello as the KP-bound (KPB). We now describe an upper bounding method which is an improvement on KPB. This improved bound, which we call the IKP-bound (IKP), dominates KPB.

We first note that, provided the optimal solution contains more than two nodes, $x_{ij} + x_{ji} \leq 1$ for all i, j. Thus, we let the weights on the nodes be

$$w_j = \min_{\substack{i \neq j, k \neq j \\ i \neq k}} \left\{ \alpha c_{ij} + (1-\alpha) c_{jk} \right\} \quad \text{for all } j.$$

This is equivalent to adding the constraint

$$x^2_{ij} + x^1_{ji} \leq 1 \quad \text{for all } i, j$$

to F_2. When C is a symmetric matrix, this always results in larger weights, and, hence, tighter bounds, than KPB. Furthermore, if C is symmetric, then, for the weights defined above, the optimal value of α is 1/2. On the other hand, when C is symmetric, the weights defined by Laporte and Martello result in

$$w_j = \alpha \min_{i \neq j} \left\{ c_{ij} \right\} + (1-\alpha) \min_{k \neq j} \left\{ c_{jk} \right\} = \min_{k \neq j} \left\{ c_{jk} \right\} \quad \text{for all } j$$

regardless of the value of α. Henceforth, unless otherwise stated, we will set $\alpha = 1/2$.

We also note that a node cannot be in a feasible solution unless it is "reachable" from node 1. The set of reachable nodes is

$$S = \left\{ j \mid c'_{1j} + c'_{j1} \leq B \right\} \cup \{1\}$$

where c'_{ij} is the length of the shortest path (treating costs as distances) from node i to node j. The node weights are, then,

$$w_j = \min_{\substack{i, k \in S \setminus j \\ i \neq k}} \left\{ \alpha c_{ij} + (1-\alpha) c_{jk} \right\} \quad \text{for } j \in S,$$

$$w_j = \infty \quad \text{for } j \notin S.$$

A second improvement is obtained by computing a tighter bound on the knapsack problem. In the method of Martello and Toth, the fractions

$$\left(\sum_{j=1}^{l+1} w_j - B\right)\frac{1}{w_l} \quad \text{and} \quad \left(B - \sum_{j=1}^{l} w_j\right)\frac{1}{w_{l+2}}$$

used in computing UB_1 and UB_2, respectively, may be greater than 1. With this in mind, we replace UB_1 and UB_2 with

$$UB_1 = v_{l+1} + \sum_{j=1}^{k} v_j + \left(B - w_{l+1} - \sum_{j=1}^{k} w_j\right)\frac{v_{k+1}}{w_{k+1}}$$

$$\text{where } k(<l) \text{ satisfies } \sum_{j=1}^{k} w_j \le B - w_{l+1} < \sum_{j=1}^{k+1} w_j$$

$$\text{and } UB_2 = \sum_{j=1}^{l} v_j + \sum_{j=l+2}^{k} v_j + \left(B - \sum_{j=1}^{l} w_j - \sum_{j=l+2}^{k} w_j\right)\frac{v_{k+1}}{w_{k+1}}$$

$$\text{where } k(\ge l+1) \text{ satisfies } \sum_{j=l+2}^{k} w_j \le B - \sum_{j=1}^{l} w_j < \sum_{j=l+2}^{k+1} w_j.$$

Since the weights w_j were calculated assuming the optimal solution contains more than two nodes, the case where it contains exactly two nodes must also be considered. To account for this case, we let

$$k = \underset{i \ne 1}{\text{argmax}}\{v_i \mid c_{1i} + c_{i1} \le B\}.$$

We then have

$$UB = \max\{v_1 + v_k, UB_1, UB_2\}.$$

We make two additional observations. First, in the case where all nodes have equal values, the knapsack problems associated with KPB and IKP can be solved exactly. Second, when $\alpha = 0$, the knapsack problems associated with KPB and IKP are equivalent to F_1.

An even tighter bound (TKP) can be computed by specifying the nodes preceding and following node 1 in the subtour. Let r and s denote the nodes preceding and following node 1, respectively, and let

$$S = \left\{ j | c'_{1j} + c'_{j1} \leq B \right\}$$

$$\text{and } T = \left\{ (r,s) | c_{1s} + c'_{sr} + c_{r1} \leq B \right\}$$

where c'_{ij} is the length of the shortest path (treating costs as distances) from node i to node j. Then

$$w_j(r,s) = \min_{\substack{i \in S \setminus r \\ k \in S \setminus \\ i \neq k}} \left\{ \alpha c_{ij} + (1-\alpha) c_{jk} \right\} \quad \text{for } j \in S \setminus r,s$$

$$w_1(r,s) = \min_{i,k \in S \setminus r,s} \left\{ \alpha c_{1r} + (1-\alpha) c_{sk} \right\} + c_{r1} + c_{1s}$$

$$w_j(r,s) = \infty \quad \text{for } j \notin \{1\} \cup S \setminus r,s$$

and

$$\text{UB}(r,s) = \max(v_1 + v_r + v_s, \text{UB}_1, \text{UB}_2)$$

where UB_1 and UB_2 are calculated as in IKP. Again, the case where the optimal solution contains exactly two nodes must also be considered. Accounting for this case, we have

$$\text{UB} = \max_{(r,s) \in T} (v_1 + v_k, \text{UB}(r,s))$$

where

$$k = \operatorname{argmax}_{i \neq 1} \left\{ v_i | c_{1i} + c_{i1} \leq B \right\}.$$

Computing TKP requires $O(n^3 \log n)$ time, while KPB and IKP require only $O(n^2)$.

7.2 The Parametric Assignment Bound

An upper bound on the number of nodes which can be in any feasible solution to CCTSP can be found by solving a parametric assignment problem. For the case where all nodes have equal value, this results in an upper bound on the value of an optimal solution to CCTSP. For cases where nodes do not have the same value, we can still use the parametric assignment problem to derive an upper bound on the value of an optimal

solution. We first obtain a bound m on the number of nodes in an optimal solution. We then sum the values of the m most valuable nodes. Henceforth, we will refer to the bound based on the parametric assignment problem as the parametric assignment bound (PAB). The bound PAB is designed for problems where all nodes have equal values, but, as noted above, can also be applied to the general problem. However, we do not expect the bound to be very good in the case where nodes do not have equal values.

To compute PAB, we use the following relaxation, F_3, of IP_1, assuming all nodes have equal value.

$$\max \sum_{i=1}^{n}\sum_{j=1}^{n} x_{ij} \qquad (F_3)$$

$$\text{subject to: } \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} \le B$$

$$\sum_{j=1}^{n} x_{ij} \le 1 \qquad \text{for } i = 2,3,\ldots,n$$

$$\sum_{i=1}^{n} x_{ij} \le 1 \qquad \text{for } j = 2,3,\ldots,n$$

$$\sum_{j=2}^{n} x_{1j} = \sum_{i=2}^{n} x_{i1} = 1$$

$$x_{ii} = 0 \qquad \text{for all } i$$

$$x_{ij} \in \{0,1\} \qquad \text{for all } i,j$$

This is equivalent to forming a bipartite graph with n nodes on each side and finding a maximum cardinality assignment subject to the cost constraint.

We define the parametric assignment problem (PAP) as the problem where $n-m$ nodes from each side may be assigned, without cost, to dummy nodes on the opposite side.

$$z_m = \min \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \qquad \text{(PAP)}$$

$$\text{subject to: } \sum_{j=1}^{n+1} x_{ij} = 1 \qquad \text{for } i = 1,2,\ldots,n$$

$$\sum_{i=1}^{n+1} x_{ij} = 1 \qquad \text{for } j = 1,2,\ldots,n$$

$$\sum_{j=2}^{n} x_{1j} = \sum_{i=2}^{n} x_{i1} = 1$$

$$\sum_{j=1}^{n+1} x_{n+1,j} = \sum_{i=1}^{n+1} x_{i,n+1} = n - m$$

$$x_{ii} = 0 \qquad \text{for } i = 1,2,\ldots,n$$

$$x_{ij} \in \{0,1\} \qquad \text{for all } i, j$$

Several efficient algorithms (e.g. the Hungarian Method [PS]) are available for solving assignment problems such as PAP.

Suppose the optimal value of F_3 is m^*. Then, $z_m \leq B$ for $m \leq m^*$, and $z_m > B$ for $m > m^*$. Thus, we can solve F_3 by adjusting the parameter m in PAP until we find m^* such that $z_{m^*} \leq B$ and $z_{m^*+1} > B$. This gives an upper bound on the number of nodes in any feasible solution to CCTSP. An upper bound on the value of any solution to CCTSP is

$$\text{PAB} = \sum_{i=1}^{m^*} v_i,$$

where the nodes are labelled such that $v_2 \leq v_3 \leq \ldots \leq v_n$. In the case where all nodes have equal value, this reduces to $\text{PAB} = m^*$.

Note that the constraints for F_1 are a subset of the constraints for F_3. Thus, in the case of CCTSP where all nodes have equal value, PAB dominates both KPB and IKP when $\alpha = 0$. However, this is not necessarily the case for other values of α.

7.3 The Cost-Constrained Assignment Bound

We now present an upper bounding method which uses a relaxation of the formulation IP_2 (defined in Section 3.1). We replace the objective function

$$\max \sum_{i=1}^{n} v_i(1-x_{ii})$$

with

$$\max\ V - \sum_{i=1}^{n}\sum_{j=1}^{n} p_{ij}x_{ij}$$

where

$$V = \sum_{i=1}^{n} v_i \text{ and } p_{ij} = \begin{cases} 0 & \text{if } i \neq j \\ v_i & \text{if } i = j. \end{cases}$$

We force node 1 to be in the solution by setting $c_{11} = \infty$, and set $c_{ii} = 0$ for all $i \neq 1$. Dropping the subtour elimination constraints and the cost constraint leads to a standard assignment problem. We drop the subtour elimination constraints but retain the cost constraint. We call this formulation the cost-constrained assignment problem (CAP). Since CAP is a relaxation of CCTSP, its solution provides an upper bound, the cost-constrained assignment bound (CAB), for CCTSP.

$$\text{CAB} = \max\ V - \sum_{i=1}^{n}\sum_{j=1}^{n} p_{ij}x_{ij} \qquad \text{(CAP)}$$

$$\text{subject to: } \sum_{i=1}^{n} x_{ij} = 1 \qquad \text{for } j = 1,2,\ldots,n$$

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad \text{for } i = 1,2,\ldots,n$$

$$x_{ij} \in \{0,1\} \qquad \text{for all } i,j$$

Note that any feasible solution to CAP is also a feasible solution to F_2 and F_3. Thus, CAB dominates IKP and PAB.

Since CAP is, itself, an NP-hard problem, we compute an upper bound on CAP using the method of Lagrange multipliers **[Fi]**. Incorporating the cost constraint into the objective function with a Lagrange multiplier, $\lambda \geq 0$, results in the following assignment problem:

$$z(\lambda) = \max\ V - \sum_{i=1}^{n}\sum_{j=1}^{n} p_{ij}x(\lambda)_{ij} - \lambda\left(\sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x(\lambda)_{ij} - B\right) \qquad \text{CAP}(\lambda)$$

$$\text{subject to: } \sum_{i=1}^{n} x(\lambda)_{ij} = 1 \qquad \text{for } j = 1,2,\ldots,n$$

$$\sum_{j=1}^{n} x(\lambda)_{ij} = 1 \qquad \text{for } i = 1,2,\ldots,n$$

$$x(\lambda)_{ij} \in \{0,1\} \qquad \text{for all } i,j\,.$$

We can simplify the objective function to

$$z(\lambda) = V + \lambda B - \min \sum_{i=1}^{n}\sum_{j=1}^{n}\left(p_{ij} + \lambda c_{ij}\right)x(\lambda)_{ij} = V + \lambda B - \min(px + \lambda cx).$$

For a fixed value of λ, CAP(λ) is a standard assignment problem. To obtain the best upper bound on CAP, we desire λ^* such that

$$z(\lambda^*) = \min_{\lambda} z(\lambda).$$

Our upper bound on CCTSP is then

$$\text{CAB}\lambda = z(\lambda^*)$$

where $z(\lambda^*)$ is the optimal value of CAP(λ^*). Note that, while CAB dominates IKP and PAB, this is not necessarily the case for CABλ.

As shown by Everett [**Ev**],

$$cx^*(\lambda) = \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x^*(\lambda)_{ij}$$

is monotonically decreasing in λ, where $x^*(\lambda)$ is an optimal solution to CAP(λ). However, since the feasible region of CAP(λ) is non-convex, there may be "gaps" in the value of $cx^*(\lambda)$ as λ increases. If there exists a λ^* such that $cx^*(\lambda^*) = B$, then

$$z(\lambda^*) = \min_{\lambda} z(\lambda)$$

and $x^*(\lambda)$ solves CAP. However, in most cases, this does not occur.

We can replace the objective function in CAP(λ) with

$$z'(\lambda) = \min \sum_{i=1}^{n} \sum_{j=1}^{n} \left(\frac{1}{\lambda} p_{ij} + c_{ij} \right) x(\lambda)_{ij}$$

We then have

$$z(\lambda) = V + \lambda B - \lambda z'(\lambda)$$

Using $z'(\lambda)$ instead of $z(\lambda)$ results in the cost matrix

$$\frac{1}{\lambda} P + C = \begin{bmatrix} \infty & c_{12} & \cdots & c_{1n} \\ c_{21} & \frac{1}{\lambda} v_2 & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & \frac{1}{\lambda} v_n \end{bmatrix}$$

where only the diagonal elements depend on λ. This facilitates reoptimization when the value of λ is changed.

Let β be the basis associated with an optimal solution $x^*(\lambda)$ to CAP(λ), and let u and w be the optimal dual variables associated with β. Then, according to the complementary slackness theorem **[Mu]**

$$u_i + w_j \leq \begin{cases} \frac{1}{\lambda} v_i & \text{for all } i, j,\ i = j,\ i \neq 0 \\ c_{ij} & \text{for all } i, j,\ i \neq j \end{cases}$$

$$\text{and } u_i + w_j = \begin{cases} \frac{1}{\lambda} v_i & \text{if } (i,j) \in \beta,\ i = j \\ c_{ij} & \text{if } (i,j) \in \beta,\ i \neq j\,. \end{cases}$$

Define a second set of variables u^* and w^* such that

$$u_i^* + w_j^* = \begin{cases} v_i & \text{if } (i,j) \in \beta,\ i = j \\ 0 & \text{if } (i,j) \in \beta,\ i \neq j. \end{cases}$$

Suppose we increase $1/\lambda$ by Δ. Let

$$u_i' = u_i + \Delta u_i^*$$

$$\text{and } w_j' = w_j + \Delta w_j^*.$$

Then,

$$u_i' + w_j' = \begin{cases} \left(\frac{1}{\lambda} + \Delta\right) v_i & \text{if } (i,j) \in \beta,\ i = j \\ c_{ij} & \text{if } (i,j) \in \beta,\ i \neq j \end{cases}$$

Thus, the solution $x^*(\lambda)$ remains optimal as long as

$$u_i + w_j + \Delta\left(u_i^* + w_i^*\right) \leq c_{ij} \quad \text{for all } i,\ j,\ i \neq j$$

$$\text{and} \quad u_i + w_i + \Delta\left(u_i^* + w_i^*\right) \leq \left(\frac{1}{\lambda} + \Delta\right) v_i \quad \text{for all } i$$

or, equivalently, as long as the increase in $1/\lambda$ does not exceed

$$\Delta^+ = \min\left(\min_{\substack{i,j,i\neq j \\ u_i^* + w_j^* > 0}} \frac{c_{ij} - u_i - w_j}{u_i^* + w_j^*},\ \min_{\substack{i \\ u_i^* + w_i^* - v_i > 0}} \frac{\frac{1}{\lambda} v_i - u_i - w_i}{u_i^* + w_i^* - v_i} \right).$$

Similarly, suppose we decrease $1/\lambda$ by Δ. Let

$$u_i' = u_i - \Delta u_i^*$$

$$\text{and } w_j' = w_j - \Delta w_j^*.$$

Then,

$$u_i' + w_j' = \begin{cases} \left(\frac{1}{\lambda} - \Delta\right) v_i & \text{if } (i,j) \in \beta,\ i = j \\ c_{ij} & \text{if } (i,j) \in \beta,\ i \neq j \end{cases}$$

Thus, the solution $x^*(\lambda)$ remains optimal as long as

$$u_i + w_j - \Delta\left(u_i^* + w_j^*\right) \leq c_{ij} \quad \text{for all } i,\ j,\ i \neq j$$

$$\text{and} \quad u_i + w_i - \Delta\left(u_i^* + w_j^*\right) \leq \left(\frac{1}{\lambda} - \Delta\right) v_i \quad \text{for all } i,$$

or, equivalently, as long as the decrease in $1/\lambda$ does not exceed

$$\Delta^- = \min\left(\min_{\substack{i,j,i\neq j \\ u_i^* + w_j^* < 0}} \frac{u_i + w_j - c_{ij}}{u_i^* + w_j^*},\ \min_{\substack{i \\ v_i - u_i^* - w_i^* > 0}} \frac{\frac{1}{\lambda} v_i - u_i - w_i}{v_i - u_i^* - w_i^*} \right).$$

Let

$$\lambda^- = \frac{1}{\frac{1}{\lambda} + \Delta^+} \quad \text{and} \quad \lambda^+ = \frac{1}{\frac{1}{\lambda} - \Delta^-}$$

Then, $x^*(\lambda)$ is an optimal solution to CAP(λ') for all λ' in the interval (λ^-, λ^+).

