AN APPROACH TO A PROBLEM IN NETWORK DESIGN USING GENETIC ALGORITHMS

DISSERTATION

Submitted in Partial Fulfillment
of the Requirements for the
Degree of

DOCTOR OF PHILOSOPHY (Computer Science)
at the
POLYTECHNIC UNIVERSITY

by
Charles Campbell Palmer
April 1994

Approved:

____
Department Head

____
Date

Copy No.

Approved by the Guidance Committee:

Major: Computer Science

Aaron Kershenbaum
Adjunct Professor of Electrical Engineering and Computer Science

Major: Computer Science

Susan Flynn Hummel
Assistant Professor of Computer Science

Major: Computer Science

Richard M. Van Slyke
Professor of Electrical Engineering and Computer Science

Minor: Electrical Engineering

Robert R. Boorstyn
Professor of Electrical Engineering and Computer Science

Microfilm or other copies of this dissertation
are obtainable from

UNIVERSITY MICROFILMS
300 N. Zeeb Road
Ann Arbor, Michigan 48106

Vita

Charles Campbell Palmer was born on August 31, 1956 in Leesville, Louisiana. He attended Louisiana State University and then Oklahoma State University where he earned the Bachelor of Science degree in Computer Science. While working for Standard Oil of Indiana in Tulsa, Oklahoma, he moved with the company to Denver, Colorado where he began part-time graduate studies in Computer Science at the Denver campus of the University of Colorado. After being transferred to New Orleans, Louisiana, he finished his Master of Science degree in Computer Science at Tulane University. He joined IBM's Thomas J. Watson Research Center, Yorktown Heights, New York, in 1984 and entered the part-time doctoral program at Polytechnic University in 1986. He began his dissertation research in 1989 with his advisor, Dr. Aaron Kershenbaum. The entirety of his research was carried out as an approved independent research project at IBM Research.

to Darby, who kept my feet warm as long as she could

Acknowledgements

It's been a very long journey, and I would have made many a wrong turn if it weren't for some very special people who helped me stay between the lines. I would like to thank them now for all the help they gave me.

First, I must thank my parents for the wonderful environment they made for me while I was growing up. Their efforts to introduce me to the wonders of this world and their encouragement to ask questions about it gave me the curiosity that is the lifeblood of original research. I was truly blessed to have them as parents and friends. My brother John, whose innate musical gifts and flourishing theatrical career kept my interest in music alive and kept me from becoming a complete nerd. My grandparents, Walter and Effie Campbell, who were always ready to explain the secrets of fox hunting and the workings of the sewing machine, and J.G. and Elizabeth Palmer, whose stories of travel around the world could hold any youngster's attention for hours. The memories of these four wonderful people still are a source of inspiration.

In 1973, my high school guidance counselor, Joy Russell, did something for me that changed my life forever: she helped get me into the National Science Foundation's Student Science Training Program at Loyola University in New Orleans, Louisiana. There I met one of the most brilliant people I am ever likely to meet: Dr. John F. Christman. This man originated, administered, and taught in this fine program. We spent that summer immersed in computer science and organic chemistry, and we were treated to short courses in aerodynamics, psychology, and genetics. In the midst of all these incredibly fascinating studies, we also learned a lot about ourselves as people. Dr. Christman told us that after that summer, we would all have to refer to our lives as "before the program, and after the program." He was absolutely right.

Many years later, while attending Tulane University next door to Loyola, I met Dr. Larry Reeker who agreed to be my Master's thesis advisor. His guidance through my first real research experience was invaluable. Although, I really could have done without the introduction to Vegemite.

Shortly after coming to IBM's T. J. Watson Research Center in 1984, I entered the Ph.D. program at Polytechnic. Throughout my years of part-time study, my managers have always been very supportive and understanding. I especially want to thank the ones who endured my research: Parviz Kermani, Phil Rosenfeld, Steve Brady, and Hamid Ahmadi. In addition to management support, a person couldn't ask for a finer institution in which to do research.

I have to thank Robert Cahn, who first introduced me to Genetic Algorithms. Nici Schraudolph of the University of California at San Diego was a quite helpful while I was porting his GAucsd 1.4 version of the Genesis tool to IBM's RiscSystem/6000. Cem Ersoy, of Bogazici University in Istanbul, Turkey, and Shiv Panwar of Polytechnic University shared their research into the use of simulated annealing on a problem similar to mine. Their help is greatly appreciated. I must thank Bob Camm and everyone at "the farm" for their superb heavy-duty computing support, even when they would call to ask "is your job really supposed to still be running?" I would have never gotten this beast formatted if it weren't for Andy Shooman who shared his Polytechnic University dissertation LaTeX style file with me, and Jeff Kravitz who was always there to solve my LaTeX problems, even when they were really cockpit problems.

The thanks and appreciation I owe two more people cannot be adequately expressed in words, but I must try. As my advisor, Aaron Kershenbaum was the most patient, understanding, and enthusiastic mentor a student could ever dream of. His tireless explanations were both inspiring and brilliant. He always had time to talk about my work, or anything else for that matter. I would have never survived working full time while doing this research without his unbending confidence in me and his emotional support. I was honored to have him as my advisor, and privileged to have him as a friend.

To Elaine, my wife of seventeen wonderful years and my best friend for even longer, I really owe it all. Without her love and support, and the inspiration she brings to my life, I would probably still be at Louisiana State trying to finish a B.S. degree. Throughout this long effort she selflessly assumed most of our life's chores so that I might have more time to devote to my studies. As a computer science researcher herself, she regularly provided insights and assistance with everything from C program debugging to proofreading to Unix administration. She really is, without a doubt, the best thing that ever happened to me.

Finally, to both of our families for understanding the curtailed holiday visits and to all of our friends who kept on inviting us to parties and outings in spite the many times we had to decline so that I could work on my research: thank you for your patience, I'm finally finished.

Abstract

In the work of communications network design there are several recurring themes: maximizing flows, finding circuits, and finding shortest paths or minimal cost spanning trees, among others. Some of these problems appear to be harder than others. For some, effective algorithms exist for solving them, for others, tight bounds are known, and for still others, researchers have few clues towards a good approach. One of these latter, nastier problems arises in the design of communications networks: the Optimal Communication Spanning Tree Problem (OCSTP). First posed by Hu in 1974, this problem has been shown to be in the family of NP-complete problems. So far, a good, general-purpose approximation algorithm for it has proven elusive.

This thesis describes the design of a genetic algorithm for finding reliably good solutions to the OCSTP. The genetic algorithm approach was thought to be an appropriate choice since they are computationally simple, provide a powerful parallel search capability, and have the ability to move around in the solution space without a dependence upon structure or locality. As an added benefit, well-designed genetic algorithms adapt very well to changes in the problem definition, without suffering changes to the genetic algorithm itself. This adaptability can be used to provide hints to the designer of a new algorithm to what good solutions look like. It can also be applied in cases when an existing algorithm is faced with a change in the problem parameters and its effectiveness under those changes is not known.

The selection of a genetic algorithm approach spawned a prerequisite area of investigation: genetic algorithm researchers have never found a good scheme for efficiently representing and evolving populations of solutions to tree problems. Because of this fact, the design of such a representation became a major part of this research.

Trials of the genetic algorithm showed that it reliably produced results comparable to or better than other approaches, such as heuristics and simulated annealing. It adapted extremely well to changes in the problem description ranging from varying traffic requirements to producing solutions to an entirely different tree-related network design problem.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the work of communications network design there are several recurring themes: maximizing flows, finding circuits, and finding shortest paths or minimal cost spanning trees, among others. For some of these, highly efficient algorithms have been developed that solve them exactly. An algorithm for finding the shortest path between a pair of nodes in a graph was designed by Dijkstra [6]. Prim [35] designed an algorithm to find the minimal cost spanning tree over the n nodes of a graph. Both of these algorithms can accomplish their goal by doing an amount of work that is on the order of n^2 operations. Even for large problems of perhaps thousands of nodes, these algorithms can produce optimal results in a timely enough manner so as to be useful. For other problems, this is not always the case. For some problems, there are no practical algorithms known, or perhaps, even possible. The most famous representative of this genre is the Traveling Salesman Problem (TSP). This problem arises in many real-life situations and it is one of a family of problems that are accepted as being extremely hard: the NP-complete problems. For these kinds of problems, it is generally accepted that any algorithm to exactly solve one of them would have to do an amount of work on the order exponential in n. For example, suppose an algorithm required 2^n operations. For problems of even a reasonable size of 100 nodes, this would amount to 1.2677×10^{30} operations. Even if a computer existed that could perform one billion of these operations per second, it would still take over $40,000,000,000,000$ years to finish.

What makes the NP-complete problems so difficult is not precisely known. Perhaps if this question is ever answered then an algorithm might be designed to get around the "hard part" just as a vaccine may be designed once the offending virus is identified. One may observe that these problems lack two common characteristics of problems that can be efficiently solved. First, their solutions lack structure. For example, in Prim's minimal cost spanning tree algorithm, the heuristic proceeds by always choosing the least expensive link to add to the solution tree, one not already in the tree, that doesn't create a cycle. This problem has an underlying structure that can be used to find an optimal solution. Second, the NP-complete problems lack locality. A problem is said to possess locality when small changes in a solution will result in small changes in the solution's desirability. A gradient search technique can be applied to such problems because of these "neighborhoods" of solutions. Such techniques are much less effective on NP-complete problems.

Years of research have been spent trying to design algorithms that can find reliably good solutions to some of these problems in an acceptable amount of time. A great many of these problems now have algorithms that do not claim to solve them perfectly, but rather to find very good approximate solutions for them. However, since these problems are so hard, it is quite difficult to judge the quality of a solution produced by one of these approximation algorithms; there are no yardsticks against which to measure the quality of a solution. Research has produced bounds that can be used as absolute limits, but the bounds are often too loose to be helpful in most cases.