We now present a method for finding λ^*. We start with λ_1 and λ_2 such that $cx^*(\lambda_1) > B$ and $cx^*(\lambda_2) < B$, where $x^*(\lambda)$ is the optimal solution to CAP(λ). Equations which can be used to compute initial values of λ_1 and λ_2 are given in Appendix A. After solving CAP(λ_1) and CAP(λ_2), we compute λ_1^+ and λ_2^-. We let $\lambda_3 = (\lambda_1^+ + \lambda_2^-)/2$ and solve CAP(λ_3). If $cx^*(\lambda_3) < B$, we replace λ_2 with λ_3. If $cx^*(\lambda_3) > B$ we replace λ_1 with λ_3. We continue this process until we find λ_1 and λ_2 such that $\lambda_1^+ \geq \lambda_2^-$. Typically, we will find $\lambda_1^+ = \lambda_2^-$. We then select any λ^* in the overlapping region of the intervals (λ_1, λ) and (λ_2^-, λ_2). Associated with this value of λ^* are two optimal solutions, $x_1^*(\lambda^*) = x^*(\lambda_1)$ and $x_2^*(\lambda^*) = x^*(\lambda_2)$, to CAP($\lambda^*$),with $cx_1^*(\lambda^*) > B$ and $cx_2^*(\lambda^*) < B$.

Theorem 7.1: The value λ^* found by the above method solves $z(\lambda^*) = \min_{\lambda} z(\lambda)$, where $z(\lambda)$ is the optimal solution to CAP(λ).

Proof: Suppose $\bar{\lambda} > \lambda^*$. Since $B - cx_2^*(\lambda^*) > 0$, we have

$$z(\bar{\lambda}) = V + \bar{\lambda}B - \min(px + \bar{\lambda}cx) \geq V + \bar{\lambda}B - (px_2^*(\lambda^*) + \bar{\lambda}cx_2^*(\lambda^*))$$
$$= V - px_2^*(\lambda^*) + \bar{\lambda}(B - cx_2^*(\lambda^*)) > V - px_2^*(\lambda^*) + \lambda^*(B - cx_2^*(\lambda^*)) = z(\lambda^*).$$

Similarly, suppose $\bar{\lambda} < \lambda^*$. Since $cx_1^*(\lambda^*) - B > 0$, we have

$$z(\bar{\lambda}) = V + \bar{\lambda}B - \min(px + \bar{\lambda}cx) \geq V + \bar{\lambda}B - (px_1^*(\lambda^*) + \bar{\lambda}cx_1^*(\lambda^*))$$
$$= V - px_1^*(\lambda^*) - \bar{\lambda}(cx_1^*(\lambda^*) - B) > V - px_1^*(\lambda^*) - \lambda^*(cx_1^*(\lambda^*) - B) = z(\lambda^*).$$

Thus, $z(\lambda^*) = \min_{\lambda} z(\lambda)$. □

This method of computing an upper bound is based on the ideas of Gensch **[Ge]**. However, Gensch's method of finding λ^* contains substantial errors and lacks efficiency.

Furthermore, Gensch claims that $x_2^*(\lambda^*)$ actually solves CAP. A counterexample to this claim is presented in Appendix B.

Kataoka and Morito [KM] present a method of computing an upper bound where CAP is solved exactly. They use a branch-and-bound algorithm where the bounds are computed by solving the linear programming relaxation of CAP and branching is done by selecting a fractional x_{ij} and setting it to 0 and 1 on alternate branches. Their method of solving the linear programming relaxation also uses the Lagrangian relaxation CAP(λ), but their method of finding λ^*, $x_1^*(\lambda^*)$, and $x_2^*(\lambda^*)$ differs substantially. They solve the linear relaxation of CAP by taking a convex combination of $x_1^*(\lambda^*)$ and $x_2^*(\lambda^*)$ such that the cost constraint is satisfied with equality.

7.4 Computational Results

Computational experiments were conducted to compare the quality of the bounds generated by the previously discussed methods. The parametric assignment method was only applied to problems in which the nodes had equal values. Table 7.1 shows a selected set of representative computational results. All problems have 20 nodes and results are averaged over a sample size of 10. The full set of computational results is given in Appendix C. In general, the improved knapsack bound (IKP) performed significantly better, sometimes by a factor of 10, than Laporte and Martello's knapsack bound (KP), the parametric assignment bound (PAB), and the constrained assignment bound (CABλ). The improved knapsack bound also required less computation time than the two assignment bounds. The tighter knapsack bound (TKP) generally performed better than the improved knapsack bound, as expected. However, the difference in performance between IKP and TKP is not dramatic as it is between KP and IKP, and, in many cases, the difference in performance is not statistically significant. In addition, TKP required a great deal more computation time.Taking this into account, IKP should be the favored upper bounding method for most purposes.

The difference in performance between the upper bounding methods decreases as the problem budget (B) increases. For small budgets, there is a dramatic difference between the bounds obtained by IKP and the bounds obtained by KP, PAB, and CABλ.

Problem type	B	IKP	KP	TKP	PAB	CABλ
Euclidean		0.00	0.00	0.03	0.11	0.09
uniform	0.25	11.33	**119.00**	7.33	**104.33**	**104.33**
Euclidean		0.00	0.00	0.18	0.12	0.12
uniform	0.50	33.72	**63.20**	30.59	**52.67**	**52.67**
Euclidean		0.00	0.01	0.27	0.06	0.08
uniform	0.75	16.67	**25.50**	16.04	**23.67**	**23.67**
Euclidean		0.00	0.00	0.02	0.13	0.10
clusters	0.25	14.72	**209.68**	10.44	**189.37**	**216.04**
Euclidean		0.01	0.00	0.11	0.10	0.10
clusters	0.50	58.51	**112.84**	**39.80**	**102.60**	**102.60**
Euclidean		0.00	0.00	0.25	0.00	0.03
clusters	0.75	29.58	29.58	29.58	29.58	29.58
non-Eucl.		0.00	0.00	0.28	0.10	0.13
uniform	0.75	10.76	12.49	10.76	**6.83**	**6.83**

Table 7.1. Selected computational results comparing the performance of upper bounding methods using test problems with 20 nodes. In all problems, the nodes were given equal values. Results are averaged over a sample size of 10. The numbers in larger print are the average percent errors from optimality of the upper bounds while the numbers in smaller print are the average computation times. Bold type indicates that, when compared with IKP, the difference in performance was statistically significant at the 95% level.

These differences are also more dramatic for problems with clusters than for uniform problems. On the other hand, when the budget is large, the difference in performance between IKP and the other bounds is smaller for problems with clusters than for uniform problems. For non-Euclidean problems, while IKP was superior to PAB and CABλ for small budgets, PAB and CABλ obtained tighter bounds when the budget was large. This, however, was not true for problems with clusters. In the case of clusters, IKP was always superior to PAB and CABλ.

Chapter 8
EXACT ALGORITHMS

Finding exact solutions to NP-hard problems, such as CCTSP, is a difficult, time consuming task and often computationally infeasible. However, algorithms which guarantee a true optimal solution do exist for most NP-hard problems. These algorithms generally rely on the processes of recursion and enumeration. As a worst case, an algorithm may consist of enumerating and evaluating all possible solutions. The total number of possible solutions for CCTSP is

$$\sum_{i=1}^{n-1}\binom{n-1}{i}i! = (n-1)!\sum_{i=1}^{n-1}\frac{1}{(n-1-i)!} = (n-1)!\sum_{i=0}^{n-2}\frac{1}{i!} \approx e(n-1)!$$

$$\approx e^{-(n-2)}(n-1)^{n-1}\sqrt{2\pi(n-1)} \text{ (using Stirling's approximation).}$$

The techniques of dynamic programming and branch-and-bound are often used to improve on total enumeration. Dynamic programming can be used to solve many problems that have a factorial number of feasible solutions with only an exponentially growing number of computational steps. As we will show, CCTSP is one such problem. Although branch-and-bound algorithms are equivalent to total enumeration in the worst case, the use of good branching and bounding techniques can result in algorithms that are relatively efficient. Unfortunately, even for a very efficient branch-and-bound algorithm, it is rarely possible to establish any good bounds on the computation time.

In this chapter, we present a dynamic programming algorithm and two types of branch-and-bound algorithms for CCTSP. The first branch-and-bound algorithm and the dynamic programming algorithm are closely related. A noteworthy feature of these two algorithms is that they can also be applied to the two extensions of CCTSP discussed in Chapter 3. Computational experiments were performed for the branch-and-bound algorithms only.

8.1 A Dynamic Programming Algorithm

Dynamic programming is a technique used on problems involving a sequence of interrelated decisions, where the goal is to determine the combination of decisions that maximizes overall effectiveness. Many different types of problems can be solved using a dynamic programming approach. The key characteristics of dynamic programming problems are:

(i) The problem can be divided into stages with a policy decision required at each stage.

(ii) Each stage has a state associated with it.

(iii) The policy decision at each stage determines the state associated with the next stage.

(iv) Given the current state, an optimal policy for the remaining stages is independent of the policy decisions of the previous stages. This is referred to as "the principle of optimality," and, stated differently, says that knowledge of the current state is all the information necessary to determine the optimal policy henceforth.

The Cost Constrained Traveling Salesman Problem can be viewed as a sequential decision problem. Each stage consists of visiting an additional node, and the policy decision to be made is which node, if any, to visit next. The state at each stage is specified by the set of nodes which have already been visited (S) and the node which was visited last (l). Thus, if the current state is (S,l) and the policy decision is to visit node $k \notin S$ next, then the state at the next stage is $(S+k,k)$. Given the current state, the optimal sequence of remaining nodes is independent of the sequence used to get to that state.

We have developed a dynamic programming algorithm for CCTSP based on Held and Karp's **[HK]** dynamic programming algorithm for TSP. Their algorithm is based on the following recursion equations:

> Given $S \subseteq \{2,3,\ldots,n\}$ and $l \in S$, let $C(S,l)$ be the minimum cost of starting at node 1 and visiting all nodes in the set S, terminating at node l. Then

$$\text{(a)}\quad C(\{l\},l) = c_{1l}, \quad \text{for all } l \tag{1}$$

$$\text{and (b)}\quad C(S,l) = \min_{m \in S-l}\left[C((S-l),m) + c_{ml}\right].$$

The optimal value is then

$$\min_{l \in \{2,3,\ldots,n\}}\left[C(\{2,3,\ldots,n\},l) + c_{l1}\right].$$

In our algorithm for CCTSP, we use the same definition of $C(S,l)$ and the same recursion equations. We define

$$V(S,l) = \sum_{i \in S} v_i + v_1.$$

Then the optimal value to CCTSP is

$$V_{\text{opt}} = \max_{\{(S,l)\,|\,C(S,l)+c_{l1} \le B\}} V(S,l). \tag{2}$$

We know that a partial permutation $(1,i_2,i_3,\ldots,i_m)$ is optimal if and only if

$$\left(\{i_2,i_3,\ldots,i_m\},i_m\right) = \arg\max_{\{(S,l)\,|\,C(S,l)+c_{l1} \le B\}} V(S,l) \tag{3}$$

and, for $2 \le p \le m-1$,

$$C\left(\{i_2,i_3,\ldots,i_p,i_{p+1}\},i_{p+1}\right) = C\left(\{i_2,i_3,\ldots,i_p\},i_p\right) + c_{i_p i_{p+1}}. \tag{4}$$

In the first phase, equation (1) is used recursively to compute the quantities $C(S,l)$ and then V_{opt} is computed from (2). In the second phase, equations (3) and (4) are used to compute an optimal solution.

The fundamental operations employed in the computations are additions and comparisons. The number of computations in the first phase is on the order of

$$\sum_{k=2}^{n-1} k(k-1)\binom{n-1}{k} = (n-1)(n-2)\sum_{k=0}^{n-3}\binom{n-3}{k} = (n-1)(n-2)2^{n-3}.$$

The number of computations in the second phase is at most on the order of

$$\sum_{k=2}^{n-1} k = [n(n-1)/2] - 1.$$

Thus, the growth rate of this algorithm is $O(2^n)$. Since each number $C(S,l)$ must be stored, the number of storage locations required is

$$\sum_{k=1}^{n-1} k\binom{n-1}{k} = (n-1)\sum_{k=0}^{n-2}\binom{n-2}{k} = (n-1)2^{n-2}.$$

In many cases, the computation time can be significantly improved since it is not necessary to compute $C(S,l)$ for every possible (S,l). At any point in our computations, we can divide the states (S,l) into two sets: *candidates*, those for which $C(S,l) \leq B$, and *candidate solutions*, those for which $C(S,l) + c_{l1} \leq B$. If $C(S,l) > B$, then, for any $k \notin S$,

$$C(S,l) + c_{lk} > B \text{ and } C(S,l) + c_{l1} > B.$$

Thus, any (S,l) that is not a candidate need not be considered in any further computations and, since it is also not a candidate solution, may be deleted. If the triangle inequality holds for the cost matrix C, then

$$C(S,l) + c_{lk} + c_{k1} \geq C(S,l) + c_{l1}$$

and any (S,l) that is not a candidate solution may be deleted. If no candidates remain to be used in the computation of $C(S+k,k)$, then $C(S+k,k)$ need not be computed. When no additional $C(S+k,k)$s can be computed, the candidate solutions are examined to find V_{opt}. If (S,l) is a candidate solution, and (S,k) is a candidate solution, where $S' \subset S$, then (S,k) need not be considered in the computation of V_{opt}. These observations can greatly reduce the number of computations required, especially in cases where the optimal solution does not contain all n nodes.

8.2 A Branch-and-Bound Algorithm

Branch-and-bound algorithms are implicit enumeration techniques which iteratively reduce the number of feasible solutions that must be examined. They rely on the process of repeatedly breaking the set of feasible solutions into subsets (branching), and calculating bounds on the value of all feasible solutions contained within them (bounding). During the branch-and-bound process, the best feasible solution found thus far is called the *incumbent*. A heuristic algorithm is often used to generate an initial incumbent. For a

maximization problem, the set of all feasible solutions is partitioned into two or more subsets and, for each subset, an upper bound on the objective function is obtained for the solutions within that subset. If the upper bound for a subset is lower than the value of the incumbent, then that subset cannot contain an optimal solution and is *fathomed* — excluded from further consideration. A subset is also fathomed if it is shown to contain no feasible solutions or if the best feasible solution within the subset has been found. In the latter case, if the value of the solution exceeds that of the incumbent, the solution replaces the incumbent. A *branching rule* is then used to select one of the remaining subsets and partition it further into two or more new subsets. The process is repeated until there are no remaining subsets, i.e. all subsets have been fathomed. The success of a branch-and-bound algorithm is highly dependent on starting with an incumbent solution that is close to optimal and on the tightness of the bounding function used.

Branch-and-bound methods are a common technique for solving integer linear programs. For example, one well-known branch-and-bound algorithm for solving such problems is that of Dakin [Da]. In his method, the Simplex (or Dual-Simplex) method is used to solve linear programming relaxations of subproblems of the original problem. These solutions provide the bounds. Branching is done by selecting a fractional variable in the linear programming solution and generating two new subproblems, one where the fractional value is cut off from above by the addition of an inequality constraint, and one where the fractional value is cut off from below. Generally, this type of branch-and-bound algorithm does not exploit any special combinatorial structure of the problem and, thus, could, in principle, be applied to virtually all linear integer programming problems, including CCTSP.

We present a branch-and-bound algorithm, based on the method of Laporte and Martello [LM], that does exploit the structure of the problem. In their algorithm, Laporte and Martello partition the set of feasible solutions by specifying the initial sequence of nodes in a subtour. Initially, the specified sequence consists of node 1 only. A subset of feasible solutions is partitioned by adding a node to the end of the specified initial sequence. One partition is formed for each node not in the current specified sequence. If a

specified sequence has a value greater than that of the incumbent and can be turned into a subtour without violating the cost constraint, the incumbent is replaced with the subtour corresponding to the specified sequence. A subset of feasible solutions is fathomed if the cost of the specified node sequence exceeds the budget, or if an upper bound on the value of a solution containing the specified sequence is less than the value of the incumbent. A subset is also fathomed if it is shown that no additional nodes can be added to the sequence without violating the cost constraint, or if the specified sequence contains all n nodes. Upper bounds are computed using the knapsack bound (KPB) described in Chapter 7, where node 1 is a "super-vertex," corresponding to the specified initial node sequence.

In our branch-and-bound algorithm, we use the same partitioning method and fathoming rules as Laporte and Martello, but replace their upper bounding method with the improved knapsack bound (IKP) presented in Section 7.1. We experimented with using the tighter knapsack bound (TKP) whenever a subset could not be fathomed based on IKP. Although doing this resulted in significantly smaller branch-and-bound trees, the excess time required to compute TKP resulted in an overall increase in computation time. To generate their initial incumbent solution, Laporte and Martello use two heuristic algorithms, one based on the nearest neighbor TSP algorithm and one based on the cheapest insertion TSP algorithm, and select the better solution. In our method, we begin with an incumbent produced by the new heuristic algorithm for CCTSP presented in the following chapter.

8.3 An Alternate Branch-and-Bound Approach

An alternate branch-and-bound approach is based on the ideas of Gensch **[Ge]**. This approach uses the cost-constrained assignment problem for computing upper bounds and uses subtour elimination for branching. Gensch proposed a branch-and-bound algorithm where upper bounds are computed by solving the cost-constrained assignment problem (CAP) using Lagrangean relaxation, and, when the resulting assignment consists of more than subtour, partitioning is done using Garfinkel's procedure for subtour elimination **[Ga73]**. Due to a number of errors, including the method of solving CAP (discussed in Section 7.3), Gensch's algorithm does not guarantee an optimal solution.

Using the same ideas, Kataoka and Morito [KM] developed a branch-and-bound algorithm that does guarantee an optimal solution. They use Lagrangean relaxation to find an optimal solution to the linear programming relaxation of CAP. This is then used as the bounding method in a branch-and-bound algorithm for CAP, where partitioning is done by setting a fractional variable to 0 and 1, respectively. If the solution to CAP has a value greater than the incumbent but is not feasible for CCTSP, that is, if it contains a subtour that includes more than a single node and does not include node 1, partitioning is done by selecting an arc in this subtour and creating two subproblems, one where the arc is always used and one where the arc is excluded. Otherwise, the subproblem is fathomed. This algorithm has two undesirable features: the embedding of a branch-and-bound algorithm within a branch-and-bound algorithm, and the use of a partitioning scheme that is less efficient than Garfinkel's subtour elimination procedure.

This branch-and-bound procedure can be improved in two ways. First, rather than solving CAP exactly, an upper bound can be computed using the method discussed in Section 7.3. Second, using the two assignments, $x_1^*(\lambda^*)$ and $x_2^*(\lambda^*)$, produced as byproducts of this upper bounding method, to identify illegal subtours, Garfinkel's subtour elimination procedure can be used to partitioning the subproblems. This will eliminate $x_1^*(\lambda^*)$ and/or $x_2^*(\lambda^*)$ from the feasible set of assignments, resulting in tighter upper bounds at subsequent iterations. If possible, a subtour that appears in both assignments should be chosen for elimination. Branch-and-bound need be applied to CAP only in the case where both $x_1^*(\lambda^*)$ and $x_2^*(\lambda^*)$ contain no illegal subtours. In this case, $x_2^*(\lambda^*)$ is a feasible solution to CCTSP, but there may exist other feasible solutions with a value between that of $x_2^*(\lambda^*)$ and the upper bound.

8.4 Computational Results

The computation time required by Laporte and Martello's branch-and-bound algorithm and by our modified version utilizing the improved knapsack bound were compared using uniform problems in which the nodes all had equal values. A "timeout" was set for each algorithm. Laporte and Martello's algorithm was aborted if the computation time exceeded 4000 seconds and the improved algorithm was aborted if the

computation time exceeded 200 seconds. The computational results are given in Table 8.1. A sample size of 10 was used and results were averaged over the problems for which both algorithms completed within the allotted times. Computation times for the improved method are generally at least 10 times faster and sometimes hundreds of time faster than Laporte and Martello's method. We note that Euclidean problems required less

Class	Nodes	Budget	CPU time (seconds) Imp.	CPU time (seconds) L&M	Ratio	Sample size
	20	0.10	0.00	0.01		10
	20	0.20	0.01	0.18	**11.89**	10
	20	0.30	0.11	2.35	**22.00**	10
	20	0.40	0.65	20.23	**31.28**	10
Euclidean	20	0.50	4.26	139.29	**32.71**	10
	20	0.60	47.45	1231.02	**25.94**	10
	20	0.70	49.30	1067.60	**21.66**	5
	20	0.80	26.00	1021.74	**39.29**	3
	20	0.90	4.86	1338.61	**275.30**	4
	25	0.10	0.00	0.01	**3.00**	10
	25	0.20	0.04	0.77	**17.15**	10
Euclidean	25	0.30	0.74	30.65	**41.24**	10
	25	0.40	7.78	497.34	**63.92**	9
	25	0.50	40.27	2123.24	**52.73**	3
	20	0.10	0.01	0.06	**4.88**	10
	20	0.20	0.30	5.93	**19.65**	10
	20	0.30	12.62	212.40	**16.84**	10
Non-	20	0.40	39.49	491.26	**12.44**	10
Euclidean	20	0.50	66.82	763.69	**11.43**	10
	20	0.60	180.69	2016.82	**11.16**	10
	20	0.70	41.57	1198.24	**28.83**	5
	20	0.80				0
	20	0.90	14.68	1135.74	**77.35**	4
	25	0.10	0.02	0.23	**10.62**	10
Non-	25	0.20	1.47	28.53	**19.38**	10
Euclidean	25	0.30	58.31	1039.14	**17.82**	9
	25	0.40	72.71	1831.64	**25.19**	2
	25	0.50				0

Table 8.1. Computational results comparing Laporte and Martello's branch-and-bound algorithm and the improved version. Problems were generated using a uniform distribution and all nodes had equal value. Results are averaged over the problems that successfully completed within the allotted computation times.

computation time than the non-Euclidean problems. This is believed to be an artifact of the way the problems were generated and attributable to the amount of variability in the cost matrix rather than to the class of problem. Because of the way the problems were generated, the non-Euclidean problems showed less variability in their cost matrices than the Euclidean problems.

The two branch-and-bound approaches were compared by repeating Kataoka and Morito's computational experiments using our improved version of Laporte and Martello's branch-and-bound algorithm. Our improved algorithm was applied to problems generated from the same distributions as those used by Kataoka and Morito. Computation times for our algorithm were then adjusted to account for the difference in computers. The SUN 4/330 computation times were multiplied by a factor of 10.20, which was the factor of difference observed in experiments we conducted comparing the speed of the SUN 4/330 with a machine comparable to the one used by Kataoka and Morito. The adjusted computation times were then compared with Kataoka and Morito's published computation times. Results are given in Table 8.2. Unless otherwise stated, the problem parameters are as follows:

$c_{ij} \sim \text{unif}(30,70)$
$v_i \sim \text{unif}(5,15)$
$B = 250$
$n = 10.$

Computation times are averaged over a sample of 50 problems. The results show Kataoka and Morito's approach to be much inferior. Although the improvements discussed in the previous section would improve the performance of this approach, it seems unlikely that the improvement would be great enough to make this approach competitive. Hence, no further experimentation was done.

Problem parameters	CPU time (seconds) K&M	Imp.	Ratio
$B = 60$	17.10	0.00	**>1710.00**
$B = 80$	34.24	0.00	**>3424.00**
$B = 100$	48.86	0.02	**2443.00**
$B = 150$	77.42	0.05	**1548.40**
$B = 200$	89.08	0.25	**285.60**
$B = 250$	71.40	0.96	**74.38**
$B = 300$	46.52	2.27	**20.49**
$B = 350$	16.78	3.17	**5.30**
$B = 400$	0.54	0.68	**0.79**
$c = 50$	7.54	0.02	**377.00**
$40 \leq c \leq 60$	102.80	0.89	**115.51**
$30 \leq c \leq 70$	71.40	0.88	**81.14**
$20 \leq c \leq 80$	35.94	1.36	**26.43**
$10 \leq c \leq 90$	3.74	3.05	**1.23**
$v = 10$	10.28	0.21	**48.95**
$9 \leq v \leq 11$	61.28	0.70	**87.51**
$5 \leq v \leq 15$	71.40	0.70	**102.00**
$1 \leq v \leq 19$	54.18	0.73	**74.22**
$1 \leq v \leq 21$	55.82	0.67	**83.31**
$10 \leq v \leq 30$	72.74	0.68	**106.97**
$20 \leq v \leq 40$	85.36	0.72	**118.56**

Table 8.2. Comparison of published average computation times for Kataoka and Morito's branch-and-bound algorithm with average computation times for our improved version of Laporte and Martello's branch-and-bound algorithm. The computation times reported for the improved algorithm have been adjusted to account for the difference in computers.

Chapter 9
HEURISTIC ALGORITHMS

Having shown that CCTSP is NP-hard, we now "lower our sights" and consider *heuristic algorithms* — algorithms which find "good" (but not necessarily optimal) solutions within an acceptable amount of time. The most common technique used in heuristic algorithms is "neighborhood search," in which a predefined set of operations is used to iteratively improve an initial solution, until no further improvements can be obtained with these operations. The resulting solution is "locally optimal" with respect to the predefined operations. All of the heuristic algorithms that are discussed here use this technique. They begin with an initial subtour consisting of one or two nodes and iteratively improve it, maintaining feasibility throughout the process. Two types of improvements are possible — changing the set of nodes in a subtour, either by inserting an additional node or replacing a node with a more desirable one, and rearranging the order of the nodes in a subtour such that the cost of the subtour is reduced.

We have identified five key characteristics of heuristic algorithms for CCTSP. These are:

node selection – How is a node selected for insertion into a subtour?

insertion method – Where is the selected node inserted?

recourse – Once inserted, can a node later be deleted?

subtour improvement – Is an attempt made to reduce the cost of a subtour by rearranging its nodes?

repetition – Is a single subtour generated or are several generated and the best selected?

Repetition may be based on probabilistic events (e.g., randomization in the node selection process) or deterministic influences (e.g., starting the search with a different initial solution).

In this chapter, we discuss previous heuristic algorithms for CCTSP, along with some modifications, and then present a new one, which combines the strong points of the previous algorithms while attempting to avoid their shortcomings. In addition to comparing the solutions obtained by the algorithms, we also examine the effects of the individual features of the new algorithm on both solution quality and computation time. Several modifications of the new algorithm, with progressively decreasing computation time, are considered and the trade-off between solution quality and computation time examined.

9.1 Previous Heuristics

As noted in Chapter 4, four heuristic algorithms have been developed previously for CCTSP. The first two are based on simple TSP algorithms while the second two incorporate many new ideas. Since the second two algorithms are only applicable to Euclidean problems, we also present modifications which generalize them.

Golden, Levy and Dahl **[GLD]** developed a heuristic for a generalization of CCTSP that is based on TSP's cheapest insertion algorithm. In their generalization, rather than the nodes having values, the arcs have both costs and profits. The goal is to find a subtour that maximizes total profit while not exceeding a specified cost. Their algorithm varies from the cheapest insertion algorithm for TSP in that the node selection criterion takes into account profit as well as insertion costs. When we adapt their algorithm to CCTSP, the node selection criterion at iteration k becomes

$$v_j - R_k \Delta T_j,$$

where

P = value of current subtour,

T = cost of current subtour,

ΔT_j = cost of inserting node j at cheapest insertion point,

and $R_k = \alpha(P/T) + (1-\alpha)R_{k-1}$, where $0 \le \alpha \le 1$.

Once selected, a node is inserted into the subtour at its cheapest insertion point and all variables are updated. The process is continued until no further nodes can be inserted without violating the cost constraint. Several subtours can be generated by using different

values of R_0 and α. In our experiments, we used 0.1, 0.5, and 1.0 for α and 1, 10, 25, and P/T for R_0 where P/T is the value/cost ratio of the best solution generated thus far.

Tsiligirides [Tsi] modified the nearest neighbor algorithm for TSP to incorporate node values and randomization. In his algorithm, the node selection criterion is

$$\left(v_j / c_{last,j}\right)^4,$$

where *last* is the last node in the current subtour before returning to node 1. The node to be inserted is selected randomly from among the top four using probabilities proportional to these scores. The selected node is then inserted at the end of the subtour. The process is continued until no more nodes can be added without violating the cost constraint. This method of generating a subtour is repeated many times and the highest valued subtour selected. Tsiligirides suggests 3000 repetitions, a number which we found to be computationally prohibitive. In our experiments, the process is repeated 100 times. We found that, in general, exceeding 100 iterations was not productive.

Golden, Levy and Vohra [GLV] developed an algorithm for CCTSP using a new idea, "center of gravity." An initial subtour is generated using a node selection criteria based on a linear combination of value, distance from node 1, and distance from the center of gravity of, initially, all nodes, and later, the nodes in the current subtour. Nodes are inserted at their cheapest insertion point. When the cost constraint prohibits the insertion of additional nodes, a two-opt procedure is applied and more nodes are inserted if possible. The center of gravity, $cg = (\bar{x}, \bar{y})$, of this subtour (L_0) is then computed, where

$$\bar{x} = \sum_{i \in L_0} v_i x_i \Big/ \sum_{i \in L_0} v_i \quad \text{and} \quad \bar{y} = \sum_{i \in L_0} v_i y_i \Big/ \sum_{i \in L_0} v_i .$$

A new subtour is then generated using

$$v_j / \text{dist}(\text{node } j, cg)$$

as the node selection criterion, where $\text{dist}(i, j)$ refers to the distance between i and j, and then inserting the selected nodes at their cheapest insertion point. When no additional nodes can be inserted without violating the cost constraint, an attempt is made to reduce the cost of the subtour using a two-opt procedure. If possible, more nodes are then inserted.

The process is repeated, using each time the center of gravity of the previously generated subtour, until 10 subtours have been generated or until a center of gravity repeats itself. We note that this algorithm is only applicable to Euclidean problems where nodes are represented by x-y coordinates and costs are based on a distance function.

We generalize this center of gravity algorithm to make it applicable to non-Euclidean problems by using the node which best corresponds to the center of gravity of a set of nodes in place of an actual center of gravity. This node is computed as follows:

$$cg = \operatorname{argmin}_i \sum_{j \in S \setminus i} v_j c_{ij}^2.$$

In the node selection criterion, dist(node j, cg) is replaced by $c_{j,cg}$.

Golden, Wang and Liu's algorithm **[GWL]**, which they call the "Multi-Faceted Heuristic," incorporates many new ideas. Rather than considering a node's individual value, a "neighborhood value" is considered for each node. The neighborhood value for a specific node is an aggregate of its own value and the discounted values of all other nodes, where the discount factor depends on the distance of the node from the specified node. The aggregate value for a node j is

$$v_j + \sum_{i \neq j} v_i e^{-\mu c_{ij}},$$

for some discount factor $\mu \geq 0$. The desired value of μ depends on the scale of the problem. In our experiments, we used

$$\mu = 10/\max c_{ij}.$$

The node selection criterion uses a linear combination of aggregate value, distance from center of gravity, and distance from node 1. Each of these components is first scaled such that the maximum value over all of the nodes is n. The aggregate values are then multiplied by a learning measure (whose original value is 1). Weights of 0.7, 0.2 and 0.1, respectively, are used in the linear combination. Using this selection criterion, a node is selected randomly from the top five remaining nodes using equal probabilities. The selected node is inserted into the current subtour at its cheapest insertion point. If this results in a violation of the cost constraint, a node is then deleted from the subtour. To

select this node, the cost reduction (savings) resulting from deletion is computed for each node in the subtour. From those nodes for which the savings are not less than the current cost overrun, the one with the highest savings to value ratio is deleted. This node may be the one that was just inserted. Once deleted, a node is not reconsidered. The process is continued until no nodes remain for consideration. At this point, a two-opt procedure is applied to reduce the cost of the subtour. If the cost is reduced, the insertion/deletion process is repeated. This subtour generation process is repeated 20 times, each time replacing the center of gravity used with the center of gravity of the tour just generated. After each iteration, the learning measures are updated as follows:

$$LM_i = \frac{1}{|R_i|} \sum_{\ell \in R_i} (\text{value of subtour } \ell / \text{average subtour value}),$$

where

R_i = set of subtours generated thus far that include node i.

In addition, the entire process is repeated five times starting with different initial centers of gravity. To compute these initial centers of gravity, the smallest rectangle, with sides parallel to the x and y axes, that encloses all of the nodes is drawn. The five initial centers of gravity are the center of this rectangle and the centers of each of the four quadrants of the rectangle. As with the center of gravity algorithm, this algorithm is applicable only to Euclidean problems.

We generalize the Multi-Faceted Heuristic to make it applicable to non-Euclidean problems by using the node which best corresponds to the center of gravity of a set of nodes in place of an actual center of gravity, as we did for the center of gravity algorithm. The five initial centers of gravity are replaced by the following four points:

$$\left.\begin{aligned} p_1 &= k \\ p_2 &= l \end{aligned}\right\} \text{ where } c_{kl} = \max c_{ij},$$

$$p_3 = \underset{i}{\text{argmax}}\left\{\min\left(c_{p_1 i}, c_{p_2 i}\right)\right\},$$

$$\text{and } p_4 = \underset{i}{\text{argmin}}\left\{c^2_{p_1 i} + c^2_{p_2 i} + c^2_{p_3 i}\right\}.$$

9.2 A New Heuristic Algorithm

Before presenting our new heuristic algorithm, we will begin by analyzing some of the characteristics of the previous methods. The methods by Tsiligirides and by Golden, Levy, and Dahl are myopic in their node selection processes. Tsiligirides' algorithm is also myopic in its insertion method. The randomization process used in Tsiligirides' algorithm is biased toward always making the same selection since the probabilities are proportional to the fourth power of the nodes' value/cost ratio. On the other hand, since the node selection criterion is dependent on the last node added, when a different selection is made, the entire future of the algorithm may be changed. In contrast, the Multi-Faceted Heuristic (MFH) uses equal probabilities in the node selection process. However, since the node selection criteria are independent of the progress of the algorithm, the randomization only has the effect of causing local shuffling in a pre-ordered list. Both the center of gravity algorithm (CofG) and the MFH are somewhat less myopic since they consider distance from a center of gravity in the node selection process. This, in some sense, causes them to consider the relationship of the node to the overall tour. On the other hand, the algorithms are deficient in that they do not consider the actual cost of inserting a node. Furthermore, the emphasis on the center of gravity of the previously generated solution causes the algorithm to focus on similar solutions. The procedure of starting with five different initial centers of gravity, in MFH, counteracts this effect to some extent.