Another difficulty faced by algorithm developers is that discovering efficient approaches to find good solutions to these kinds of problems requires a keen insight into the problem, algorithmic intuition, and a bit of luck. In addition, the effectiveness of an algorithm may be a function of its input data. Some algorithms, such as Dionne and Florian's [7] algorithm for finding solutions to the optimal communications network problem, work quite well for typical kinds of input data but perform poorly, perhaps even failing to terminate, for some special cases of input data.

Before an algorithm can be designed, the researcher must have some hint as to how to approach the problem. This hint could be in the form of a process that a person might follow in order to find a solution, or perhaps a different process that a person can understand but that requires too much bookeeping for a human to handle. Without this key insight into the problem at hand, the algorithm designer is lost. Once it is designed, the algorithm must be tested and its solutions judged to see if the insight/intuition/luck was correct. Unfortunately, if any of the information supplied to the designer changes over time or if the input data (e.g., traffic flow patterns, telecommunications tariffs, etc.) changes, the algorithm designer must be called back to check whether or not the algorithm will still work under the new conditions. In the worst case, the designer might not even know exactly what information is really being used by the algorithm and what information is being ignored.

While it is widely accepted that these problems are very hard, that does not change the fact that they continue to pop up in such applied fields as computer science, operations research, and in multi-discipline fields like communications network design. It also turns out that some of these problems appear to be harder than others. For some, effective algorithms exist for solving them, for others, tight bounds are known, and for still others, researchers have few clues towards a good approach. One of these latter, nastier problems arises in the design of communications networks: the Optimal Communication Spanning Tree Problem (OCSTP). First posed by T. C. Hu in 1974, this problem has been shown to be in the family of NP-complete problems. So far, a good, general-purpose approximation algorithm for it has proven elusive.

A totally different approach to hard problems like these was needed because of their lack of structure and locality, and in 1975, John Holland [22] provided one: the Genetic Algorithm. He likened this new search technique to the search for more highly "fit" organisms that is effectively carried out in nature through the processes of evolution. Processes like "survival of the fittest", " reproduction", and "mutation" are applied to a "population" of solutions to the problem with the hope that a new, more highly "fit" solution will appear.

This thesis describes the work that was done towards the design of a genetic algorithm that would find reliably good solutions to the OCSTP. The genetic algorithm approach was thought to be an appropriate choice since it is computationally simple, provides a powerful parallel search capability, and has the ability to move around in the solution space without a dependence upon structure or locality. As an added benefit, well-designed genetic algorithms adapt very well to changes in the problem definition, without suffering changes to the genetic algorithm itself. Choosing a genetic algorithm approach spawned a secondary area of investigation: genetic algorithm researchers have never found a good scheme for efficiently representing and evolving populations of solutions to tree problems. Because of this fact, the design of such a representation became a major part of this research.

Thus, the goals of this work are to (1) design a good representation for trees in a genetic algorithm, (2) design a genetic algorithm to find reliably good solutions to the OCSTP, while retaining its ability to adapt to changes in the problem definition. The first of these will enable the use of genetic algorithms on a host of new problems relating to trees. The second goal will serve a dual purpose. First, it will provide a reliable source of good, if not optimal, solutions to tree problems. This can be quite useful in verifying the "goodness" of solutions produced by existing and new heuristics when some of the problem parameters change (e.g., new communications traffic patterns, new tariff structures, etc.). Second, it will provide examples of the kinds of trees that as yet undefined heuristics should strive to produce. Without such examples, the designer must rely upon intuition and insights into the problem as guidelines toward a good heuristic. Then, the designer will still need something to compare against, which could be the genetic algorithm.

The following chapters describe the OCSTP and its history in much more detail, explain the genetic algorithm technique, discuss the past difficulties in using a genetic algorithm to solve tree-related problems, and this thesis' proposed solution. Then, in chapter 5, the results from trials of the proposed genetic algorithm are presented. The results show that the genetic algorithm reliably produced results as good or better than other approaches. The results further show that it adapted extremely well to changes in the problem description ranging from varying traffic requirements to producing solutions to an entirely different tree-related network design problem. Central to these successes was the innovative technique for using genetic algorithms on tree-related problems which will open up new areas of research on the application of genetic algorithms to other tree-based problems. The thesis concludes with a summary of the contributions of this work and a discussion of areas for future study.

Chapter 2

The OCST Problem

2.1 Background

The OCST problem is a member of a hierarchy of problems that begins with the Network Design Problem (NDP) [26]. This hierarchy is shown in figure 2.1. In order to better understand the problems in this hierarchy, an explanation should begin with the most general form of these problems. The NDP may be described as follows:

> NETWORK DESIGN PROBLEM (NDP): You are to build a telecommunications network that will continuously interconnect a group of N towns. The simplest approach to this problem would be to install telecommunication links between all pairs of towns. However, this is too expensive, so your job is to find a less expensive alternative. Using a graph analogy to describe the network, we will call the completely interconnected network described above a *graph*, G, and the set of all possible town-to-town telecommunications links the *edges*, E. Finally, call the set of towns the vertices, V. There are two kinds of costs involved: installation and communication costs. The installation of a link, (i, j), between the towns i and j incurs a one-time charge of $L_{i,j}$. For building your network you are given a building budget, B, from which you must pay for the links you choose to install. The cost of ongoing communications

> between one town and another is computed as the product of the amount of bandwidth required between the two towns, $R_{i,j}$, and the symmetrical cost per unit of capacity of each link, $C_{i,j}$, along the shortest path between them. It is assumed that all of the $R_{i,j}$ and $C_{i,j}$ have values ≥ 0. All of the towns must be able to communicate with all of the others so this can get expensive. Your goal is to find a subgraph, G', over a subset, E', of the possible links, that has the minimum overall cost.

Expressed mathematically, the NDP objective function is

$$\min_{G'} \sum_{s,d} \left(\sum_{(i,j) \in SP_{s,d}(G')} R_{s,d} C_{i,j} \right) \tag{2.1}$$

subject to

$$\sum_{(i,j) \in G'} L_{i,j} \leq B$$

where G' is the set of all subgraphs of G and $SP_{s,d}(G')$ is the shortest path from s to d in a particular G'.

In figure 2.1, the circles represent different problems that have been proven to be NP-complete (shaded) or solvable in polynomial time (unshaded). All of these problems share the same basic structure as the NDP. Within a circle, the letter **G** or **T** indicates whether a solution of the problem can be a graph or is constrained to be a tree. This goal can be determined by the value of the budget, B. For example, if all the $L_{i,j} = 1$ and $B = N - 1$, any valid solution would have to be a tree. Below the **G** or **T**, the designation following **r** or **c** refers to the range of values that the requirements or costs, respectively, may have: **any** means there is no restriction on the range of values (other than ≥ 0) and **equal** means all of the values are the same (also ≥ 0). For example, since solutions for the NDP are graphs and the $R_{i,j}$ and $C_{i,j}$ may take on any values, the NDP circle contains **(G, r any, c any)**. From the NDP many other problems can be derived. The "single source" problem is a solvable derivation of NDP because in it there is only one source, s, for which equation 2.1 must be minimized.

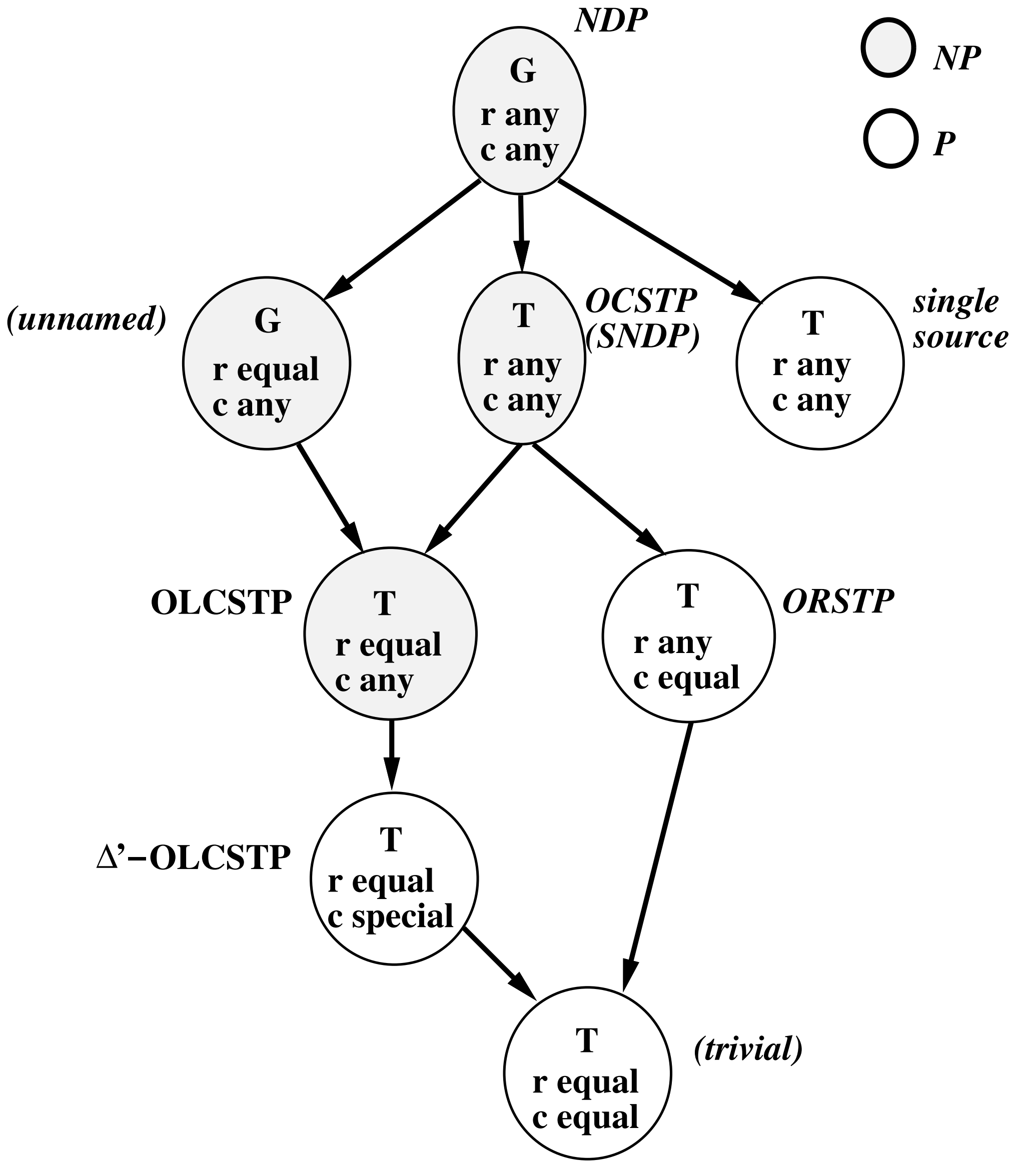


Figure 2.1: NDP Problem Derivation Hierarchy.