A notable feature of MFH is the use of "neighborhood scores," rather than individual node scores, in the node selection process. When evaluating a node, the number and value of nodes nearby are taken into consideration by using an aggregate node value. We note that, since all nodes are eventually included in a TSP solution, this is not an important consideration for TSP algorithms. On the other hand, it can be of extreme importance in solving CCTSP, particularly in problems where nodes occur in clusters. However, the aggregate values used by MFH are not updated as the algorithm progresses. Thus, a node may receive undue favoritism for bringing the path close to nodes that have already been included in the subtour or that have already been excluded from further consideration. As the remaining budget gets smaller, the amount of aggregation of the

node values should decrease since it does no good to bring the path close to other nodes if there is not enough budget remaining to include these nodes. Another favorable feature of MFH is that it allows a limited amount of recourse. Once inserted, a node may later be deleted in order to allow insertion of a more desirable node.

We developed an algorithm based on these observations about the previous methods. In our algorithm, aggregate node scores are used in the node selection process, but are updated as the algorithm proceeds. The aggregate node value for node i is

$$agg_i = v_i + \sum_{\substack{j \in S \\ j \neq i}} v_j e^{-\mu c_{ij}} \quad \text{where } S = \{\text{remaining candidates for insertion}\}.$$

The node selection criterion is

$$\frac{LM_i \times agg_i}{insertcost_i}$$

where $insertcost_i$ is the cost of inserting node i at the cheapest insertion point in the current subtour and LM_i is the same as in MFH. Nodes are selected randomly from the top five using equal probabilities. Note that the values given by the node selection criterion change after each insertion. Thus, the randomization has a greater effect than in MFH.

Initially, our algorithm uses the same insertion process as MFH. The selected node is inserted in the current subtour at its cheapest insertion point. If this causes the cost to exceed the budget, the node with the highest savings/value ratio, subject to the condition that the savings are at least as great as the current cost overrun, is deleted from the subtour. Once deleted, a node is not reconsidered. When no nodes remain for consideration, a two-opt procedure is applied to reduce the cost of the subtour. At this point, the procedure differs from MFH. Regardless of whether the two-opt procedure resulted in cost reduction, an attempt is then made to insert additional nodes. This time deletions are *not* allowed. Nodes are selected deterministically using the ratio of value to insertion cost as the node selection criterion. Nodes are considered only if their insertion does not violate the cost constraint. When no additional nodes can be inserted, the two-opt procedure is repeated. An attempt is then made to increase the value of the subtour by swapping nodes. Nodes are selected for insertion according to their values and inserted at their cheapest

insertion points. If the cost constraint is violated after an insertion, the lowest-value node, subject to the requirement that its deletion reduce the cost to within the cost constraint, is deleted from the subtour. This process is continued until no further improvements can be made. Figure 9.1 shows a flowchart of this entire procedure.

Our algorithm begins with a tour consisting of node 1 and a specified "focus point." Repeating the algorithm using different focus points can have a drastic effect on the initial insertion costs and, thus, drastically change the course of the algorithm. We repeat the algorithm with five different focus points for Euclidean problems and four for non-Euclidean problems. The focus points used in non-Euclidean problems are the same as the initial centers of gravity used by MFH. In Euclidean problems, the focus points used are the points nearest the five initial centers of gravities used by MFH. For a given focus point, 10 iterations of the algorithm are executed.

9.3 Computational Results

Computational experiments were conducted to compare the five heuristic algorithms. Both the percent error from optimality and the computation time were recorded. Initially, problems with 20 nodes were used and the results were averaged over a sample size of 10. Selected representative results are given in Table 9.1. Complete results are given in Appendix C. Out of 540 test problems, MFH found the optimal solution 510 times, while our new heuristic (NewH) found the optimal solution 524 times. Tsiligirides' algorithm (Tsi), CofG, and Golden, Levy, and Dahl's algorithm (GLD) were clearly inferior in solution quality to MFH and NewH. However, if speed of computation is essential, CofG might be preferred. Since these experiments did not show a statistically significant difference between MFH and NewH, further experiments were done with these two algorithms using 50-node problems, a sample size of 40 for Euclidean problems, and a sample size of 20 for non-Euclidean problems. Selected results are shown in Table 9.2 and complete results are included in Appendix C. Overall, NewH appears to outperform MFH.

Next, the effect of individual features of NewH on computation speed and solution quality were examined. Five versions of NewH were created by dropping the following individual features:

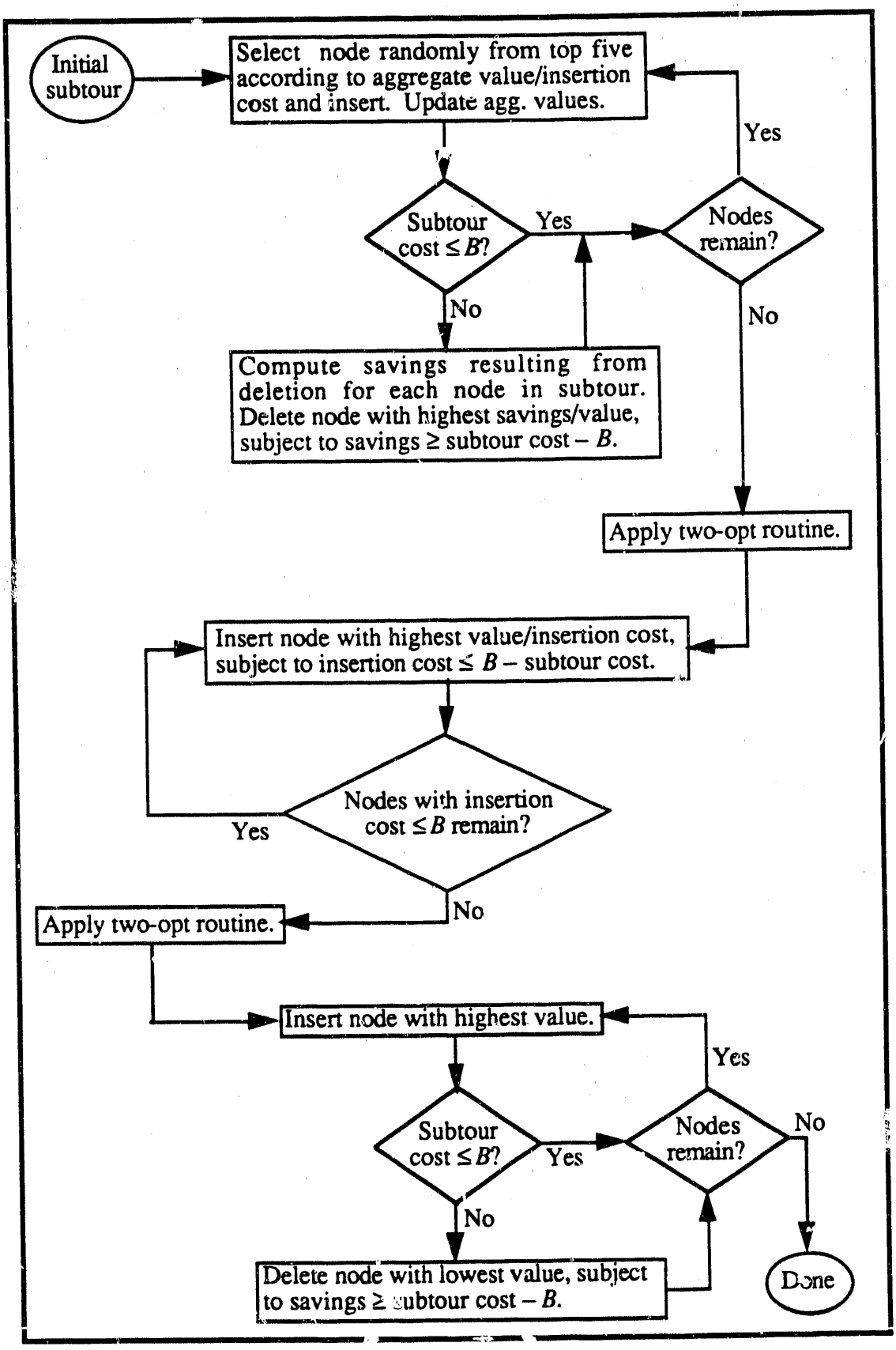

Figure 9.1. Insertion procedure for NewII.

Budget	NewH	MFH	Tsi	CofG	GLD
	0.61	0.41	0.20	0.02	0.01
0.25	0.00	0.77	0.27	**4.95**	**9.40**
	0.85	0,62	0.74	0.04	0.02
0.50	0.00	0.14	**4.82**	**4.87**	**11.84**
	1.30	1.24	1.24	0.07	0.03
0.75	0.00	0.10	**4.41**	**6.13**	**13.53**

Table 9.1. Selected computational results comparing heuristic algorithms. The results shown are for Euclidean, uniform problems with 20 nodes, averaged over a sample size of 10. Node values are uniformly distributed between 1 and 10. Numbers in large type are the average percent error from optimality and numbers in small type are average computation times. Bold type indicates that the difference in performance from that of NewH was statistically significant at the 95% level.

	clusters, equal v's sample size = 40		ave. over all Euclid. types sample size = 360	
Budget	NewH	MFH	NewH	MFH
	3.66	2.29		
0.25	0.00	**3.26**	0.27	**1.40**
	7.86	7.56		
0.50	0.08	**1.24**	0.86	0.97
	17.28	20.45		
0.75	0.30	0.66	0.63	**0.90**

Table 9.2. Selected computational results comparing NewH and MFH. The results shown are for Euclidean problems with 50 nodes. Numbers in large type are the average percent error from the best known solution and numbers in small type are average computation times. Bold type indicates that, when compared with NewH, the difference in performance was statistically significant at the 95% level.

aggregate values (agg.) – aggregate values were replaced by individual node values in the node selection criterion,

learning measure (LM) – the learning measure was dropped in the node selection criterion,

updating of aggregate values (updat.) – aggregate values were computed as in MFH and not updated as the algorithm progressed,

focus points (foc. pts.) – the insertion process was started with a subtour consisting of node 1 only, and

randomization (rand.) – the highest scoring node, according to the selection criterion, was always selected.

The experiments were conducted using problems with 50 nodes, a sample size of 40 for Euclidean problems, and a sample size of 20 for non-Euclidean problems. Selected results are given in Table 9.3 and complete results in Appendix C. Further experiments were conducted for the learning measure and aggregate values using problems with 100 nodes and a sample size of 20. These results are given in Table 9.4. The updating of aggregate

Node values	*B*	NewH	no agg.	no LM	no updat.	no foc.pts.	no rand.
		3.66	2.75	3.58	2.76	0.77	0.34
equal	0.25	0.16	0.13	0.35	0.13	**1.99**	**2.68**
		7.75	6.63	7.67	6.80	1.62	0.75
equal	0.50	0.49	**1.10**	0.49	**1.00**	**2.50**	**2.99**
		16.62	15.28	16.49	15.52	3.37	1.63
equal	0.75	0.88	1.12	0.65	**1.17**	**2.65**	**2.87**
		3.57	2.64	3.51	2.68	0.75	0.33
exp.	0.25	0.44	**0.04**	0.37	0.78	**1.60**	**3.29**
		7.32	6.19	6.31	6.50	1.56	0.71
exp.	0.50	1.47	**1.01**	0.72	**2.81**	**4.74**	**5.44**
		15.61	14.41	15.49	15.13	3.17	1.59
exp.	0.75	0.84	**0.60**	1.05	**1.66**	**2.97**	**2.73**

Table 9.3. Selected computational results showing the effects of dropping individual features of NewH. The results shown are for Euclidean problems with 50 nodes and a uniform distribution. Results are averaged over a sample size of 40. Numbers in large type are the average percent error from the best known solution and numbers in small type are average computation times. Bold type indicates that, when compared with NewH, the difference in performance was statistically significant at the 95% level.

Node values	Budget	NewH	no agg.	no LM
equal	0.25	16.35 0.96	12.91 **2.13**	15.81 0.32
equal	0.50	49.66 0.00	43.08 **2.66**	49.06 1.16
equal	0.75	128.77 0.94	116.20 **2.28**	128.27 1.40
Unif(1,10)	0.25	16.17 1.97	12.52 1.97	15.58 0.73
Unif(1,10)	0.50	47.36 1.68	40.87 **2.83**	46.91 2.41
Unif(1,10)	0.75	122.53 0.75	111.00 **1.39**	122.32 0.61

Table 9.4. Computational results showing the effects of dropping the aggregate value and learning measure features of NewH. The results shown are for Euclidean problems with 100 nodes and a uniform distribution. Results are averaged over a sample size of 20. Numbers in large type are the average percent error from the best known solution and numbers in small type are average computation times. Bold type indicates that, when compared with NewH, the difference in performance was statistically significant at the 95% level.

values, use of focus points, and randomization in the node selection process show a statistically significant beneficial effect on solution quality. The use of focus points and randomization cause a substantial increase in computation time. We note that the amount of this increase is determined by the number of focus points used or the number of iterations conducted. The aggregation of node values had a beneficial effect on solution quality, except in cases where the node values were distributed exponentially. In those cases, the effect was negative. Use of the learning measure does not appear to have a significant effect.

Finally, we experimented with several versions of NewH, requiring progressively decreasing computation time, to examine the trade-off between computation time and solution quality. In the first two variations, we dropped the use of focus points and randomization, respectively. Next, we dropped both focus points and randomization. The

final variation used a straight cheapest insertion algorithm without recourse, where the node selection criterion was simply the node value to insertion cost ratio, followed by two-opt and node swapping for improvement. Selected results are shown in Table 9.5 andcomplete results in Appendix C. The problems were also solved with CofG, since as mentioned earlier, CofG might be the favored algorithm if speed of computation is essential. Our results show that the faster variations of NewH are preferable to CofG, except for cases where the nodes values are exponentially distributed. We note that these are the same cases where using aggregate node values is detrimental and that all the variations of NewH use aggregate node values, except for cheapest insertion.

Node values	*B*	NewH	no foc.pts.	no rand.	no foc. no rand.	chp. insert.	CofG
		3.54	0.74	0.33	0.09	0.03	0.14
equal	0.25	0.13	**2.11**	3.18	**7.07**	7.63	9.90
		7.65	1.56	0.74	0.17	0.10	0.42
equal	0.50	0.31	**2.49**	3.65	**8.10**	**14.15**	11.54
		16.62	15.28	16.49	15.52	3.37	1.63
equal	0.75	0.82	**2.80**	2.80	**7.14**	**10.99**	8.72
		3.46	0.72	0.32	0.09	0.02	0.12
exp.	0.25	0.00	**0.62**	**3.11**	**8.79**	7.17	5.07
		7.23	1.52	0.70	0.16	0.10	0.30
exp.	0.50	0.90	**4.76**	**5.66**	**10.90**	9.64	7.86
		15.70	3.08	1.59	0.37	0.28	0.70
exp.	0.75	0.52	**2.60**	2.21	**5.61**	5.50	4.08

Table 9.5. Selected computational results for experiments examining the trade-off between solution quality and computation time. The results shown are for Euclidean problems with 50 nodes and a uniform distribution. Results are averaged over a sample size of 20. Numbers in large type are the average percent error from the best known solution and numbers in small type are average computation times. Bold type indicates that, when compared with the algorithm to the immediate left in the table, the difference in performance was statistically significant at the 95% level.

Chapter 10
CONCLUSIONS

Since relatively little previous research had been done on the Cost-Constrained Traveling Salesman problem, we undertook a comprehensive study, touching on many areas rather than focusing on one specific aspect. In this chapter, we give a brief summary of our results. This is followed by a discussion of open questions and areas for future research.

10.1 Summary of Results

The Cost-Constrained Traveling Salesman Problem is a difficult combinatorial optimization problem with many practical applications. CCTSP is NP-hard, and no K-approximation algorithm or fully polynomial approximation scheme exists, unless P = NP. Although, in theory, CCTSP is equivalent to the Traveling Salesman Problem, in practice it appears to be more difficult. CCTSP requires both selection and sequencing, unlike TSP, which requires sequencing only. As a consequence, most results for TSP cannot be extended to CCTSP. We were, however, able to show that several special cases, which are solvable for TSP using low order polynomial algorithms, are also solvable for CCTSP using polynomial algorithms of degree 3 or less. These are the cases of outer-sum matrices, small matrices, circulant matrices, and upper triangular matrices.

Algorithms for CCTSP, which outperform previous methods, were developed in three areas: upper bounding methods, exact algorithms, and heuristics. Extensive computational studies were undertaken to evaluate and compare algorithms. These computational studies also examined the sensitivity of performance to problem characteristics.

We found that a bounding strategy based on the knapsack problem performs better, both in speed and in the quality of the bounds, than methods based on the assignment

problem. We note that the preferred method depends primarily on the selection aspect of CCTSP and, hence, is not related to upper bounding methods for TSP.

Likewise, we found that a branch-and-bound approach using the knapsack bound and a very simple branching strategy was superior to a method, analogous to a common branch-and-bound method for TSP, that uses a constrained assignment problem for bounding and subtour elimination for branching. In addition, the preferred branch-and-bound method is easy to implement and can be applied to several extensions of CCTSP as well as the basic problem.