The work by Johnson et al. [26] includes a proof that NDP is NP-complete by demonstrating that the KNAPSACK NP-complete problem [11] is reducible to NDP. If the budget constraint is reduced to $B = n - 1$, all solutions must be tree networks. Leaving **r** and **c** set to **any** and limiting the solutions to trees in this manner moves the focus down the hierarchy from the NDP to the Optimal Communication Spanning Tree Problem (OCSTP) posed by Hu [23] (also known as the Simple Network Design Problem in [26]):

> Optimal Communication Spanning Tree Problem (OCSTP): Given a set of N cities, $n_0, n_1, \ldots, n_N$, the cost per unit of capacity, $C_{i,j}$, of a communication link between each pair of cities, and a set of requirements, $R_{i,j}$, representing the bandwidth requirement between all pairs of cities, what spanning tree connecting these cities will handle all of the traffic requirements most economically?

Expressed mathematically, the OCSTP objective function is

$$\min_{T} \sum_{s,d} \left(\sum_{(i,j) \in P_{s,d}(T)} R_{s,d} C_{i,j} \right) \tag{2.2}$$

where T is the set of all trees in the graph defined by the $C_{i,j}$ and $P_{s,d}(T)$ is the unique path between two nodes in T. Example solutions to a simple OCSTP are shown in figure 2.2. The tree requirement may be met using the trailing term in equation 2.1 in either of two ways: by setting all of the $L_{i,j} = 1$ and using a budget $B = n - 1$, or by setting all the $L_{i,j}$ to very large numbers (with respect to the $C_{i,j}$) and ignoring the budget restriction. Either of these will cause the minimization to drive toward a tree solution. Johnson et al. proved that OCSTP is NP-complete by first showing how an instance of the EXACT 3-COVER (X3C) problem could be reduced to an instance of the OCSTP and then proving that the X3C instance has a solution if and only if the OCSTP instance does.

Hu chose not to address the "general communications spanning tree" problem in his paper, choosing instead to address two subproblems derived from the OCSTP as shown in figure 2.1.

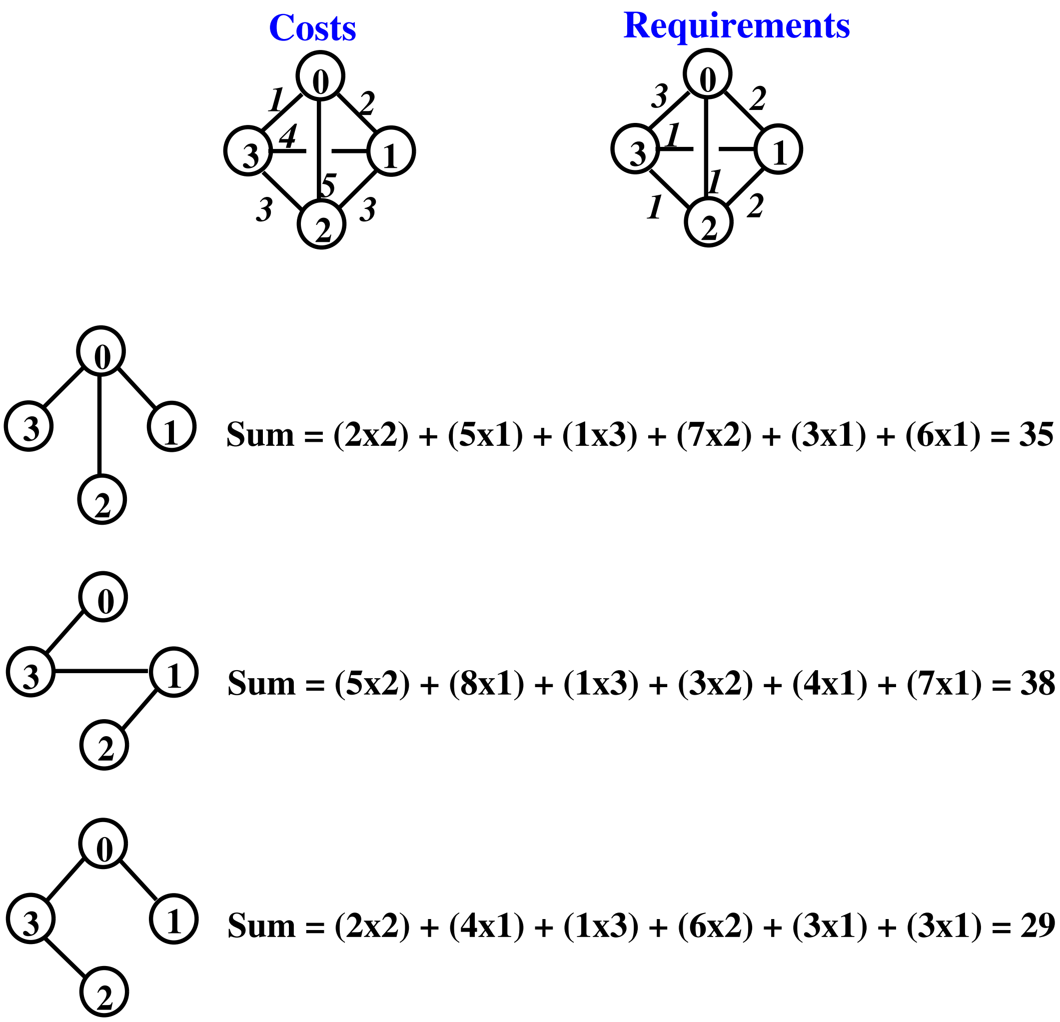


Figure 2.2: Example solutions to a simple OCSTP.

Subproblem 1: When all the requirements are equal and the link costs are arbitrary, this is the Optimum Link Cost Spanning Tree Problem (OLCSTP). This problem is still NP-complete except for the two cases for which Hu provided a polynomial time solution [11]:

Case 1: When all the link costs are equal, then the optimum link cost spanning tree is a star around the city with the highest requirements.

Case 2: Let a, b, and c represent the link costs between any three cities that form a triangle in the n-node network ($n \geq 4$), where $a \leq b \leq c$. If there exists some positive number $t \leq \frac{n-2}{2n-1}$ such that $a + tb \geq c$ for all triangles in the network, then the optimum link cost spanning tree is a star. This subproblem is called the Δ'-OLCSTP.

Subproblem 2: When all the link costs are equal and the requirements are arbitrary, the problem may be solved in polynomial time using the Gomory-Hu spanning tree algorithm [16][23]. Hu called this the Optimum Requirement Spanning Tree Problem (ORSTP).

In the general case, the OCSTP is NP-complete [11].

Other problems can be derived from these as shown in figure 2.1. The unnamed NP-complete problem must be NP-complete if it is derived from the NP-complete NDP, and the NP-complete OLCSTP is, in turn, derived from it. Finally, the trivial problem at the bottom is derived from two polynomial time problems and so it too must be a polynomial time problem. It is, in fact, a trivial problem since any star over its nodes would be minimal.

2.2 Heuristic and Exact Approaches

The NDP may be found at the heart of many design problems. Some of these include the design of transportation networks, gas pipelines, VLSI layouts, and communications networks. Several researchers have provided algorithms that perform quite well for large mesh (or, graph) networks. Hu provided the algorithms described earlier,

n	graph budget %	tree budget %
7	30-60	29
8	30-60	25
9	30-60	22
10	30-60	20
20	58-65	10
29	73-80	7

Table 2.1: Dionne and Florian algorithm budgets for their graphs and budgets they would have had to use to obtain trees.

but they are limited to special cases. He did make the observation that a big part of the problem is the selection of the interior nodes. This observation proved to be a key insight toward the development of the approach proposed by this thesis.

Dionne and Florian [7] and Lin [30] studied the general problem at length and provided heuristic algorithms that required little computation time and that produced solutions that were either optimal or nearly so. However, they made some assumptions that are incompatible with the OCSTP. They used three sets of test networks for their experiments. The first set was for a small number of nodes ranging from 7 to 9, the second set had 10 nodes, and the third had 20 and 29 nodes. The first incompatibility was that for the first and third network sets, their experiments began with less than a complete graph of links. In particular, the third set used links that represented an interurban road network, thereby eliminating most of the longer links and greatly reducing the difficulty posed by the problem. The OCSTP must begin with a complete graph. The second incompatibility was that their budget constraint for all three network sets allowed more than the $n-1$ link limit for trees. While this approach works well for the general problem, it begins to break down for the OCSTP as n increases. For example, in order to use their algorithm to find a tree, several changes must be made. First, the L_{ij} would all have to equal 1 and the budget would have to be reduced to

$$\frac{n-1}{\binom{n}{2}} = \frac{2}{n}$$

For comparison, table 2.1 contains the budgets they used and the budgets that would be necessary to limit the algorithm's output to trees. In addition, the only restriction on the l_{ij} in the networks used to test their algorithms was $l_{ij} > 0$. For the OCSTP, all of the $\frac{n(n-1)}{2}$ links in the complete graph would have to be considered, all the l_{ij} would be set to 1, and the budget set to $n-1$, all of which would greatly increase the problem's difficulty. Further, simply setting the budget in this way would still not guarantee that the $n-1$ links found would, in fact, form a tree. Further checks to verify tree-ness would be necessary. In their paper, Dionne and Florian wrote that "... (their algorithm's) computational time has a tendency to increase for very large problems with low budget levels." This fact appears to be backed up by the data they presented there. As a comparison, the size of OCSTP problems that the genetic algorithm was able to handle had five times as many nodes, and eighty-eight times as many links as their algorithm did for general graph solutions.

Their algorithm considers the problem as a series of decisions as to whether a link should be included or excluded. This scheme works well for the general problem because the topology is not constrained. The effectivness of these algorithms is adversely affected by the tree constraint of the OCSTP because good solutions are not found in neighborhoods. This means that the way these algorithms move from one solution to another, changing one link (or two links in the case of a link exchange) at a time, they will not always be able to move far enough to reach another good solution. To do so would require changing several links at once, perhaps moving groups of links from one node to another. Identifying which group of links to move is very difficult.

Gavish [12] applied Lagrangian relaxation in order to find good solutions to the general problem. These approaches work very well when the fixed cost of installing links is low to moderate, with respect to the cost per unit capacity of the links. However, as the fixed cost begins to dominate, the approach tends to break down. This can be attributed to the relaxation essentially linearizing the fixed cost. This, in turn, leads to a loosening of the bounds provided by the relaxation as the fixed cost increases. This loosening worsens as the optimization drives towards solutions with fewer and fewer edges, and becomes acute in the case of tree solutions.

This discussion described the inherent difficulties of the OCSTP and why effective approaches to it have been elusive. Both heuristic and exact techniques run into the problem's lack of "locality" wherein good solutions tend not to occur in neighborhoods. What is needed is a more "global" approach. To visualize what this approach might be, one has only to think of how a human would approach such a problem. To the trained eye, good candidates for groups of links to move or which nodes should be interior nodes might be obvious. However, codifying the knowledge in an algorithm that would lead to such an observation would be difficult. What is needed is some other approach that can move easily from one point in the solution space to another that is significantly different. Genetic algorithms [22] have this capability and have been successfully applied to many optimization problems in the past [15]. The next chapter discusses the genetic algorithm technique, and the succeeding chapters describe how genetic algorithms were successfully applied to the OCSTP and other related problems.

Chapter 3

Genetic Algorithm Technique

The genetic algorithm technique has been successfully applied in numerous problem areas, including combinatorial optimization problems. Problems exhibiting little, if any, locality are typically good candidates for a genetic algorithm. Thus, this thesis examines how to employ a genetic algorithm (GA) to search for solutions to the OCSTP. The goal is not to outperform or to find better solutions than traditional approaches. Rather, the goal is to show that genetic algorithms may be effectively applied to this problem, that they can reliably find good, if not optimal, solutions even when faced with changes to the problem, and to describe in detail how to do so. In this chapter, the genetic algorithm technique is described in detail, giving particular attention to areas that are important to the problem at hand. These discussions are followed by an example which demonstrates many of the strengths of the genetic algorithm approach.

3.1 Motivation

Algorithms for the OCSTP might employ heuristics to find good solutions. However, designing a good heuristic depends upon having good insight into a problem. For example, one might use the heuristic "Star structures are good starts" for solutions to the OCSTP. However, this heuristic can be used only so long as the problem definition remains the same. When any aspect of the original problem is changed, the heuristic must be reexamined. The heuristic may be ineffective, or even unstable, when faced with the new problem.

One of the key advantages of the genetic algorithm approach is that genetic algorithms do not rely upon specific knowledge of the problem definition. The genetic algorithm approach performs a purely non-heuristic search through the solution space. Only the fitness function needs to apply information specific to the problem. This information is usually in the form of data such as relative link costs or in the form of algorithms like a shortest-path algorithm. Since the fitness function is really a part of the input to the general-purpose genetic algorithm process, these data and algorithms are not the target of the optimization.

3.2 Genetic Algorithm Background

The genetic algorithm technique was invented by Holland [22] in the early 1970's and has been successfully applied to numerous combinatorial search space problems. Cox et al. [3] used a genetic algorithm to address a combinatorial system control problem of dynamic anticipatory routing in circuit-switched telecommunications networks. Whitley et al. [39] applied genetic algorithms to the traveling salesman and sequence scheduling problems. To address the problem of producing language-to-keyboard mappings for East Asian languages, Glover [14] also chose to use a genetic algorithm. Several other examples may be found in [15, 4, 10]. The success of genetic algorithms is due to the fact that genetic algorithms are computationally simple while providing a powerful parallel search capability. Genetic algorithms are not affected by a problem's lack of "locality" because, as will be explained later, their search takes place simultaneously throughout the solution space without regard for discontinuities or other complications.

The genetic algorithm is an attempt to emulate the process of natural selection in a search procedure. In nature, organisms have certain characteristics that affect their ability to survive and reproduce. In all but the simplest of organisms, these characteristics are represented by long strings of information contained in the chromosomes of the organism. In sexual reproduction, the offspring's chromosomes will consist of a combination of the chromosomal information from each parent[1]. The process of natural selection provides that the more "fit" individuals will have the opportunity to mate most of the time. This in turn leads to the expectation that the offspring stand a good chance of being similarly highly fit.

Occasionally, mutations can arise which can make striking changes to the characteristics of an individual. Sometimes these random changes will be lethal, in that the individual may no longer be fit enough to survive or reproduce. Other times, the changes may actually improve the fitness of the individual, thereby improving its chances, or the changes may have no effect at all. Without mutation, the population tends to converge to a homogeneous state wherein individuals vary only slightly from each other.

One can think of the evolution via natural selection of a randomly chosen population of individuals, or more specifically, chromosomes, as a search through the space of possible chromosomal values. In addition, that search is being done in parallel – each generation provides for the mating of the individuals in the population, resulting in a new, probably stronger population. It is these features of natural selection and parallelism that make genetic algorithms appealing for large search problems that exhibit little or no locality.

3.3 Genetic Algorithm Components

As described in Davis' book [4], a genetic algorithm has five components:

1. a means of encoding solutions to the problem as a chromosome
2. a function that evaluates the "fitness" of a solution
3. a means of obtaining an initial population of solutions
4. reproduction operators for the encoded solutions
5. appropriate settings for the genetic algorithm control parameters

[1]As opposed to asexual reproduction wherein only one parent and one set of chromosomes is available.

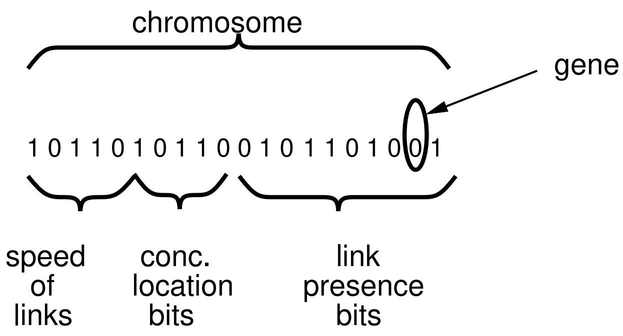


Figure 3.1: Example chromosome.

3.3.1 The chromosome

The first, and perhaps the most important, step in applying a genetic algorithm to a problem is to choose a way to represent a solution to the problem as a finite-length string over a finite alphabet. These strings are referred to as chromosomes. For example, in a concentrator network design problem, the chromosome might be a string of binary digits representing different aspects of a design. Figure 3.1 shows an example for a five-node network where the chromosome contains three parts:

- a 5-bit number representing the speed of all the links;
- a 5 bit series, one bit per node, that specifies whether or not that node is a concentrator;
- a 10 bit series, one bit per possible link, that specifies whether or not that link is present

The values on the chromosome may be arranged and interpreted as needed. They may represent boolean values, integers, or even discretized real numbers. For example, the five bits for link speed might be used to select a speed from a list, or it may be a representation of the actual speed itself. If the need arises to encode a real-valued parameter, the encoding must discretize the number for representation as a finite string.

The choice of how to encode solutions on a chromosome is of primary importance to the success of the genetic algorithm approach to a problem. The encoding of information on the chromosome should be *right for the problem* rather than specific to the problem. The encoding should be able to represent all of the truly relevant parameters of the problem and should avoid other parameters. Using parameters that aren't directly relevant will cause the genetic algorithm to be subject to changes in the problem that would not otherwise effect it, thereby making it no more useful than a specialized heuristic. Some knowledge of the search space is, of course, unavoidable. Rawlins [36] observed

> ... if an algorithm is to be more effective than random search, then "prior knowledge" must be included in the choice of encoding. Just as familiar numerical hillclimbing techniques, if they are to perform well, require knowledge to be contained in the usual binary or floating point encodings in the form of smoothness, differentiability, etc., so too do genetic algorithms require some sort of knowledge to be built into the objective function.

This built-in knowledge has a minimal impact on the genetic algorithm since this kind of information changes much more slowly than the information required for a heuristic. For example, the fitness function that was used in a genetic algorithm used to position cases of beer on pallets and then to order the pallets in delivery trucks [27] (yet another combinatorial optimization problem) made use of knowledge of the truck's interior dimensions. While it is true that the distributor may eventually buy new trucks, that sort of change is expected to occur less often than, say, the brands and corresponding shapes and sizes of cases and pallets that are to be delivered.