In our study of heuristic algorithms for CCTSP, we made several observations. First, when selecting nodes for the subtour, it is important to consider the "neighborhood" of the nodes. A node with low value that brings the subtour near many other nodes may be more desirable than an isolated node of high value. Second, an algorithm that generates many different solutions and selects the best one results in better solutions than one that generates a single solution. However, such an algorithm also requires more computation time. We found two types of repetition to be desirable: repetitions based on randomization in the subtour building process, and repetitions focusing the subtour toward different nodes or areas. We developed a heuristic algorithm that incorporated these features. Computational experiments show that this method outperforms previous methods in solution quality. By varying the number and type of repetitions done by our method, we can adjust the computation time required and obtain algorithms that outperform previous methods in both speed and solution quality.

10.2 Open Questions

One outstanding question about CCTSP relates to its complexity. For the general case of TSP, it has been shown that there cannot exist a polynomial algorithm A with a performance guarantee of the form

$$length_A \leq r \times length_{opt},$$

unless P = NP. However, several polynomial algorithms with this type of performance guarantee have been developed for the case where the triangle inequality holds. No such

results have been obtained for CCTSP. We conjecture that the first result holds for CCTSP as well. That is, we conjecture that there cannot exist a polynomial algorithm A for CCTSP with a performance guarantee of the form

$$V_A \leq r \times V_{opt},$$

unless P = NP. However, since CCTSP appears to be more difficult than TSP, we will not speculate that a polynomial algorithm with a performance guarantee can be obtained for the case of CCTSP where the triangle inequality holds, even when the nodes have equal value.

A great deal of research has been done to characterize the facets of the underlying polytope of feasible solutions for TSP. Another open question about CCTSP regards the relationship between its polytope of feasible solutions and that of TSP. Using results regarding the facial structure of TSP polytopes, exact algorithms which can solve very large problems have been obtained. Similar results for CCTSP could prove to be very useful.

10.3 Areas for Future Research

Many areas for future research remain. As mentioned above, results concerning the facial structure of the CCTSP polytope might be very useful. With such results, a branch-and-cut algorithm similar to that of Crowder and Padberg for TSP [CP] could be developed. The success of the branch-and-cut method for TSP has been overwhelming. The variant tackled by Padberg and Rinaldi [PR] with a similar method is much more complex than CCTSP. While the results of Padberg and Rinaldi generate some doubts about the efficiency of such a method for CCTSP, it would be premature to draw any conclusions.

Further refinement of heuristic algorithms for CCTSP could prove to be fruitful. For example, when considering "neighborhood scores" for nodes, the discount function used by our method may not be the best. Different parameters in the discount function, or a different discount function altogether, may improve performance. Also, rather than considering the distance, or cost, between nodes when aggregating node values, it may

prove more beneficial to consider how the inclusion of a node affects the cost of including other nodes. This would require comparing the insertion costs of other nodes, given that the specified node has been inserted into the tour, with the insertion costs prior to the inclusion of the specified node. We did not take this into account in our method because it increases the computation time by a factor of n. However, if it improves performance substantially, it may be worth the extra computation time. We note that, while our algorithm found optimal or very near optimal solutions for problems with 20 nodes, since we did not obtain exact solutions for larger problems, we can say very little about its performance, relative to optimality, on larger problems. We know only that it outperforms previous heuristic methods.

Finally, another area for future work is the development of algorithms for extensions or variations of CCTSP. Two extensions which incorporate time dependencies were discussed in Chapter 3. Other interesting extensions and variations surely exist.

REFERENCES

[Ba] Baker, E., "An Exact Algorithm for the Time-Constrained Traveling Salesman Problem," *Operations Research*, Vol. 31, 1983, pp. 938-945.

[Be] Berenguer, X., "A Characterization of Linear Admissible Transformations for the *m*-Travelling Salesmen Problem," *European Journal of Operational Research*, Vol. 3, 1979, pp. 232-249.

[BJ] Bhattacharyya, G. and R. Johnson, *Statistical Concepts and Methods*, John Wiley & Sons, 1977, pp. 516-523.

[Ch] Christofides, N., *Worst-Case Analysis of a New Heuristic for the Traveling Salesman Problem*, Report 388, Graduate School of Industrial Administration, Carnegie-Mellon University, 1976.

[CP] Crowder, H. and M. Padberg, "Solving Large-Scale Symmetric Traveling Salesman Problems to Optimality," *Management Science*, Vol. 26, 1980, pp. 495-509.

[Da] Dakin, R., "A Tree Search Algorithm for Mixed Integer Programming Problems," *Computer Journal*, Vol. 8, 1965, pp. 250-255.

[Dan] Dantzig, G., "Discrete-Variable Extremum Problems," *Operations Research*, Vol. 5, 1957, pp. 266-277.

[DBR] Dantzig, G., W. Blattner, and M. Rao, "Finding a Cycle in a Graph with Minimum Cost to Times Ratio with Application to a Ship Routing Problem," *Theory of Graphs*, edited by P. Rosenstiehl, Gordon and Breach, 1967, pp. 77-84.

[Ev] Everett, H., "Generalized Lagrange Multiplier Method for Solving Optimization Problems," *Operations Research*, Vol. 11, 1963, pp. 399-417.

[Fi] Fisher, M., "The Lagrangian Relaxation Method for Solving Integer Programming Problems," *Management Science*, Vol. 27, 1981, pp. 1-18.

[FT] Fischetti, M. and P. Toth, "An Additive Approach for the Optimal Solution of the Prize-Collecting Travelling Salesman Problem," *Vehicle Routing: Methods and Studies*, edited by B. Golden and A. Assad, Elsevier Science Publishers, 1988, pp. 319-343.

[Ga73] Garfinkel, R., "On Partitioning the Feasible Set in a Branch-and-Bound Algorithm for the Asymmetric Traveling-Salesman Problem," *Operations Research*, Vol. 21, 1973, pp. 340-343.

[Ga77] Garfinkel, R., "Minimizing Wallpaper Waste, Part I: A Class of Traveling Salesman Problems," *Operations Research*, Vol. 25, 1977, pp. 741-751.

[Ge] Gensch, D., "An Industrial Application of the Traveling Salesman's Subtour Problem," *AIIE Transactions*, Vol. 10, 1978, pp. 362-370.

[GLD] Golden, B., L. Levy, and R. Dahl, "Two Generalizations of the Traveling Salesman Problem," *OMEGA*, Vol. 9, 1981, pp. 439-441.

[GLS] Gilmore, P., E. Lawler, and D. Shmoys, "Well-Solved Special Cases," *The Traveling Salesman Problem*, edited by E. Lawler, J. Lenstra, A Rinnooy Kan, and D. Shmoys, John Wiley & Sons Ltd., 1985, pp. 87-120.

[GLV] Golden, B., L. Levy, and R. Vohra, "The Orienteering Problem," *Naval Research Logistics*, Vol. 34, 1987, pp. 307-318.

[GWL] Golden, B., Q. Wang, and L. Liu, *A Multi-Faceted Heuristic for the Orienteering Problem*, Working Paper Series MS/S 87-006, College of Business and Management, University of Maryland, 1987.

[HK] Held, M. and R. Karp, "A Dynamic Programming Approach to Sequencing Problems," *SIAM Journal of Applied Mathematics*, Vol. 10, pp. 196-210.

[HW] Hoffman, A. J. and P. Wolfe, "History," *The Traveling Salesman Problem*, edited by E. Lawler, J. Lenstra, A Rinnooy Kan, and D. Shmoys, John Wiley & Sons Ltd., 1985, pp. 1-9.

[KM] Kataoka, S. and S. Morito, "An Algorithm for Single Constraint Maximum Collection Problem," *Journal of the Operations Research Society of Japan*, Vol. 31, 1988, pp. 515-530.

[Li] Little, J., K. Murty, D. Sweeney, and C. Karel, "An Algorithm for the Traveling Salesman Problem," *Operations Research*, Vol. 11, 1963, pp. 972-989.

[Lin] Lin, S., "Computer Solutions of the Traveling Salesman Problem," *Bell System Technical Journal*, Vol. 44, 1965, pp. 2245-2269.

[LM] Laporte, G. and S. Martello, *The Selective Travelling Salesman Problem*, GERAD Report G-87-15, Ecole des Hautes Etudes Commerciales, 1988.

[MT] Martello, S. and P. Toth, "An Upper Bound for the Zero-One Knapsack Problem and a Branch and Bound Algorithm," *European Journal of Operations Research*, Vol. 1, 1977, pp. 169-175.

[Mu] Murty, K., *Linear Programming*, John Wiley & Sons Ltd., 1983, pp. 197-198.

[PR] Padberg, M. and G. Rinaldi, "A Branch-and-Cut Approach to a Traveling Salesman Problem with Side Constraints," *Management Science*, Vol. 35, 1989, pp. 1393-1412.

[PS] Papadimitriou, C. and K Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Inc., 1982.

[RSL] Rosenkrantz, D., R. Stearns, and P. Lewis II, "An Analysis of Several Heuristics for the Traveling Salesman Problem," *SIAM Journal of Computing*, Vol. 6, 1977, pp. 563-581.

[SG] Sahni, S. and T. Gonzalez, "*P*-Complete Approximation Problems," *Journal of the Association of Computing Machinery*, Vol. 23, 1976, pp.555-565.

[Tsi] Tsiligirides, T., "Heuristic Methods Applied to Orienteering," *Journal of the Operational Research Society*, Vol. 35, 1984, pp.797-809.

[Vo] Voigt, B., *Der Handlungsreisende, wie er sein soll und was er zu thun hat, um Auftrage zu erhalten und eines glucklicken Erfolgs in seinen Geschaften gewiss zu wein, Von einem alten Commis-Voyageur*, Ilmenau, 1831.

BIBLIOGRAPHY

Combinatorial Optimization

Garey, M. and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Co., 1979.

Lawler, E., *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, 1976.

Papadimitriou, C. and K Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Inc., 1982.

Integer Programming

Dakin, R., "A Tree Search Algorithm for Mixed Integer Programming Problems," *Computer Journal*, Vol. 8, 1965, pp. 250-255.

Fisher, M., "The Lagrangian Relaxation Method for Solving Integer Programming Problems," *Management Science*, Vol. 27, 1981, pp. 1-18.

Geoffrion, A. and R. Marsten, "Integer Programming: A Framework and State-of-the-Art Survey," *Management Science*, Vol. 18, 1972, pp. 465-491.

Traveling Salesman Problem

Bellman, R., "Dynamic Programming Treatment of the Travelling Salesman Problem," *Journal of the Association for Computing Machinery*, Vol. 9, 1962, pp. 61-63.

Bellmore, M. and G. Nemhauser, "The Traveling Salesman Problem: A Survey," *Operations Research*, Vol. 16, 1968, pp. 538-558.

Berenguer, X., "A Characterization of Linear Admissible Transformations for the *m*-Travelling Salesmen Problem," *European Journal of Operational Research*, Vol. 3, 1979, pp. 232-249.

Christofides, N., *Worst-Case Analysis of a New Heuristic for the Traveling Salesman Problem*, Report 388, Graduate School of Industrial Administration, Carnegie-Mellon University, 1976.

Crowder, H. and M. Padberg, "Solving Large-Scale Symmetric Traveling Salesman Problems to Optimality," *Management Science*, Vol. 26, 1980, pp. 495-509.

Dantzig, G., R. Fulkerson, and S. Johnson, "Solution of a Large-Scale Traveling-Salesman Problem," *Operations Research*, Vol. 2, 1954, pp. 303-410.

Dantzig, G., R. Fulkerson, and S. Johnson, "On a Linear-Programming, Combinatorial Approach to the Traveling-Salesman Problem," *Operations Research*, Vol. 7, 1959, pp. 58-66.

Flood, M., "The Traveling-Salesman Problem," *Operations Research*, Vol. 4, 1956, pp. 61-75.

Garfinkel, R., "Minimizing Wallpaper Waste, Part I: A Class of Traveling Salesman Problems," *Operations Research*, Vol. 25, 1977, pp. 741-751.

Garfinkel, R., "On Partitioning the Feasible Set in a Branch-and-Bound Algorithm for the Asymmetric Traveling-Salesman Problem," *Operations Research*, Vol. 21, 1973, pp. 340-343.

Held, M. and R. Karp, "A Dynamic Programming Approach to Sequencing Problems," *SIAM Journal of Applied Mathematics*, Vol. 10, pp. 196-210.

Lawler, E., J. Lenstra, A Rinnooy Kan, and D. Shmoys (editors), *The Traveling Salesman Problem*, John Wiley & Sons Ltd., 1985.

Lin, S., "Computer Solutions of the Traveling Salesman Problem," *Bell System Technical Journal*, Vol. 44, 1965, pp. 2245-2269.

Little, J., K. Murty, D. Sweeney, and C. Karei, "An Algorithm for the Traveling Salesman Problem," *Operations Research*, Vol. 11, 1963, pp. 972-989.

Rosenkrantz, D., R. Stearns, and P. Lewis II, "An Analysis of Several Heuristics for the Traveling Salesman Problem," *SIAM Journal of Computing*, Vol. 6, 1977, pp. 563-581.

Sahni, S. and T. Gonzalez, "*P*-Complete Approximation Problems," *Journal of the Association of Computing Machinery*, Vol. 23, 1976, pp. 555-565.

Cost Constrained Traveling Salesman Problem and Other Variants

Baker, E., "An Exact Algorithm for the Time-Constrained Traveling Salesman Problem," *Operations Research*, Vol. 31, 1983, pp. 938-945.

Dantzig, G., W. Blattner, and M. Rao, "Finding a Cycle in a Graph with Minimum Cost to Times Ratio with Application to a Ship Routing Problem," *Theory of Graphs*, P. Rosenstiehl, editor, Dunod, Paris, Gordon, and Breach, 1967, pp. 77-84.

Fischetti, M. and P. Toth, "An Additive Approach for the Optimal Solution of the Prize-Collecting Travelling Salesman Problem," *Vehicle Routing: Methods and Studies*, edited by B. Golden and A. Assad, Elsevier Science Publishers, 1988, pp. 319-343.

Gensch, D., "An Industrial Application of the Traveling Salesman's Subtour Problem," *AIIE Transactions*, Vol. 10, 1978, pp. 362-370.

Golden, B., L. Levy, and R. Dahl, "Two Generalizations of the Traveling Salesman Problem," *OMEGA*, Vol. 9, 1981, pp. 439-441.

Golden, B., L. Levy, and R. Vohra, "The Orienteering Problem," *Naval Research Logistics*, Vol. 34, 1987, pp. 307-318.

Golden, B., Q. Wang, and L. Liu, *A Multi-Faceted Heuristic for the Orienteering Problem*, Working Paper Series MS/S 87-006, College of Business and Management, University of Maryland, 1987.

Kataoka, S. and S. Morito, "An Algorithm for Single Constraint Maximum Collection Problem," *Journal of the Operations Research Society of Japan*, Vol. 31, 1988, pp. 515-530.

Laporte, G. and S. Martello, *The Selective Travelling Salesman Problem*, GERAD Report G-87-15, Ecole des Hautes Etudes Commerciales, 1988.

Padberg, M. and G. Rinaldi, "A Branch-and-Cut Approach to a Traveling Salesman Problem with Side Constraints," *Management Science*, Vol. 35, 1989, pp. 1393-1412.

Tsiligirides, T., "Heuristic Methods Applied to Orienteering," *Journal of the Operational Research Society*, Vol. 35, 1984, pp. 797-809.

Voigt, B. F, *Der Handlungsreisende, wie er sein soll und was er zu thun hat, um Auftrage zu erhalten und eines glucklicken Erfolgs in seinen Geschaften gewiss zu wein, Von einem alten Commis-Voyageur*, Ilmenau, 1831.

Miscellaneous

Bhattacharyya, G. and R. Johnson, *Statistical Concepts and Methods*, John Wiley & Sons, 1977, pp. 516-523.

Dantzig, G., "Discrete-Variable Extremum Problems," *Operations Research*, Vol. 5, 1957, pp. 266-277.

Everett, H., "Generalized Lagrange Multiplier Method for Solving Optimization Problems," *Operations Research*, Vol. 11, 1963, pp. 399-417.

Floyd, R., "Algorithm 97: Shortest Path," *Communications of the Association for Computing Machinery*, Vol. 5, 1962, p. 345

Martello, S. and P. Toth, "An Upper Bound for the Zero-One Knapsack Problem and a Branch and Bound Algorithm," *European Journal of Operations Research*, Vol.1, 1977, pp. 169-175.

Murty, K., *Linear Programming*, John Wiley & Sons Ltd., 1983.

APPENDIX A: Initial Values of λ_1 and λ_2 for Computing CABλ

We desire initial values λ_1 and λ_2 such that the optimal solutions to

$$\max \quad \sum_{i=1}^{n} v_i\left(1-x(\lambda)_{ii}\right)-\lambda\left(\sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x(\lambda)_{ij}-B\right)=vx(\lambda)-\lambda\left(cx(\lambda)-B\right) \qquad \text{CAP}(\lambda)$$

$$\text{subject to:} \quad \sum_{i=1}^{n} x(\lambda)_{ij}=1 \qquad \text{for } j=1,2,\ldots,n$$

$$\sum_{j=1}^{n} x(\lambda)_{ij}=1 \qquad \text{for } i=1,2,\ldots,n$$

$$x(\lambda)_{ij}\in\{0,1\} \qquad \text{for all } i,j$$

satisfy $cx^*(\lambda_1) > B$ and $cx^*(\lambda_2) < B$, where $c_{11} = \infty$ and $c_{ii} = 0$ for all $i \neq 1$.