So, while the genetic algorithm approach specifically avoids the use of any heuristics (like requiring smoothness of the search space) for finding solutions to a given optimization problem, genetic algorithms must retain enough search-space specific information so as to allow the objective function to operate efficiently. Care must be taken to ensure that the selected encoding can uniformly represent all the possible solutions to the problem, regardless of any prior expectation of their viability. All sorts of genetic material must be possible in the population since good solutions may result from the mixing of one or more inferior ones.

Another key to understanding what makes a good encoding is a *schema*. The concept of a schema [21, 22] is the basis of genetic algorithm theory. Holland described a schema as a template describing a subset of strings in a population with similarities at certain string positions. For example, *101* and *100* are identical when the rightmost position is ignored. If $*$ is the "don't care" symbol then these two strings may be represented by the schema *10*$*$. The genetic algorithm search process has only a measure of the fitness of a chromosome to use as a guide. What other information can be gleaned from the population? If a person looks at a series of strings and their fitness values, similarities between the strings will become apparent. For example, the strings and fitnesses may seem to imply that strings starting with *10* are better than the others (at least in a given population). These similarity templates are the schemata.

Goldberg offered two basic principles for choosing a genetic algorithm encoding [15]. The first addressed the kinds of symbols used to represent the information contained in the chromosome:

> *Principle of Minimal Alphabets*: select the smallest alphabet that permits a natural expression of the problem.

This would favor the use of a binary encoding, for example, over an alphabet-based encoding wherein a set of several symbols is used. Binary encodings maximize the number of schemata available to the genetic algorithm's search process [15]. If each position on a binary-encoded chromosome of length L can have schema values *0*, *1*, and $*$, then there are 3^L possible schemata. If a different alphabet were used, say one with 4 symbols a, b, c, d, and $*$, then the length required to encode the same number of values would be a smaller number L', resulting in $5^{L'}$ schemata. It can then be easily shown that the binary alphabet will provide more schemata than any other coding. For example, using this four symbol alphabet to encode the integers $[0, 15]$ results in a length $L' = 2$ which results in a total of $5^2 = 25$ schemata. The binary encoding for the same numbers would have $L = 4$ and would result in $3^4 = 81$

schemata. In addition, it is clearly harder to identify the similarities between the 4-ary values than it is to identify the similarities between the binary values:

0110 0111 1110 1111

bc *bd* *cc* *cd*

Goldberg's second principle gave recommendations for the size of these schemata:

> *Principle of Meaningful Building Blocks*: select an encoding so that short, low-order (i.e. having a small number of symbols) schemata are relevant to the underlying problem and relatively unrelated to schemata over other fixed positions.

This points out that when a series of bits of information on the chromosome are closely related to each other, they should be few in number and located as near to each other as possible. These goals reduce their chance of being separated by the mating process, or *crossover*, wherein two chromosomes exchange portions of their genetic material. In a schema, each of the positions of the chromosome is specified as being a *0*, *1*, or a $*$. The length of a schema, $l\,(schema)$, is defined as the distance from the first to the last fixed position in the schema. The order of a schema, $o\,(schema)$, is simply the number of fixed positions it specifies. For example, the 8-bit schema $< * * \mathit{101} * \mathit{1} * >$ has a length of five and an order of 4. Since mutation is defined in terms of the probability, p_m, of each position being changed after crossover, the overall probability of a schema surviving mutation intact is $(1 - p_m)^{o\,(schema)}$.

While strict adherence to this second principle can be quite difficult, it should be respected as much as possible. If it is not followed at all, the performance of a genetic algorithm's search can be so degraded as to be similar to that of a random search. In our concentrator design example, the chromosome consists of 20 bits, and the bits that define link speed are all together at one end. The probability that this group of five bits will be split during crossover is only

$$\frac{l\ (schema) - 1}{L - 1} = \frac{4}{19}$$

where L is the overall length of the chromosome. If instead, the five bits were arranged as two bits on one end and three on the other, this probability would increase to

$$\frac{l\ (schema) - 1}{L - 1} = \frac{19}{19} = 1$$

This would result in almost guaranteed disruption[2] of the link speed schema in every generation, resulting in these bits having much less impact on the overall fitness of chromosome.

The concept of schemata also explains the parallel search nature of genetic algorithms. Recall that given a binary-coded chromosome of length L there are 3^L possible schemata. A specific chromosome represents 2^L schemata because each position will match a schema wherein it takes on the value in the chromosome or $*$. In a population of size n, there are between 2^L (all the chromosomes are identical) and $n2^L$ (all are different) schemata represented. It was shown in Holland's [22] and Goldberg's [15] works that despite the processing of only n chromosomes during each generation, the genetic algorithm actually processes $O(n^3)$ schemata each generation. Goldberg described the derivation of this value in detail. First he estimated the total number of schemata in a string to be $2^{(l(schema)-1)} \times (L - l\,(schema) + 1)$. Calling this an overestimate since there would undoubtedly be duplicates of low-order schemata in larger populations, he refined the estimate by choosing a population size of $2^{l(schema)/2}$. Goldberg then observed that the number of schemata is binomially distributed, one-half having higher and one-half having lower order than this. Choosing to count only the higher order schemata, Goldberg derived the $O(n^3)$ estimate given above. This apparently unique advantage of genetic algorithms processing only n chromosomes but actually searching $O(n^3)$ schemata was named *implicit parallelism* by Holland.

3.3.2 The fitness function

A function is needed that will interpret the chromosome and produce an evaluation of the chromosome's fitness. This function must be defined over the set of possible chromosomes and is assumed to return some non-negative value representing the fitness. The definition of this function is crucial because it must accurately measure the desirability of the features described by the chromosome. In addition, the function must make this evaluation in a very efficient manner due to the large number of times the function will be called during the execution of the genetic algorithm. For example,

[2] Disruption is only *almost* guaranteed because crossover itself will only occur with some probability specified as one of the Genetic Algorithm control parameters.

with a population of 100 chromosomes that runs for $1,000$ generations, there could be as many as $100,000$ calls to this evaluation function during execution. For the case when exact methods are too expensive, the use of approximation functions for the fitness were investigated by Grefenstette and Fitzpatrick [20]. They were studying the use of a genetic algorithm to perform image registration as part of a digital subtraction angiography system. In this process, two 100x100 pixel x-ray images must be accurately aligned, or registered, with each other and then the pixels of one image "subtracted" from the other to produce an image of what changed between the two x-rays. A full evaluation of their fitness function would have required $10,000$ transformations and image difference equations to produce the image. Realizing that this would be prohibitively slow, they performed experiments to see if a sampling of the images might perform acceptably well. They found that they could get excellent results with a sample size of only 10 of the above $10,000$ calculations. This ability of genetic algorithm to tolerate a noisy fitness function is explained by genetic algorithm theory in that it is the schemata, not the individual chromosomes that are driving the genetic algorithm's search.

3.3.3 Reproduction operators

These operators perform the selection, crossover (mating), and mutation of chromosomes. These operators actually manipulate the individual chromosomes and must, therefore, be written with the underlying encoding of the chromosome in mind. For example, when defining an operator such as crossover, which will actually perform the mating or "crossing-over" of two chromosomes, care must be taken to balance the mixing of gene values with producing viable offspring and/or the cost of repairing non-viable offspring[3]. Crossover doesn't always occur, rather it does so with some probability p_c. If two chromosomes are selected for mating, with probability $1 - p_c$ they will not undergo crossover and the selection process starts over. Again using the concentrator network design example, a simple crossover operator is shown in figure 3.2. In this example, a random point is chosen between two of the positions, or

[3]See comments in chapter 6 about Davis' recent work [32] in the area of chromosomal repair.

crossover here:

10110 10110 01 011 01001

01010 10001 01 101 11101

results in:

10110 10111 01101 11101

01010 10000 01011 01001

which may be mutated into:

10110 10111 01101 11101

01000 10000 01011 01001

Figure 3.2: Example of crossover.

genes, of the two parent chromosomes. Each of the chromosomes is cut at that point and the two ends are exchanged. This typically results in two different chromosomes with different characteristics. After crossover is completed, each position in the new chromosomes will be changed, or mutated, with some typically small probability, p_m, to another value. For the binary alphabet case, this would simply be changing the bit value from a 0 to a 1 or vice versa. This mutation operator is employed to ensure, with some probability, that selection and crossover don't lose potentially useful genetic material. The new chromosomes resulting from the selection and application of the crossover and mutation operators are the "offspring" of this mating and will be evaluated and added to the population of the next generation.

3.3.4 Choosing an initial population

In a "pure" genetic algorithm, the initial population is chosen randomly, with the goal of selecting chromosomes from all over the search space. Whatever genetic material is in the initial population will be the only material, except for the rare changes due to mutation, available to the genetic algorithm during its search. One might employ a heuristic to choose the initial population in an attempt to introduce the "right" genetic building blocks into the population. However, this can lead to problems since genetic algorithms are "notoriously opportunistic" [18]. The presence of just a few chromosomes with fitnesses far better than all the others may cause genetic algorithms to converge prematurely to a local optimum.

3.3.5 Genetic algorithm control parameters

There are other parameters that govern the genetic algorithm search process. Some of these are:

Population size determines how many chromosomes are available, and therefore how much genetic material is available, for use during the search. If there is too little, the search has no chance to adequately cover the space. If there is too much, the genetic algorithm wastes time evaluating excess chromosomes.

Generation Gap specifies what percentage of the population will be replaced with new offspring in each generation, with the lesser-fit chromosomes of a generation replaced first.

Generations specifies how many times a *Generation Gap* percentage of the population will be replaced through reproduction

Crossover Rate specifies the "probability" of crossover (mating) occuring between two chromosomes. This value can be thought of as "the average number of crossovers occuring in a generation." A "probability" > 1 is interpreted to mean that the chromosomes may undergo crossover more than once. For example, a value of 1.4 would be interpreted to mean "perform crossover on the parents, and then on the children with probability $p_c = 0.4$."

Mutation Rate specifies the probability that a value in the chromosome of a newly-created offspring will be randomly changed.

Elitism is a switch that specifies that after the usual selection is performed, the chromosome with the best fitness always survives intact into the next generation. Without such a guarantee it is possible that the best chromosome of a generation might be lost due to mutation, crossover, or selection. DeJong originated this idea and studied its effect on a selection of problems in his dissertation [5].

Scaling specifies the level of scaling used. Scaling is done to maintain good levels of competition throughout the search process. In the absence of scaling, a few highly-fit chromosomes could dominate the population very early in the process. To counter this effect, the fitness measure is scaled down to avoid premature convergence. Later, the population becomes more and more similar and thus the competition between chromosomes is smaller. To counter this effect, the fitness function is scaled up to magnify the remaining differences between chromosomes. The sigma-truncation technique [9] was used for this work. This technique subtracts a constant times the standard deviation of the population fitness values from the raw fitness values as follows

$$fitness_{scaled} = fitness_{raw} - (fitness_{average} - c\sigma)$$

where c is a constant usually in the range $[1, 5]$.

The values chosen for these parameters later in chapter 4.

3.4 Genetic Algorithm Search Process

Once you have the above components, the basic genetic algorithm process proceeds as follows:

1. create an initial population of solutions (chromosomes)

2. evaluate the fitness of each chromosome

3. while the population shows sufficient diversity

(a) select pairs of chromosomes, using a random selection weighted by their fitness

(b) perform crossover on the chromosomes in order to exchange information

(c) with a low probability, apply mutation to the offspring chromosomes

(d) evaluate the fitness of the offspring

(e) replace some or all of the previous population with the offspring population

3.5 Fundamental Theorem of Genetic Algorithms

To show how all these pieces fit together into an effective search mechanism, the schema theorem, also known as the fundamental theorem of genetic algorithms [15], is examined. The goal of the genetic algorithm is for each successive generation's population to have a favorable chance of containing more highly fit chromosomes. As explained earlier, the search through the schemata is the central issue. It is, therefore, not surprising that the theorem is in terms of a prediction of how a particular schema will fare from one generation to the next. If the number of instances of a schema, s, present in a population at generation g is $I(s,g)$, then an estimate of the expected number of copies of schema s in the next generation is

$$I(s,g+1) \geq \underbrace{I(s,g)}_{a}\underbrace{\frac{f(s)}{\overline{f}}}_{b}\underbrace{\left[1-\left(\frac{l(s)}{L-1}\right)p_c - o(s)\,p_m\right]}_{c} \tag{3.1}$$

where $f(s)$ is the average of the fitnesses of all the chromosomes matched by the schema s and $\overline{f}$ is the average fitness of all the chromosomes in the population. The terms L, $l(s)$, and $o(s)$ are the overall length of the chromosome, length of the schema, and order of the schema, respectively, as defined earlier. In words, the theorem says that the number of instances of a schema in the next generation may be estimated by the product of (a) how many instances there are now, (b) the ratio of the average fitness of the chromosomes of the schema to the average of the fitnesses of all the chromosomes, and (c) the probability that the schema will survive crossover and mutation. This theorem backs up the observations that above-average, short, and low-order schemata will grow in number over the generations.

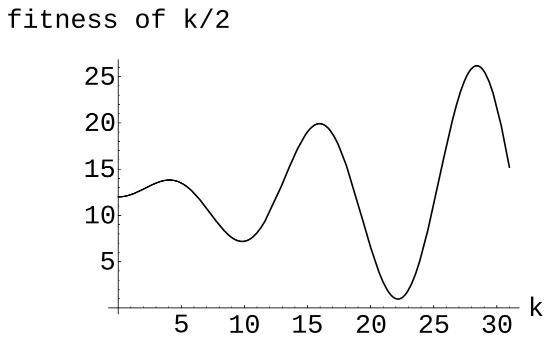


Figure 3.3: First example objective function.

3.6 Genetic Algorithms at Work

To demonstrate the inner workings of genetic algorithms, two simple ones will now be described and their operation examined in this section. The first example is a maximization problem. The function that is to be maximized is

$$x \sin x + 12.0, \text{ where } 0 \leq x < 16$$

The 12.0 is added to ensure the fitness function values are non-negative over the range. This function has two local maxima in addition to the actual maximum and is shown in figure 3.3. It is also a continuous function, but the genetic algorithm uses a discrete domain. So, the domain must be discretized. One way of doing this is to allow the chromosome to have a single value that is interpreted as

$$x = \left(\frac{k}{2}\right), \; k \in \{0, 1, ..., 31\}$$

Now the five-bit chromosome can be used directly as input to the objective function. This version of the objective function is shown in figure 3.4. For this example, a population size of 4 will be used. Figure 3.5 shows the objective function with the

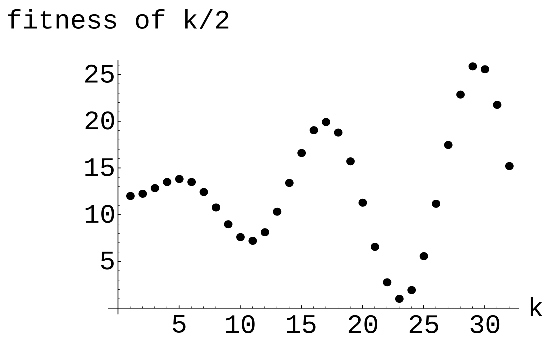


Figure 3.4: Discretized first example objective function.

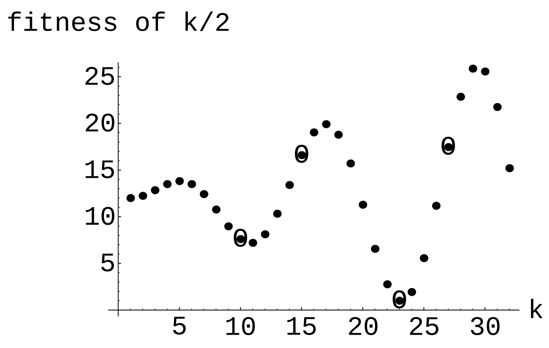


Figure 3.5: Sample GA 1: Initial population.

chromosome	**value**	f	$\frac{f}{\sum f}$	$\frac{f}{\overline{f}}$	**actual**
01001	9	6.6	17%	0.68	1
11010	26	16.5	43%	1.7	2
10110	22	0.0	0%	0.0	0
01110	14	15.6	40%	1.61	1
parent		**crossover**	**child**	**value**	f
0100|1		4	01000	8	7.97
1101|0		4	11011	27	21.9
01|110		2	01010	10	6.21
11|010		2	11110	30	20.8

Table 3.1: Example GA 1 Initial population: $\sum f = 38.66$, $\overline{f} = 9.67$.

initial population of random chromosomes indicated. Before evolution begins, this zero'th generation must be evaluated to determine the fitness of the chromosomes. The top half of table 3.1 summarizes the initial population's information. The first column holds the chromosome itself. The second holds the chromosome value. The third holds the objective function value (fitness). The fourth holds the percentage of the sum of the fitnesses this chromosome represents. The fifth holds a calculation of the expected number of times this chromosome would be selected for breeding. Once evaluated, the selection process is performed to identify mating pairs. The last column shows how many times each chromosome was selected. In the bottom half of table 3.1 the results of the first breeding are shown. The entries are shown in groups of two parents. The first column holds the parent chromosome and the second holds the crossover point. The third column holds the child chromosome and the fourth holds the child value. The last column holds the child's fitness. As the table shows, the original population member that had a fitness of zero was not selected for breeding, and so disappeared. The results of this first generation are shown in figure 3.6. The successive generations of this sample genetic algorithm are shown in tables 3.2 through 3.4 and figures 3.7 through 3.9. In the final population, the population has converged to the two values nearest the optimum. In the absence of mutation, future generations could not produce any offspring different from the set of parents. If mutation was being used, then the probability of remaining in this

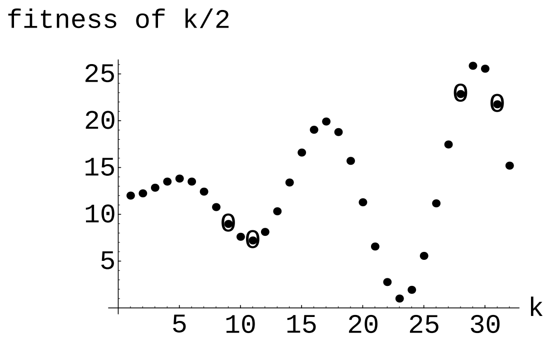


Figure 3.6: Sample GA 1: First generation.

static state, $P\{static\}$, drops from 1.0 to $(1-\mu)^{nl}$, where μ is the bitwise mutation rate, n is the size of the population, and l is the length in bits of each chromosome in the population. So, in this example a mutation rate of 0.001 would result in $P\{static\} = (0.999)^{4\times 5} = 0.98$.

The second example is a minimization problem. The function that is to be minimized is

$$1 + \sin x, \text{ where } 0 \leq x \leq 16\pi$$

One is added to $\sin x$ so that all of the function values will be non-negative. This function is shown in figure 3.10. This objective function has eight equal minima at intervals of $\frac{3\pi}{2} + k2\pi, k = 0, 1, \ldots, 7$. The hope is that the genetic algorithm will find as many of these minima as possible. This example is given to focus in on the importance of the encoding and how the wrong encoding can lead to incorrect or, as in this example, incomplete, results. The first attempt at an encoding used a single-valued chromosome that varied from 0 to 53 radians, which covers about $8\frac{1}{2}$ cycles of length 2π. The simple fitness function used was

$$fitness = 1 + \sin(chromosome)$$

chromosome	**value**	f	$\frac{f}{\sum f}$	$\frac{f}{\overline{f}}$	**actual**
01000	8	7.8	14%	0.56	1
11011	27	21.9	38%	1.50	1
01010	10	6.1	11%	0.44	0
11110	30	20.8	37%	1.46	2
parent		**crossover**	**child**	**value**	f
1|1110		1	11011	27	21.9
1|1011		1	11110	30	20.8
111|10		3	11100	28	24.9
010|00		3	01010	10	6.2

Table 3.2: Example GA 1 Second generation: $\sum f = 56.88$, $\overline{f} = 14.22$.