First, we consider λ_1. We will find a λ_1 such that

$$vx^*(\lambda_1)=\sum_{i=1}^{n} v_i = V$$

Provided $\lambda_1 > 0$, $x^*(\lambda_1)$ will also satisfy

$$cx^*(\lambda_1)=\min\ cx(\lambda_1)$$

$$\text{subject to:} \quad vx(\lambda_1) = V$$

$$x(\lambda_1) \text{ feasible for CAP}(\lambda).$$

If $cx^*(\lambda_1) \leq B$, then $x^*(\lambda_1)$ solves CAP and we need not search for λ^*.

If $x^*(\lambda_1)_{ii} = 0$ for all i, then $vx^*(\lambda_1) = V$. Select λ_1 such that

$$0 < \lambda_1 < \min_{\substack{i\neq 1, j\neq 1\\ i\neq j}} \frac{v_i}{c_{1i}+c_{ij}-c_{1j}}.$$

To show that $x^*(\lambda_1)_{ii} = 0$ for all i, suppose $x^*(\lambda_1)_{ii} = 1$ for some i and $x^*(\lambda_1)_{1j} = 1$ (this holds for some j since $c_{11} = \infty$). The change in the objective function resulting from inserting node i between node 1 and node j is

$$v_i - \lambda_1\left(c_{1i} + c_{ij} - c_{1j}\right) > 0,$$

contradicting the optimality of $x^*(\lambda_1)$.

Now we consider λ_2. We will find a λ_2 such that $x^*(\lambda_2)$ minimizes $cx(\lambda)$. Let D be the matrix of shortest pathlengths between each pair of nodes. If C satisfies the triangle inequality, then $D = C$. The minimum possible value of $cx(\lambda)$ is

$$d_{\min} = d_{1\hat{i}} + d_{\hat{i}1}, \text{ where } \hat{i} = \underset{i}{\operatorname{argmin}}\left(d_{1i} + d_{i1}\right).$$

Let

$$d_{\text{next}} = \min_{d_{1j}+d_{j1} > d_{\min}} \left(d_{1j} + d_{j1}\right)$$

and let $x_{\min}$ be the solution formed by taking the shortest path form node 1 to node $\hat{i}$, followed by the shortest path from node $\hat{i}$ to node 1, and setting $x_{ii} = 1$ for all nodes i not in this subtour. (Note that no node may appear twice in this subtour since that would contradict the definition of $\hat{i}$. Thus, $x_{\min}$ is a valid solution to CAP(λ).)

Select λ_2 such that.

$$\lambda_2 > \frac{V}{d_{\text{next}} - d_{\min}} \text{ and } \lambda_2 > \frac{v_i}{c_{ij}} \text{ for all } i, j.$$

To show that $cx^*(\lambda_2) = d_{\min}$, first suppose $x^*(\lambda_2)$ contains a subtour that does not include node 1. The change in the objective function resulting form replacing each x_{ij} in this subtour with $x_{ii} = 1$ is

$$\sum_{\substack{i,j \\ x^*(\lambda_2)_{ij} \text{ in subtour}}} (\lambda_2 c_{ij} - v_i) > 0,$$

contradicting the optimality of $x^*(\lambda_2)$. Thus $x^*(\lambda_2)$ consists of a single subtour containing node 1 and some number of self loops ($x_{ii} = 1$). Suppose $cx^*(\lambda_2) > d_{\min}$. The change in objective function resulting from replacing $x^*(\lambda_2)$ with $x_{\min}$ is

$$\begin{aligned}
&\lambda_2\left(cx^*(\lambda_2)\right) - vx^*(\lambda_2) + vx_{\min} - \lambda_2 d_{\min} \geq \lambda_2 d_{\text{next}} - V + 0 - \lambda_2 d_{\min} \\
&= \lambda_2(d_{\text{next}} - d_{\min}) - V > 0,
\end{aligned}$$

contradicting the optimality of $x^*(\lambda_2)$. Thus, $cx^*(\lambda_2) = d_{\min}$. If $d_{\min} > B$, then CAP is infeasible. If $d_{\min} = B$, then $x^*(\lambda_2)$ solves CAP and we need not search for λ^*.

APPENDIX B: A Counterexample to Gensch's claim

Gensch [Ge] claims that the solution $x_2^*(\lambda^*)$ found when computing CABλ actually solves CAP. The following counterexample to this claim consists of four nodes in the x-y plane and uses the Euclidean distance function for arc costs.

Node	(x,y)	Value
1	(1,1)	5
2	(0,0)	3.5
3	(0,1)	3
4	(1,2)	1

$$C = \begin{bmatrix} \infty & 1.4 & 1 & 1 \\ 1.4 & 0 & 1 & 2 \\ 1 & 1 & 0 & 1.4 \\ 1 & 2.2 & 1.4 & 0 \end{bmatrix} \qquad B = 3$$

In figure B.1, we plot each feasible solution to CAP(λ) according to its value and cost. Those points below the line Cost = 3 are feasible solutions to CAP. We also indicate the optimal solution(s) to CAP(λ) for each value of λ and the optimal solution to CAP. In this counterexample, the value of $x_2^*(\lambda^*)$ is 8, while the optimal value for CAP is 8.5.

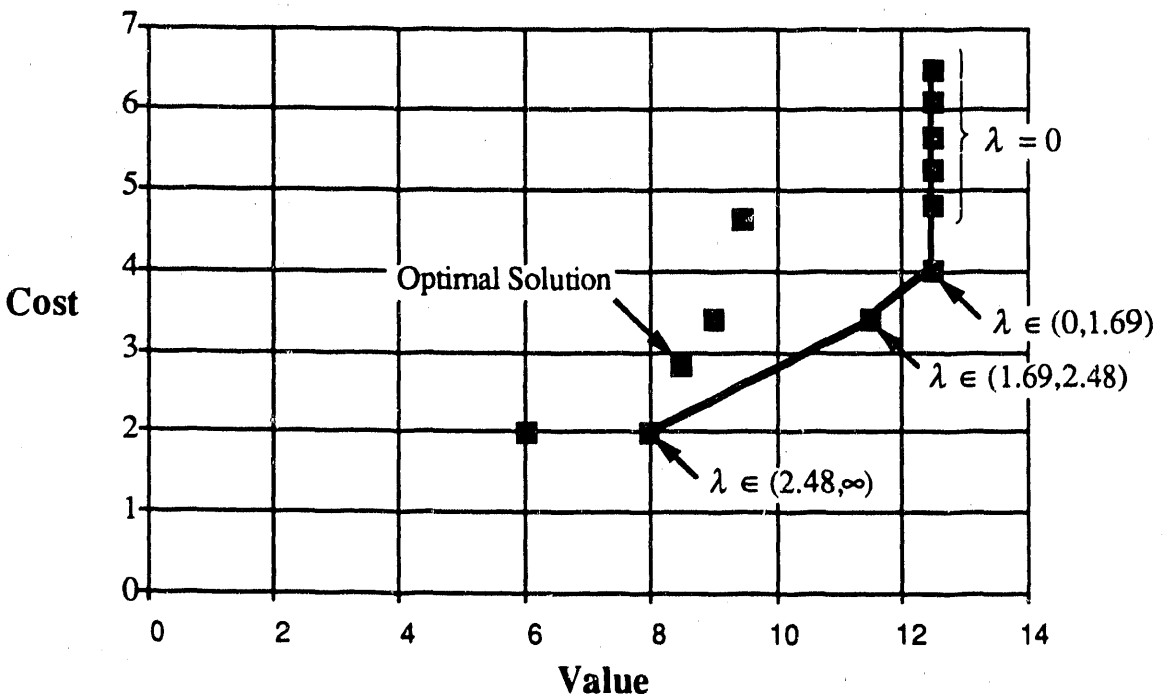

Figure B.1: Feasible solutions to CAP(λ) are plotted according to value and cost. The optimal solution(s) to CAP(λ) for each value of λ and the optimal solution to CAP are indicated. For this problem $\lambda^* = 2.48$.

APPENDIX C: Detailed Computational Results

This appendix contains complete tables of compuational results. Below is a listing of the abbreviations and terms used in the tables.

Problem Type (see Section 6.2)

Uniform, Clusters, Outliers: Refers to the method by which the cost matrix is generated (as described in Section 6.2).

v ~ equal: All nodes are given equal values.

v ~ uni(1,10): Node values are integers uniformly distributed between 1 and 10.

v ~ uni(1,3): Node values are integers uniformly distributed between 1 and 3.

v ~ uni(1,100): Node values are integers uniformly distributed between 1 and 100.

v ~ exp: Node values are exponentially distributed with a mean of 5 and rounded up to integer values.

B: Defines the budget as a fraction of the approximate cost of a complete TSP tour.

Algorithms – Upper Bounding Methods (see Sections 7.1 – 7.3)

KP: The knapsack bound by Laporte and Martello.

IKP: The improved knapsack bound.

TKP: The tighter knapsack bound.

PAB: The parametric assignment bound.

CABλ: The cost-constrained assignment bound.

Algorithms – Heuristic Algorithms (see Sections 9.1 – 9.2)

NewH: The new heuristic algorithm.

MFH: The Multi-Faceted Heuristic by Golden, Wang, and Liu.

Tsi: The heuristic algorithm by Tsiligirides.

CofG: The center of gravity algorithm by Golden, Levy, and Vohra.

GLD – the heuristic algorithm by Golden, Levy, and Dahl.

Algorithms – Variations of NewH (see Section 9.3)

no agg.: NewH without aggregated node values.

no LM: NewH without the learning measure.

no updat.: NewH without the updating of aggregated node values.

no foc. pts.: NewH without the use of focus points.

no rand.: NewH without randomization in the node selection process.

no foc. no rand.: NewH without the use of focus points and without randomization.

chp. insert.: Cheapest insertion algorithm followed by two-opt and node swapping.

Problem type	B	IKP	KP	TKP	PAB	CABλ
Uniform		0.00	0.00	0.03	0.11	0.09
$v \sim$ equal	0.25	11.33	**119.00**	7.33	**104.33**	**104.33**
Uniform		0.00	0.00	0.18	0.12	0.12
$v \sim$ equal	0.50	33.72	**63.20**	30.59	**52.67**	**52.67**
Uniform		0.00	0.01	0.27	0.06	0.08
$v \sim$ equal	0.75	16.67	**25.50**	16.04	**23.67**	**23.67**
Uniform		0.01	0.00	0.03	—	0.08
$v \sim$ uni(1,10)	0.25	26.03	**136.29**	**7.23**	—	**110.25**
Uniform		0.01	0.00	0.19	—	0.11
$v \sim$ uni(1,10)	0.50	39.65	**60.30**	37.99	—	**51.65**
Uniform		0.01	0.00	0.27	—	0.07
$v \sim$ uni(1,10)	0.75	14.90	**18.56**	15.02	—	**17.50**
Uniform		0.00	0.00	0.02	—	0.11
$v \sim$ uni(1,3)	0.25	30.81	**141.63**	**7.74**	—	**109.74**
Uniform		0.00	0.00	0.19	—	0.12
$v \sim$ uni(1,3)	0.50	37.37	**59.82**	**33.91**	—	**49.19**
Uniform		0.00	0.00	0.27	—	0.09
$v \sim$ uni(1,3)	0.75	14.99	**20.69**	14.59	—	**18.46**
Uniform		0.00	0.01	0.03	—	0.10
$v \sim$ uni(1,100)	0.25	39.99	**157.78**	**12.81**	—	**123.70**
Uniform		0.00	0.00	0.20	—	0.14
$v \sim$ uni(1,100)	0.50	45.65	**67.38**	**41.26**		**54.57**
Uniform		0.00	0.01	0.28	—	0.09
$v \sim$ uni(1,100)	0.75	14.40	**16.75**	14.45	—	**16.13**
Uniform		0.00	0.00	0.03	—	0.09
$v \sim$ exp	0.25	31.03	**173.63**	**4.58**	—	**139.70**
Uniform		0.00	0.00	0.20	—	0.13
$v \sim$ exp	0.50	44.62	**60.19**	**41.35**	—	**51.34**
Uniform		0.01	0.00	0.27	—	0.10
$v \sim$ exp	0.75	10.36	**13.34**	10.05	—	**12.22**

Table C-1. Computational results comparing the performance of upper bounding methods using Euclidean test problems. All problems have 20 nodes and results are averaged over a sample size of 10. The numbers in larger print are the average percent errors from optimality of the upper bounds while the numbers in smaller print are the average computation times. Bold type indicates that, when compared with IKP, the difference in performance was statistically significant at the 95% level.

Problem type	B	IKP	KP	TKP	PAB	CABλ
Clusters		0.00	0.00	0.02	0.13	0.10
$v \sim$ equal	0.25	14.72	**209.68**	10.44	**189.37**	**216.04**
Clusters		0.01	0.00	0.11	0.10	0.10
$v \sim$ equal	0.50	58.51	**112.84**	**39.80**	**102.60**	**102.60**
Clusters		0.00	0.00	0.25	0.00	0.03
$v \sim$ equal	0.75	29.58	29.58	29.58	29.58	29.58
Clusters		0.00	0.00	0.02	—	0.11
$v \sim$ uni(1,10)	0.25	14.65	**261.91**	8.78	—	**233.76**
Clusters		0.00	0.00	0.12	—	0.11
$v \sim$ uni(1,10)	0.50	48.89	**97.36**	**29.86**	—	**94.16**
Clusters		0.01	0.00	0.25	—	0.03
$v \sim$ uni(1,10)	0.75	25.66	25.66	25.66	—	25.66
Outliers		0.00	0.00	0.04	0.12	0.09
$v \sim$ equal	0.25	26.61	**112.98**	16.01	**101.31**	**141.31**
Outliers		0.01	0.00	0.20	0.12	0.13
$v \sim$ equal	0.50	23.41	**49.13**	21.91	**35.32**	**35.32**
Outliers		0.01	0.00	0.27	0.11	0.12
$v \sim$ equal	0.75	6.93	**14.46**	8.11	9.36	9.36
Outliers		0.00	0.00	0.04	—	0.10
$v \sim$ uni(1,10)	0.25	46.38	**142.15**	**22.61**	—	**128.24**
Outliers		0.01	0.00	0.22	—	0.14
$v \sim$ uni(1,10)	0.50	25.73	**41.66**	23.87	—	**31.39**
Outliers		0.00	0.00	0.27	—	0.14
$v \sim$ uni(1,10)	0.75	8.77	**11.65**	8.94	—	9.88

Table C-1 (cont.). Computational results comparing the performance of upper bounding methods using Euclidean test problems. All problems have 20 nodes and results are averaged over a sample size of 10. The numbers in larger print are the average percent errors from optimality of the upper bounds while the numbers in smaller print are the average computation times. Bold type indicates that, when compared with IKP, the difference in performance was statistically significant at the 95% level.

Problem type	B	IKP	KP	TKP	PAB	CABλ
Uniform		0.01	0.00	0.09	0.12	0.11
$v \sim$ equal	0.25	26.83	**76.12**	21.06	**54.01**	**54.01**
Uniform		0.01	0.00	0.26	0.11	0.13
$v \sim$ equal	0.50	14.17	**27.11**	12.08	16.27	17.70
Uniform		0.00	0.00	0.28	0.10	0.13
$v \sim$ equal	0.75	10.76	12.49	10.76	**6.83**	**6.83**
Uniform		0.00	0.00	0.10	—	0.11
$v \sim$ uni(1,10)	0.25	38.32	**83.60**	**26.76**	—	**73.46**
Uniform		0.00	0.00	0.26	—	0.15
$v \sim$ uni(1,10)	0.50	15.99	**24.19**	14.55	—	15.79
Uniform		0.01	0.00	0.28	—	0.14
$v \sim$ uni(1,10)	0.75	8.64	8.94	8.52	—	**7.23**
Uniform		0.01	0.00	0.10	—	0.10
$v \sim$ uni(1,3)	0.25	32.64	**80.78**	**25.26**	—	**55.82**
Uniform		0.00	0.00	0.26	—	0.13
$v \sim$ uni(1,3)	0.50	15.61	**27.15**	13.58	—	18.10
Uniform		0.00	0.00	0.29	—	0.13
$v \sim$ uni(1,3)	0.75	8.78	9.59	8.78	—	**6.60**
Uniform		0.00	0.00	0.10	—	0.13
$v \sim$ uni(1,100)	0.25	36.97	**88.26**	**26.38**	—	**62.15**
Uniform		0.00	0.00	0.27	—	0.15
$v \sim$ uni(1,100)	0.50	20.20	**29.89**	18.26	—	19.01
Uniform		0.01	0.00	0.28	—	0.13
$v \sim$ uni(1,100)	0.75	9.27	9.32	9.17	—	**8.36**
Uniform		0.00	0.00	0.10	—	0.11
$v \sim$ exp	0.25	43.52	**91.68**	**31.06**	—	**78.70**
Uniform		0.01	0.00	0.27	—	0.14
$v \sim$ exp	0.50	14.77	**21.30**	13.31	—	13.88
Uniform		0.00	0.00	0.29	—	0.14
$v \sim$ exp	0.75	6.16	6.48	6.16	—	**5.08**

Table C-2. Computational results comparing the performance of upper bounding methods using non-Euclidean test problems. All problems have 20 nodes and results are averaged over a sample size of 10. The numbers in larger print are the average percent errors from optimality of the upper bounds while the numbers in smaller print are the average computation times. Bold type indicates that, when compared with IKP, the difference in performance was statistically significant at the 95% level.