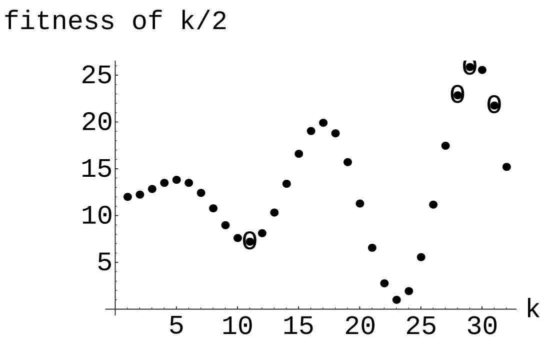


Figure 3.7: Sample GA 1: Second generation.

chromosome	**value**	f	$\frac{f}{\sum f}$	$\frac{f}{\overline{f}}$	**actual**
11011	27	21.9	30%	1.19	1
11110	30	20.8	28%	1.13	1
11100	28	24.9	34%	1.35	2
01010	10	6.2	8%	0.34	0
parent		**crossover**	**child**	**value**	f
110|11		3	11000	24	4.56
111|00		3	11111	31	14.2
11|100		2	11110	30	20.8
11|110		2	11100	28	24.9

Table 3.3: Example GA 1 Third generation: $\sum f = 73.81$, $\overline{f} = 18.45$.

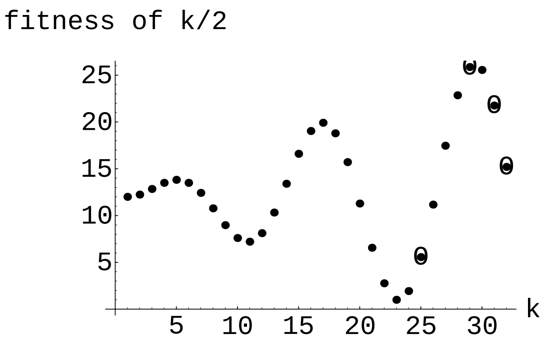


Figure 3.8: Sample GA 1: Third generation.

chromosome	**value**	f	$\frac{f}{\sum f}$	$\frac{f}{\overline{f}}$	**actual**
11000	24	4.56	7%	0.28	0
11111	31	14.2	22%	0.88	1
11110	30	20.8	32%	1.29	2
11100	28	24.9	39%	1.55	1
parent		**crossover**	**child**	**value**	f
1111|1		4	11110	30	20.8
1110|0		4	11101	29	24.6
11110			11110	30	20.8
11110			11110	30	20.8

Table 3.4: Example GA 1 Fourth generation: $\sum f = 64.46$, $\overline{f} = 16.1$.

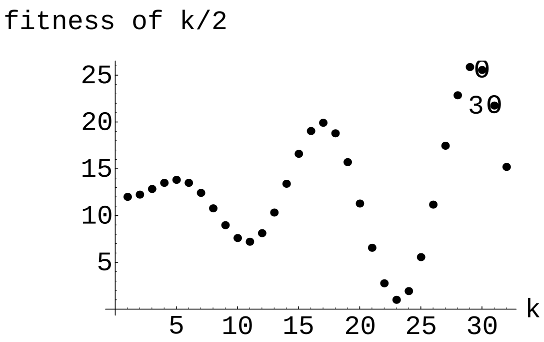


Figure 3.9: Sample GA 1: Fourth generation.

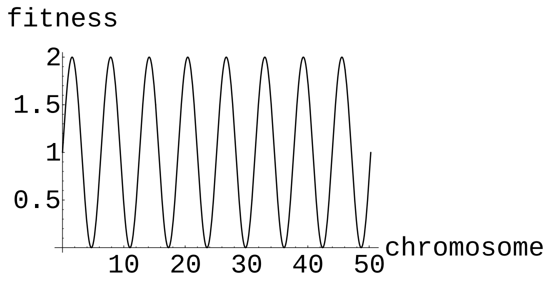


Figure 3.10: Second example objective function.

Experiments using this encoding resulted in none of the possible chromosomes actually having a fitness equal to the known minimum of 0. In fact, no two of the possible chromosomes even had the same fitness. These shortcomings allowed the genetic algorithm to find solutions near many of the minima during the run, but then forced it to finally converge to the one chromosome whose value came the closest to 0 (a chromosome with a value of 11 yielded a fitness value of 9.79×10^{-6}). In this case, the direct encoding of the objective function's argument can never be granular enough to get pairs of chromosomes with equal fitnesses.

A second experiment remedied this problem by changing the encoding to take 32 samples in each 2π cycle. This version of the function is shown in figure 3.11. Using this sampling, along with a single-value chromosome that ranged from 0 to 255, thus covering the desired range exactly, the fitness function became

$$fitness = 1 + \sin\left(chromosome \times \left(\frac{2\pi}{32}\right)\right)$$

This should result in minima at chromosomes equal to $24, 56, 88, 120, 152, 184, 216$, and 248. The genetic algorithm was run the usual five times, keeping the best twenty chromosomes found. As the results in table 3.5 show, the genetic algorithm was able,

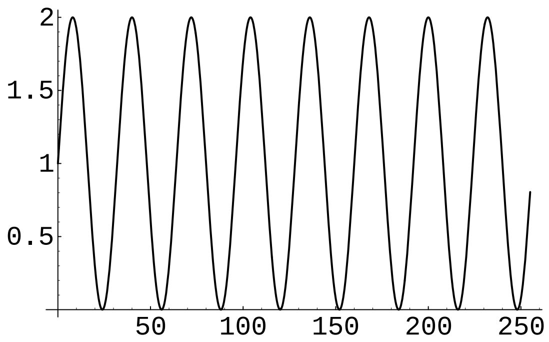


Figure 3.11: Modified second example objective function.

run/min	**48**	**112**	**176**	**240**	**304**	**368**	**432**	**496**
1	2	0	2	4	3	1	4	4
2	4	2	4	4	1	1	1	3
3	0	2	3	5	3	3	3	1
4	3	3	2	3	1	3	2	3
5	2	3	2	1	5	3	3	2

Table 3.5: GA results for minimization of $f(x) = \sin(x)$.

chromosome	**value**	**fitness**
00011000	24	0
00011000	24	0
00111000	56	0
00111000	56	0
00111000	56	0
01011000	88	0
01011000	88	0
01111000	120	0
10011000	152	0
10011000	152	0
10011000	152	0
10011000	152	0
10011000	152	0
10111000	184	0
10111000	184	0
10111000	184	0
11011000	216	0
11011000	216	0
11111000	248	0
11111000	248	0

Table 3.6: Twenty best chromosomes found during minimization of $f(x) = 1 + \sin(x)$.

on average, to find all of the minima. This is a good example of when a poor choice of encoding led to incomplete results. Another interesting aspect of this example is that the schema processing done by this genetic algorithm is particularly clear. In table 3.6 the twenty best chromosomes from the fifth experiment are shown. One schema is clearly visible in the chromosomes: <****11000*>. This schema shows that the bits in the 16 and 8 positions are always set to 1, with the bits in positions $4, 2,$ and 1 always set to 0. It should not be surprising that these particular bits are always set since this encoding forces minima at regular intervals of 24:

$$24 + k \times 32,\ k = \{0, 1, \ldots, 7\}$$

Similar patterns appeared in the other runs. Changing the genetic algorithm encoding to cover 16 cycles of 64 steps produced similar results. These results imply that a genetic algorithm may indeed find many, if not all, of the minima when there are several having equal evaluations of the fitness function. Finding multiple optima would be useful when the extension of the fitness function to include a factor discriminating amongst them would make it prohibitively expensive. Another situation where the application of a genetic algorithm to find multiple equal minima would be when these minima could be used as the starting point for a different search technique for which there exists a very-efficient algorithm.

One might argue that having prior knowledge of where the minima were guided the move to the better encoding and that in general such information would not be available. This is surely a true statement, but it misses the point of this admittedly contrived example: the choice of an encoding is probably the single most important step in the application of a genetic algorithm to a problem. As described earlier in section 3.3.1, while the genetic algorithm approach specifically avoids the use of any heuristics, they must retain enough search-space specific information so as to allow the genetic algorithm search to proceed efficiently and reliably. The developer of a genetic algorithm for a problem may have to examine several different encodings before finding the "right" one. Some encodings may perform worse than a random search, and will be abandoned. Others will produce better solutions and will merit further examination and comparison. The next chapter will describe the encoding definition process for the genetic algorithm for the OCSTP.

Chapter 4

Representing Trees in Genetic Algorithms

The approach to the OCSTP described in this thesis uses a genetic algorithm to search through the space of trees over a set of n nodes with symmetrical traffic requirements $R_{s,d}$ between nodes s and d, and symmetrical link costs $C_{i,j}$ associated with each link in the complete graph. The fitness of a solution is therefore defined as a weighted sum of the shortest paths between all pairs of nodes.

The set of n nodes, N, are labeled with the numbers 1,2, ... ,n. The (undirected) edge between nodes i and j is denoted by (i, j). All edges are assumed to be candidates for possible inclusion in the tree; i.e., the underlying graph from which the tree is formed is a complete graph. When the underlying graph for the problem is not complete, it is always possible to transform the problem into an equivalent problem[1] on a complete graph. Missing edges can be added to the original graph with sufficiently undesirable properties so as to prevent their inclusion in any solution. In the OCSTP, an example of such an undesirable property would be a very large link cost.