Problem type	B	IKP	KP	TKP	PAB	CABλ
Clusters		0.00	0.00	0.04	0.11	0.12
v ~ equal	0.25	41.49	**348.76**	24.47	**326.60**	**126.60**
Clusters		0.01	0.00	0.15	0.04	0.07
v ~ equal	0.50	40.86	**81.84**	36.14	**80.67**	**74.67**
Clusters		0.00	0.00	0.26	0.01	0.03
v ~ equal	0.75	23.84	23.84	23.84	24.60	23.84
Clusters		0.00	0.00	0.04	—	0.13
v ~ uni(1,10)	0.25	43.08	**1330.05**	21.82	—	**133.78**
Clusters		0.01	0.00	0.16	—	0.07
v ~ uni(1,10)	0.50	37.71	**73.27**	32.45	—	**69.74**
Clusters		0.00	0.00	0.26	—	0.03
v ~ uni(1,10)	0.75	23.50	23.50	23.50	—	23.50
Outliers		0.00	0.00	0.14	0.13	0.13
v ~ equal	0.25	28.43	**58.95**	24.79	**44.73**	**44.73**
Outliers		0.00	0.00	0.27	0.12	0.16
v ~ equal	0.50	8.65	**16.13**	9.28	6.88	6.88
Outliers		0.01	0.00	0.27	0.12	0.15
v ~ equal	0.75	8.22	8.77	8.77	**3.33**	**3.33**
Outliers		0.00	0.00	0.15	—	0.13
v ~ uni(1,10)	0.25	32.26	**57.31**	27.20	—	**44.54**
Outliers		0.00	0.00	0.26	—	0.17
v ~ uni(1,10)	0.50	8.74	**12.78**	8.74	—	**6.82**
Outliers		0.01	0.00	0.27	—	0.18
v ~ uni(1,10)	0.75	4.64	4.73	4.73	—	**2.83**

Table C-2 (cont.). Computational results comparing the performance of upper bounding methods using non-Euclidean test problems. All problems have 20 nodes and results are averaged over a sample size of 10. The numbers in larger print are the average percent errors from optimality of the upper bounds while the numbers in smaller print are the average computation times. Bold type indicates that, when compared with IKP, the difference in performance was statistically significant at the 95% level.

Problem type	*B*	NewH	MFH	Tsi	CofG	GLD
Uniform		0.62	0.43	0.22	0.02	0.00
$v \sim$ equal	0.25	0.00	0.00	0.00	5.67	7.67
Uniform		0.86	0.65	0.81	0.03	0.02
$v \sim$ equal	0.50	0.00	0.00	**4.76**	**8.85**	**10.65**
Uniform		1.33	1.23	1.34	0.07	0.04
$v \sim$ equal	0.75	0.00	0.00	**3.17**	**9.50**	**10.13**
Uniform		0.61	0.41	0.20	0.02	0.01
$v \sim$ uni(1,10)	0.25	0.00	0.77	0.27	**4.95**	**9.40**
Uniform		0.85	0.62	0.74	0.04	0.02
$v \sim$ uni(1,10)	0.50	0.00	0.14	**4.82**	**4.87**	**11.84**
Uniform		1.30	1.24	1.24	0.07	0.03
$v \sim$ uni(1,10)	0.75	0.00	0.10	**4.41**	**6.13**	**13.53**
Uniform		0.62	0.42	0.21	0.02	0.01
$v \sim$ uni(1,3)	0.25	0.00	0.00	0.00	**12.54**	4.85
Uniform		0.86	0.65	0.77	0.03	0.02
$v \sim$ uni(1,3)	0.50	0.00	0.00	**2.63**	**7.17**	**12.86**
Uniform		1.28	1.15	1.27	0.07	0.04
$v \sim$ uni(1,3)	0.75	0.00	0.40	**4.25**	**3.37**	**12.90**
Uniform		0.62	0.42	0.20	0.02	0.01
$v \sim$ uni(1,100)	0.25	0.00	0.00	0.58	**12.05**	**4.87**
Uniform		0.85	0.64	0.72	0.04	0.02
$v \sim$ uni(1,100)	0.50	0.36	0.52	**2.06**	**6.31**	**16.63**
Uniform		1.27	1.14	1.20	0.06	0.04
$v \sim$ uni(1,100)	0.75	0.72	0.29	**5.07**	1.16	**18.62**
Uniform		0.61	0.42	0.19	0.02	0.00
$v \sim$ exp	0.25	0.38	0.38	0.75	7.04	**9.44**
Uniform		0.85	0.65	0.66	0.03	0.02
$v \sim$ exp	0.50	1.13	1.45	**4.42**	1.85	**20.98**
Uniform		1.23	0.99	1.11	0.06	0.03
$v \sim$ exp	0.75	0.00	0.20	**7.11**	**1.97**	**17.75**

Table C-3. Computational results comparing the performance of heuristic algorithms using Euclidean test problems. All problems have 20 nodes and results are averaged over a sample size of 10. The numbers in larger print are the average percent errors from optimality while the numbers in smaller print are the average computation times. Bold type indicates that, when compared with NewH, the difference in performance was statistically significant at the 95% level.

Problem type	*B*	NewH	MFH	Tsi	CofG	GLD
Clusters		0.62	0.44	0.21	0.02	0.00
$v \sim$ equal	0.25	0.00	0.00	1.43	5.00	6.19
Clusters		0.86	0.66	0.63	0.03	0.02
$v \sim$ equal	0.50	0.00	0.00	**6.82**	**4.06**	**14.01**
Clusters		1.44	1.43	1.28	0.08	0.03
$v \sim$ equal	0.75	0.00	0.00	**5.11**	2.69	**6.98**
Clusters		0.62	0.42	0.20	0.02	0.01
$v \sim$ uni(1,10)	0.25	0.00	0.00	0.65	1.05	8.69
Clusters		0.85	0.62	0.60	0.04	0.02
$v \sim$ uni(1,10)	0.50	0.16	0.16	**6.30**	**3.12**	**27.48**
Clusters		1.41	1.33	1.20	0.06	0.04
$v \sim$ uni(1,10)	0.75	0.00	0.00	**5.72**	**4.96**	**10.91**
Outliers		0.65	0.46	0.28	0.02	0.01
$v \sim$ equal	0.25	0.00	0.00	0.00	3.93	**11.43**
Outliers		1.03	0.83	0.98	0.05	0.03
$v \sim$ equal	0.50	0.00	0.00	2.39	**4.70**	**4.82**
Outliers		1.62	1.51	1.47	0.09	0.04
$v \sim$ equal	0.75	0.00	0.00	1.70	**4.68**	**4.61**
Outliers		0.65	0.43	0.26	0.02	0.01
$v \sim$ uni(1,10)	0.25	0.00	0.00	1.60	**7.13**	**15.65**
Outliers		1.01	0.81	0.86	0.04	0.03
$v \sim$ uni(1,10)	0.50	0.14	0.14	**3.95**	**3.97**	**13.04**
Outliers		1.55	1.46	1.31	0.07	0.03
$v \sim$ uni(1,10)	0.75	0.67	0.28	**6.04**	2.68	**6.28**

Table C-3 (cont.). Computational results comparing the performance of heuristic algorithms using Euclidean test problems. All problems have 20 nodes and results are averaged over a sample size of 10. The numbers in larger print are the average percent errors from optimality while the numbers in smaller print are the average computation times. Bold type indicates that, when compared with NewH, the difference in performance was statistically significant at the 95% level.

Problem type	B	NewH	MFH	Tsi	CofG	GLD
Uniform		0.57	0.61	0.57	0.02	0.03
$v \sim$ equal	0.25	0.00	0.00	1.00	**11.58**	**9.10**
Uniform		1.00	1.26	1.38	0.04	0.04
$v \sim$ equal	0.50	0.71	0.71	**0.62**	**8.25**	**8.96**
Uniform		1.42	2.08	1.65	0.07	0.05
$v \sim$ equal	0.75	0.00	0.00	1.67	**3.37**	**3.95**
Uniform		0.56	0.59	0.52	0.02	0.02
$v \sim$ uni(1,10)	0.25	0.00	0.29	**1.20**	**12.30**	**12.03**
Uniform		0.95	1.23	1.25	0.03	0.04
$v \sim$ uni(1,10)	0.50	0.10	0.00	**3.10**	**4.82**	**12.73**
Uniform		1.40	1.99	1.54	0.06	0.05
$v \sim$ uni(1,10)	0.75	0.00	0.19	**3.51**	**2.54**	**7.10**
Uniform		0.57	0.61	0.54	0.02	0.02
$v \sim$ uni(1,3)	0.25	0.00	0.91	2.03	**15.39**	**12.10**
Uniform		0.95	1.26	1.28	0.04	0.04
$v \sim$ uni(1,3)	0.50	0.00	0.00	**2.05**	3.32	**11.56**
Uniform		1.38	1.93	1.58	0.06	0.06
$v \sim$ uni(1,3)	0.75	0.00	0.00	**4.01**	**2.89**	**7.24**
Uniform		0.57	0.57	0.53	0.03	0.02
$v \sim$ uni(1,100)	0.25	0.00	0.00	0.35	**7.27**	**13.83**
Uniform		0.95	1.17	1.20	0.03	0.05
$v \sim$ uni(1,100)	0.50	0.02	0.22	**2.96**	**5.69**	**16.86**
Uniform		1.37	1.99	1.50	0.05	0.06
$v \sim$ uni(1,100)	0.75	0.09	0.03	**3.38**	**2.42**	**9.75**
Uniform		0.56	0.54	0.44	0.02	0.02
$v \sim$ exp	0.25	0.66	0.37	1.74	**3.58**	**13.74**
Uniform		0.96	1.26	1.12	0.04	0.05
$v \sim$ exp	0.50	0.00	0.10	**3.73**	2.02	**15.00**
Uniform		1.38	1.96	1.48	0.06	0.05
$v \sim$ exp	0.75	0.11	0.38	**3.55**	**1.99**	**6.23**

Table C-4. Computational results comparing the performance of heuristic algorithms using non-Euclidean test problems. All problems have 20 nodes and results are averaged over a sample size of 10. The numbers in larger print are the average percent errors from optimality while the numbers in smaller print are the average computation times. Bold type indicates that, when compared with NewH, the difference in performance was statistically significant at the 95% level.

Problem type	*B*	NewH	MFH	Tsi	CofG	GLD
Clusters		0.45	0.41	0.30	0.02	0.01
$v \sim$ equal	0.25	0.00	0.00	2.50	**10.32**	8.10
Clusters		0.76	0.94	0.94	0.03	0.03
$v \sim$ equal	0.50	1.67	0.00	3.81	**12.92**	12.34
Clusters		1.11	1.47	1.59	0.04	0.05
$v \sim$ equal	0.75	0.00	0.00	1.34	2.59	**7.02**
Clusters		0.45	0.42	0.29	0.02	0.01
$v \sim$ uni(1,10)	0.25	0.00	0.00	0.19	1.97	**14.54**
Clusters		0.74	0.89	0.87	0.02	0.04
$v \sim$ uni(1,10)	0.50	0.00	0.15	**0.85**	**9.16**	**11.65**
Clusters		1.04	1.59	1.42	0.04	0.05
$v \sim$ uni(1,10)	0.75	0.00	0.00	0.82	2.07	**6.06**
Outliers		0.65	0.74	0.79	0.03	0.02
$v \sim$ equal	0.25	0.00	0.00	0.00	**13.62**	**12.90**
Outliers		1.20	1.63	1.49	0.04	0.05
$v \sim$ equal	0.50	0.00	0.00	1.25	**5.63**	**4.26**
Outliers		1.67	2.45	1.72	0.07	0.05
$v \sim$ equal	0.75	0.00	0.00	1.05	0.00	**2.13**
Outliers		0.64	0.71	0.74	0.03	0.03
$v \sim$ uni(1,10)	0.25	0.00	0.48	**2.93**	**8.48**	**19.75**
Outliers		1.19	1.59	1.38	0.05	0.05
$v \sim$ uni(1,10)	0.50	0.00	0.29	**3.06**	**2.64**	**7.09**
Outliers		1.56	2.35	1.65	0.07	0.05
$v \sim$ uni(1,10)	0.75	0.00	0.00	**2.28**	**0.96**	**5.84**

Table C-4 (cont.). Computational results comparing the performance of heuristic algorithms using non-Euclidean test problems. All problems have 20 nodes and results are averaged over a sample size of 10. The numbers in larger print are the average percent errors from optimality while the numbers in smaller print are the average computation times. Bold type indicates that, when compared with NewH, the difference in performance was statistically significant at the 95% level.

Problem type	B	Euclidean problems		non-Euclidean problems	
		NewH	MFH	NewH	MFH
Uniform		3.66	2.19	4.28	4.42
$v \sim$ equal	0.25	0.16	**1.48**	1.63	1.80
Uniform		7.75	7.45	10.84	13.60
$v \sim$ equal	0.50	0.49	**1.36**	1.19	1.19
Uniform		16.62	19.51	20.66	26.54
$v \sim$ equal	0.75	0.88	1.29	0.44	0.64
Uniform		3.59	2.12	4.12	4.15
$v \sim$ uni(1,10)	0.25	0.29	0.98	2.40	1.40
Uniform		7.48	6.89	10.05	12.76
$v \sim$ uni(1,10)	0.50	1.05	1.14	1.21	1.34
Uniform		15.95	18.17	19.56	25.64
$v \sim$ uni(1,10)	0.75	0.97	1.20	0.53	0.73
Uniform		3.62	2.22	4.25	4.30
$v \sim$ uni(1,3)	0.25	0.32	**1.20**	2.22	1.62
Uniform		7.59	7.01	10.43	13.04
$v \sim$ uni(1,3)	0.50	0.82	1.14	1.09	1.42
Uniform		16.16	18.69	19.96	25.80
$v \sim$ uni(1,3)	0.75	0.57	1.02	0.59	0.73
Uniform		3.58	2.04	4.13	4.12
$v \sim$ uni(1,100)	0.25	0.07	**0.96**	2.29	2.03
Uniform		7.41	6.71	9.90	12.46
$v \sim$ uni(1,100)	0.50	1.39	**0.47**	1.41	**1.90**
Uniform		15.69	17.77	19.12	25.06
$v \sim$ uni(1,100)	0.75	0.84	0.85	0.33	**0.71**
Uniform		3.57	2.00	4.11	3.87
$v \sim$ exp	0.25	0.44	0.66	2.94	**1.59**
Uniform		7.32	6.31	9.66	12.14
$v \sim$ exp	0.50	1.47	**0.72**	1.10	1.37
Uniform		15.61	18.03	19.04	24.97
$v \sim$ exp	0.75	0.84	0.83	0.27	**0.44**

Table C-5. Computational results comparing the performance of NewH and MFH using 50-node test problems. Results for Euclidean problems are averaged over a sample size of 40 and results for non-Euclidean problems are averaged over a sample size of 20. The numbers in larger print are the average percent errors from the best known solution while the numbers in smaller print are the average computation times. Bold type indicates that, when compared with NewH, the difference in performance was statistically significant at the 95% level.

Problem type	B	Euclidean problems		non-Euclidean problems	
		NewH	MFH	NewH	MFH
Clusters		3.66	2.29	3.51	3.23
$v \sim$ equal	0.25	0.00	**3.26**	0.61	1.06
Clusters		7.86	7.56	8.46	10.54
$v \sim$ equal	0.50	0.08	**1.24**	0.00	0.32
Clusters		17.28	20.45	16.48	22.28
$v \sim$ equal	0.75	0.03	0.66	0.00	0.00
Clusters		3.61	2.17	3.47	2.97
$v \sim$ uni(1,10)	0.25	0.04	**2.61**	0.57	**2.73**
Clusters		7.62	7.05	8.33	10.14
$v \sim$ uni(1,10)	0.50	0.50	**1.19**	0.11	0.36
Clusters		16.77	19.25	16.25	21.51
$v \sim$ uni(1,10)	0.75	0.63	**1.16**	0.06	0.07
Outliers		4.53	3.38	10.94	13.75
$v \sim$ equal	0.25	0.51	0.64	0.94	0.56
Outliers		13.09	14.88	23.35	32.85
$v \sim$ equal	0.50	0.87	0.61	0.11	0.11
Outliers		25.27	32.75	26.39	40.54
$v \sim$ equal	0.75	0.16	**0.53**	0.00	0.00
Outliers		4.43	3.23	10.82	13.27
$v \sim$ uni(1,10)	0.25	0.63	0.80	0.68	0.89
Outliers		12.55	13.83	23.14	32.30
$v \sim$ uni(1,10)	0.50	1.03	0.90	0.15	0.26
Outliers		24.70	31.83	26.10	39.72
$v \sim$ uni(1,10)	0.75	0.43	0.53	0.05	0.30

Table C-5 (cont.). Computational results comparing the performance of NewH and MFH using 50-node test problems. Results for Euclidean problems are averaged over a sample size of 40 and results for non-Euclidean problems are averaged over a sample size of 20. The numbers in larger print are the average percent errors from the best known solution while the numbers in smaller print are the average computation times. Bold type indicates that, when compared with NewH, the difference in performance was statistically significant at the 95% level.

Problem type	*B*	NewH	no agg.	no LM	no updat.	no foc.pts.	no rand.
Uniform		3.66	2.75	3.58	2.76	0.77	0.34
$v \sim$ equal	0.25	0.16	0.13	0.35	0.13	**1.99**	**2.68**
Uniform		7.75	6.63	7.67	6.80	1.62	0.75
$v \sim$ equal	0.50	0.49	**1.10**	0.49	**1.00**	**2.50**	**2.99**
Uniform		16.62	15.28	16.49	15.52	3.37	1.63
$v \sim$ equal	0.75	0.88	1.12	0.65	**1.17**	**2.65**	**2.87**
Uniform		3.59	2.68	3.53	2.71	0.76	0.34
$v \sim$ uni(1,10)	0.25	0.29	0.23	0.27	0.61	**1.43**	**2.33**
Uniform		7.48	6.31	7.38	6.60	1.55	0.74
$v \sim$ uni(1,10)	0.50	1.05	1.39	**0.70**	1.40	**2.29**	**3.29**
Uniform		15.95	14.70	15.96	15.21	3.26	1.61
$v \sim$ uni(1,10)	0.75	0.97	1.02	0.94	1.22	**2.48**	**2.84**
Uniform		3.62	2.72	3.56	2.73	0.77	0.34
$v \sim$ uni(1,3)	0.25	0.32	0.33	0.39	0.00	**1.63**	**2.55**
Uniform		7.59	6.47	7.50	6.64	1.55	0.74
$v \sim$ uni(1,3)	0.50	0.82	0.85	0.74	1.14	**2.73**	**2.55**
Uniform		16.16	14.82	15.96	15.18	3.31	1.61
$v \sim$ uni(1,3)	0.75	0.57	**1.09**	0.46	**1.11**	**2.17**	**3.15**
Uniform		3.58	2.65	3.51	2.69	0.76	0.33
$v \sim$ uni(1,100)	0.25	0.07	0.24	0.15	0.05	**1.64**	**2.62**
Uniform		7.41	6.19	7.27	6.54	1.53	0.72
$v \sim$ uni(1,100)	0.50	1.39	1.18	1.40	1.67	**3.32**	**4.01**
Uniform		15.69	14.30	15.65	15.10	3.18	1.53
$v \sim$ uni(1,100)	0.75	0.84	**0.64**	0.92	**1.14**	**2.53**	**3.33**
Uniform		3.57	2.64	3.51	2.68	0.75	0.33
$v \sim$ exp	0.25	0.44	**0.04**	0.37	0.78	**1.60**	**3.29**
Uniform		7.32	6.19	7.22	6.50	1.56	0.71
$v \sim$ exp	0.50	1.47	**1.01**	1.67	**2.81**	**4.74**	**5.44**
Uniform		15.61	14.41	15.49	15.13	3.17	1.59
$v \sim$ exp	0.75	0.84	**0.60**	1.05	**1.66**	**2.97**	**2.73**

Table C-6. Computational results showing the effects of dropping individual features of NewH using Euclidean test problems. All problems have 50 nodes and results are averaged over a sample size of 40. The numbers in larger print are the average percent errors from optimality while the numbers in smaller print are the average computation times. Bold type indicates that, when compared with NewH, the difference in performance was statistically significant at the 95% level.

Problem type	B	NewH	no agg.	no LM	no updat.	no foc.pts.	no rand.
Clusters		3.66	2.76	3.59	2.77	0.79	0.34
$v \sim$ equal	0.25	0.00	0.12	0.00	0.00	0.14	**0.70**
Clusters		7.86	6.64	7.73	6.89	1.66	0.76
$v \sim$ equal	0.50	0.08	**0.78**	0.33	**0.48**	**2.54**	**2.43**
Clusters		17.28	15.40	17.29	16.28	3.57	1.74
$v \sim$ equal	0.75	0.03	**0.67**	0.30	**0.67**	**2.11**	**1.97**
Clusters		3.61	2.72	3.53	2.72	0.77	0.34
$v \sim$ uni(1,10)	0.25	0.04	0.03	0.04	0.14	**1.03**	**1.54**
Clusters		7.62	6.42	7.52	6.77	1.60	0.73
$v \sim$ uni(1,10)	0.50	0.50	**0.73**	0.49	**0.74**	**2.83**	**2.63**
Clusters		16.77	15.21	16.70	15.92	3.36	1.67
$v \sim$ uni(1,10)	0.75	0.63	0.57	0.74	**0.89**	**2.63**	**2.23**
Outliers		4.53	3.58	4.45	3.62	0.98	0.42
$v \sim$ equal	0.25	0.51	**0.98**	0.36	0.60	**1.62**	**4.00**
Outliers		13.09	11.72	13.01	11.92	2.68	1.30
$v \sim$ equal	0.50	0.87	**1.09**	0.67	0.79	**2.23**	**3.34**
Outliers		25.27	24.38	25.28	24.36	5.15	2.57
$v \sim$ equal	0.75	0.16	**0.37**	0.32	0.16	**0.69**	**1.39**
Outliers		4.43	3.50	4.35	3.54	0.96	0.42
$v \sim$ uni(1,10)	0.25	0.63	0.33	0.38	0.84	**2.19**	**3.40**
Outliers		12.55	11.26	12.58	11.76	2.65	1.26
$v \sim$ uni(1,10)	0.50	1.03	0.80	1.18	0.95	**2.69**	**3.22**
Outliers		24.70	23.31	24.71	24.21	5.11	2.55
$v \sim$ uni(1,10)	0.75	0.43	**0.30**	0.28	0.44	**0.93**	**1.46**

Table C-6 (cont.). Computational results showing the effects of dropping individual features of NewH using Euclidean test problems. All problems have 50 nodes and results are averaged over a sample size of 40. The numbers in larger print are the average percent errors from optimality while the numbers in smaller print are the average computation times. Bold type indicates that, when compared with NewH, the difference in performance was statistically significant at the 95% level.

Problem type	B	NewH	no agg.	no LM	no updat.	no foc.pts.	no rand.
Uniform		4.28	3.46	4.23	3.53	1.11	0.40
$v \sim$ equal	0.25	1.63	2.67	1.47	1.85	**3.86**	**5.93**
Uniform		10.84	9.68	10.70	9.95	2.77	1.04
$v \sim$ equal	0.50	1.19	1.45	1.86	1.58	**2.79**	**3.73**
Uniform		20.66	19.60	20.30	19.35	5.13	2.05
$v \sim$ equal	0.75	0.44	0.32	0.54	0.64	**1.30**	**1.75**
Uniform		4.12	3.29	4.07	3.43	1.06	0.39
$v \sim$ uni(1,10)	0.25	2.40	2.62	2.48	2.15	**4.71**	**5.82**
Uniform		10.05	8.87	9.95	9.51	2.58	1.02
$v \sim$ uni(1,10)	0.50	1.21	1.06	1.24	1.36	**2.18**	**3.25**
Uniform		19.56	18.46	19.36	19.52	4.88	2.04
$v \sim$ uni(1,10)	0.75	0.53	0.20	0.59	0.49	**0.97**	**1.30**
Uniform		4.25	3.41	4.18	3.51	1.09	0.41
$v \sim$ uni(1,3)	0.25	2.22	1.73	1.22	1.63	3.32	**6.40**
Uniform		10.43	9.36	10.23	9.79	2.63	1.00
$v \sim$ uni(1,3)	0.50	1.09	0.82	0.93	1.48	**2.08**	**2.91**
Uniform		19.96	18.99	19.69	19.01	5.00	2.02
$v \sim$ uni(1,3)	0.75	0.59	0.63	0.96	0.80	**1.52**	**1.75**
Uniform		4.13	3.30	4.10	3.43	1.07	0.40
$v \sim$ uni(1,100)	0.25	2.29	1.72	2.13	1.69	**3.63**	**6.65**
Uniform		9.90	8.78	9.95	9.34	2.59	1.00
$v \sim$ uni(1,100)	0.50	1.41	**0.97**	1.51	1.73	**2.70**	**3.66**
Uniform		19.12	18.04	18.97	19.36	4.85	1.91
$v \sim$ uni(1,100)	0.75	0.33	0.40	0.43	0.52	**1.00**	**1.75**
Uniform		4.11	3.17	4.06	3.44	1.06	0.39
$v \sim$ exp	0.25	2.94	1.71	2.71	**3.78**	**4.37**	**8.41**
Uniform		9.66	8.45	9.81	9.46	2.49	1.01
$v \sim$ exp	0.50	1.10	**0.66**	1.06	**1.83**	**2.05**	**2.54**
Uniform		19.04	17.67	18.97	18.47	4.81	1.79
$v \sim$ exp	0.75	0.27	0.24	0.25	**0.67**	**0.62**	**0.91**

Table C-7. Computational results showing the effects of dropping individual features of NewH using non-Euclidean test problems. All problems have 50 nodes and results are averaged over a sample size of 20. The numbers in larger print are the average percent errors from optimality while the numbers in smaller print are the average computation times. Bold type indicates that, when compared with NewH, the difference in performance was statistically significant at the 95% level.

Problem type	B	NewH	no agg.	no LM	no updat.	no foc.pts.	no rand.
Clusters		3.51	2.71	3.43	2.80	0.95	0.33
$v \sim$ equal	0.25	0.61	1.92	0.53	0.53	0.53	1.57
Clusters		8.46	6.98	8.40	7.48	2.26	0.84
$v \sim$ equal	0.50	0.00	**1.51**	0.00	0.28	0.44	**0.76**
Clusters		16.48	14.29	16.45	15.62	4.21	1.60
$v \sim$ equal	0.75	0.00	0.12	0.00	0.00	0.00	0.23
Clusters		3.47	2.72	3.39	2.76	0.94	0.33
$v \sim$ uni(1,10)	0.25	0.57	0.21	0.40	0.35	0.79	**3.01**
Clusters		8.33	7.07	8.24	7.46	2.22	0.81
$v \sim$ uni(1,10)	0.50	0.11	0.38	0.05	**0.41**	**0.34**	**1.14**
Clusters		16.25	15.09	16.40	15.56	4.07	1.56
$v \sim$ uni(1,10)	0.75	0.06	0.11	0.15	0.11	**0.23**	**0.63**
Outliers		10.94	10.16	10.90	10.13	2.79	1.13
$v \sim$ equal	0.25	0.94	1.39	0.85	1.10	1.39	**2.87**
Outliers		23.35	22.46	22.93	22.80	5.91	2.24
$v \sim$ equal	0.50	0.11	0.11	0.11	0.21	0.11	0.43
Outliers		26.39	25.70	25.98	25.76	6.55	2.51
$v \sim$ equal	0.75	0.00	0.00	0.00	0.00	0.00	0.00
Outliers		10.82	9.70	10.60	10.00	2.75	1.06
$v \sim$ uni(1,10)	0.25	0.68	0.69	0.92	**1.14**	**1.88**	**3.32**
Outliers		23.14	22.43	23.15	22.57	5.84	2.16
$v \sim$ uni(1,10)	0.50	0.15	0.21	0.17	0.25	**0.25**	**0.55**
Outliers		26.10	25.38	26.46	25.52	6.50	2.62
$v \sim$ uni(1,10)	0.75	0.05	0.08	0.08	0.11	0.20	**0.42**

Table C-7 (cont.). Computational results showing the effects of dropping individual features of NewH using non-Euclidean test problems. All problems have 50 nodes and results are averaged over a sample size of 20. The numbers in larger print are the average percent errors from optimality while the numbers in smaller print are the average computation times. Bold type indicates that, when compared with NewH, the difference in performance was statistically significant at the 95% level.

Problem type	B	NewH	no foc.pts.	no rand.	no foc. no rand.	chp. insert.	CofG
Uniform		3.54	0.74	0.33	0.09	0.03	0.14
$v \sim$ equal	0.25	0.31	**2.11**	3.18	**7.07**	7.63	9.90
Uniform		7.65	1.56	0.74	0.17	0.10	0.42
$v \sim$ equal	0.50	0.31	**2.49**	3.65	**8.10**	**9.20**	11.54
Uniform		16.63	3.41	1.65	0.37	0.35	1.10
$v \sim$ equal	0.75	0.82	**2.80**	2.80	**7.14**	**10.99**	8.72
Uniform		3.49	0.72	0.33	0.08	0.03	0.12
$v \sim$ uni(1,10)	0.25	0.36	0.93	**2.64**	**6.50**	**4.37**	10.96
Uniform		7.42	1.52	0.73	0.16	0.09	0.37
$v \sim$ uni(1,10)	0.50	1.09	**2.37**	**3.56**	**7.83**	**11.70**	8.66
Uniform		16.07	3.24	1.65	0.37	0.26	0.93
$v \sim$ uni(1,10)	0.75	0.85	**1.97**	2.54	**5.82**	**8.97**	5.41
Uniform		3.51	0.74	0.33	0.09	0.02	0.13
$v \sim$ uni(1,3)	0.25	0.50	**1.67**	2.77	**5.91**	**9.33**	13.10
Uniform		7.49	1.49	0.72	0.17	0.10	0.44
$v \sim$ uni(1,3)	0.50	0.91	**2.70**	2.71	**7.89**	**13.64**	10.36
Uniform		16.50	3.31	1.64	0.36	0.24	0.92
$v \sim$ uni(1,3)	0.75	0.28	**1.38**	2.20	**5.73**	**10.60**	7.34
Uniform		3.48	0.72	0.32	0.09	0.02	0.11
$v \sim$ uni(1,100)	0.25	0.07	**1.50**	2.47	**6.17**	8.22	10.05
Uniform		7.33	1.47	0.72	0.17	0.09	0.31
$v \sim$ uni(1,100)	0.50	1.43	**3.22**	4.00	**9.20**	**11.33**	7.18
Uniform		15.97	3.22	1.54	0.33	0.24	0.61
$v \sim$ uni(1,100)	0.75	0.53	**1.76**	**3.42**	**6.21**	8.04	5.09
Uniform		3.46	0.72	0.32	0.09	0.02	0.12
$v \sim$ exp	0.25	0.00	0.62	**3.11**	**8.79**	7.17	5.07
Uniform		7.23	1.52	0.70	0.16	0.10	0.30
$v \sim$ exp	0.50	0.90	**4.76**	**5.66**	**10.90**	9.64	7.86
Uniform		15.70	3.08	1.59	0.37	0.28	0.70
$v \sim$ exp	0.75	0.52	**2.60**	2.21	**5.61**	5.50	4.08

Table C-8. Computational results for experiments examining the trade-off between solution quality and computation time. Results are for Euclidean test problems with 50 nodes and are averaged over a sample size of 20. The numbers in larger print are the average percent errors from optimality while the numbers in smaller print are the average computation times. Bold type indicates that, when compared with the algorithm to the immediate left in the table, the difference in performance was statistically significant at the 95% level.

Problem type	B	NewH	no foc.pts.	no rand.	no foc. no rand.	chp. insert.	CofG
Clusters		3.57	0.76	0.33	0.09	0.03	0.11
$v \sim$ equal	0.25	0.00	0.00	0.29	**2.73**	4.83	10.36
Clusters		7.53	1.60	0.72	0.16	0.09	0.38
$v \sim$ equal	0.50	0.00	**3.24**	2.67	**8.41**	**13.72**	11.38
Clusters		17.43	3.61	1.77	0.36	0.25	0.97
$v \sim$ equal	0.75	0.12	**1.55**	1.57	**6.41**	**11.27**	8.23
Clusters		3.54	0.75	0.33	0.09	0.02	0.12
$v \sim$ uni(1,10)	0.25	0.00	**0.97**	0.82	**2.40**	**9.79**	14.79
Clusters		7.30	1.54	0.69	0.17	0.09	0.37
$v \sim$ uni(1,10)	0.50	0.46	**2.96**	2.73	**8.73**	**15.37**	5.96
Clusters		17.10	3.39	1.68	0.35	0.26	0.86
$v \sim$ uni(1,10)	0.75	0.35	**2.11**	2.00	**5.37**	7.36	3.56
Outliers		4.50	0.97	0.42	0.11	0.04	0.24
$v \sim$ equal	0.25	0.81	**2.56**	4.47	**8.98**	10.81	7.86
Outliers		13.37	2.69	1.30	0.29	0.19	0.78
$v \sim$ equal	0.50	0.68	**2.44**	**3.15**	**5.72**	**9.87**	8.68
Outliers		25.67	5.17	2.60	0.53	0.54	1.61
$v \sim$ equal	0.75	0.11	0.31	**1.07**	**2.55**	2.46	1.81
Outliers		4.41	0.95	0.42	0.11	0.04	0.20
$v \sim$ uni(1,10)	0.25	0.37	**2.35**	3.44	**9.00**	10.27	7.57
Outliers		12.74	2.68	1.27	0.30	0.20	0.65
$v \sim$ uni(1,10)	0.50	1.33	**3.25**	3.22	**6.48**	**8.46**	7.02
Outliers		25.00	5.15	2.57	0.56	0.47	1.31
$v \sim$ uni(1,10)	0.75	0.42	**0.65**	**1.32**	**2.93**	**2.24**	1.48

Table C-8 (cont.).. Computational results for experiments examining the trade-off between solution quality and computation time. Results are for Euclidean test problems with 50 nodes and are averaged over a sample size of 20. The numbers in larger print are the average percent errors from optimality while the numbers in smaller print are the average computation times. Bold type indicates that, when compared with the algorithm to the immediate left in the table, the difference in performance was statistically significant at the 95% level.

END

DATE FILMED

02/20/91