It is possible to give an orientation to the edges in a tree by designating one node as the root and having all edges oriented towards (or, alternatively, away from) this

[1]Solutions produced for such a transformed problem will need to be checked to ensure that they do not use the missing edges. If the missing edges were not used, the solution will be feasible for both the original and the transformed problems. If the missing edges were used, the solution will not be a feasible solution to the original problem.

node. Such trees are called rooted trees. Sometimes, the nature of the problem makes this designation significant, as in the case of a tree of shortest paths to (or from) a given node. Sometimes the problem specification fixes the identity of the root. Other times its selection is part of the problem. In all cases, it is possible to represent an arbitrary tree as a rooted tree. Rooted tree representations will be used in this thesis.

It has been shown by Cayley [31] and others [13] that the number of possible trees in a complete graph on n nodes is $n^{(n-2)}$. Since each such tree can correspond to n possible rooted trees, with any node designated as the root, there are $n^{(n-1)}$ possible rooted trees. The efficiency of any representation can be measured by comparing the number of graphs that can be represented by it to the possible number of trees. If the former is much larger than the latter, many non-trees might also be represented and this is, as was mentioned earlier, a problem.

4.1 Tree Encoding Issues

By far the most important part of designing a genetic algorithm to find solutions to a given problem is deciding what information should be stored in the chromosomes and how it should be encoded there. For problems that have trees as their solutions, an effective encoding should possess the following properties:

1. It should be capable of representing all possible trees.

2. It should be unbiased in the sense that all trees are equally represented; i.e., all trees should be represented by the same number of encodings. This property allows us to effectively select an unbiased starting population for the GA and gives the GA a fair chance of reaching all parts of the solution space.

3. It should be capable of representing only trees. To the extent that non-trees can be represented, it becomes more difficult to generate a random initial population for the GA. Worse yet, it becomes possible for crossover and mutation to produce non-trees from valid parent trees.

4. It should be easy to go back and forth between the encoded representation of the tree and the tree's representation in a more conventional form suitable for evaluating the fitness function and constraints.

5. It should encourage short, low order schemata so as to help the population evolve towards more fit chromosomes. As explained in section 3.3.1, long schemata cause genetic algorithms to drift.

6. It should possess locality in the sense that small changes in the representation make small changes in the tree. This allows the GA to function properly by having the encoding truly represent the tree. Thus, when crossover takes place, parts of the parent trees are inherited and good traits can propagate from one generation to the next. Without this property, the GA tends to drift rather than converge to a highly fit population.

Ideally the representation of a tree for a GA should have all of these properties. Unfortunately, most representations trade some of these desirable traits for others. In the next sections, the traditional and some new tree representations are described and evaluated on the basis of how well they meet the aforementioned criteria. The last of these, the biased-encoding, is described and it is shown to perform quite well for the OCSTP and meets all but the second criterion.

4.2 Traditional Tree Representations

4.2.1 Characteristic vector

If one associates an index k with each link (i, j), a tree T can be represented as a vector $E = e_k$, $k = 1, 2, \ldots K$, where K is the number of edges in the underlying graph and e_k is one if edge k is part of T and zero otherwise.

In a complete graph, $K = \frac{n(n-1)}{2}$. There are thus $2^{\frac{n(n-1)}{2}}$ possible values for E and, unfortunately, most of these are not trees. Indeed, since all trees have exactly $n-1$ edges, if E contains other than $n-1$ ones it is not a tree. Even if E contains exactly $n-1$ ones, it is unlikely that it represents a tree. Indeed, the probability of a random E being the representation of a tree is infinitesimally small as n increases. It is, in fact, of order

$$\frac{2^{\binom{n}{2}}\ encodings}{n^{n-2}\ trees} \approx O\left(2^{-\left[n\left(\frac{n}{2}-\log 2(n)\right)\right]}\right)$$

Thus, if random vectors were generated in order to provide a starting population for a genetic algorithm, it is quite likely that none of them would be trees. Furthermore, when any two trees are mated in the course of a genetic algorithm, it is quite likely that neither of the offspring would be trees.

It is an $O(n^2)$ effort to go back and forth between this encoding and a tree. This is not very good, since there are other methods discussed later where only $O(n)$ effort is required.

On the positive side, all trees can be represented by such vectors, all are represented equally (once), and the representation does possess a natural locality; changing a bit in the vector adds or deletes a single edge. On the whole, however, this is a poor representation for a GA because of the extremely low probability of obtaining a tree.

4.2.2 Predecessors

An alternative representation is to designate a root, r, for the tree and then record the predecessor of each node in the tree rooted at r. Let $Pred[i] = j$ where j is the first node in the path from i to r in T. As a convention, set $Pred[r] = r$. Thus, every rooted tree T is represented by a unique n digit number, where the digits are numbers between 1 and n. For example, in figure 4.1 the vectors represent the trees shown. In other words, every (unrooted) tree is represented by n of these n digit numbers since each of the n digits gets a turn as the root. Therefore, this encoding is unbiased and covers the space of solutions.

There are n^n such n digit numbers. Since there are $n^{(n-1)}$ rooted trees, a random number of this type represents a tree with probability $\frac{n^{n-1}}{n^n} = \frac{1}{n}$. This is a great improvement over the characteristic vector, but still allows for many non-trees being generated both in the initial population and during the breeding which takes place

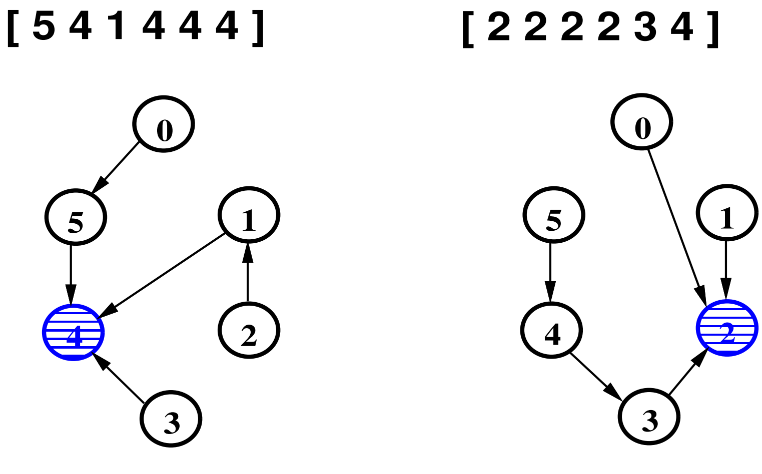


Figure 4.1: Predecessor vectors and their trees.

during the course of the GA. For example, if a crossover operator was applied to the two predecessor vectors in the top of figure 4.2 at the fourth position, the resulting predecessor vectors would not represent trees, as shown in the bottom of figure 4.2. Even converting one predecessor vector so that it has the same root as the other, non-trees are still quite possible outcomes as shown in figure 4.3. Due to the strong possibility of non-trees resulting from breeding, this encoding is not directly useful.

Given a matrix mapping node pairs into edges, which can be set up at the start of the GA and requires $O(n^2)$ space, the transformation back and forth between this representation and a list of edges requires only $O(n)$ steps. This is also an improvement over the characteristic vector representation. Thus, at least for complete graphs, this representation is significantly better than the characteristic vector. For sparse graphs, such as trees, however, the improvement diminishes.

4.2.3 Prüfer numbers

A third possible encoding is the Prüfer number [31] associated with a tree, defined as follows. Let T be a tree on n nodes. The Prüfer number, $P(T)$, is an $n-2$ digit number, where once again the digits are numbers between 1 and n , $n \geq 3$, and are

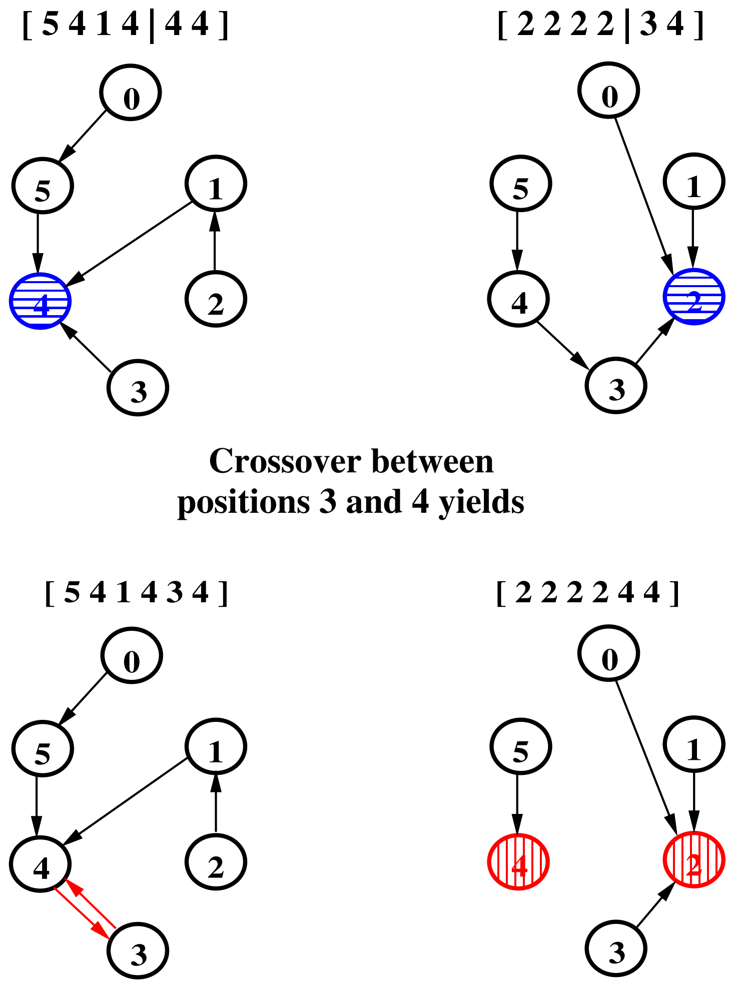


Figure 4.2: Example of simple breeding of predecessor vectors producing bad children.

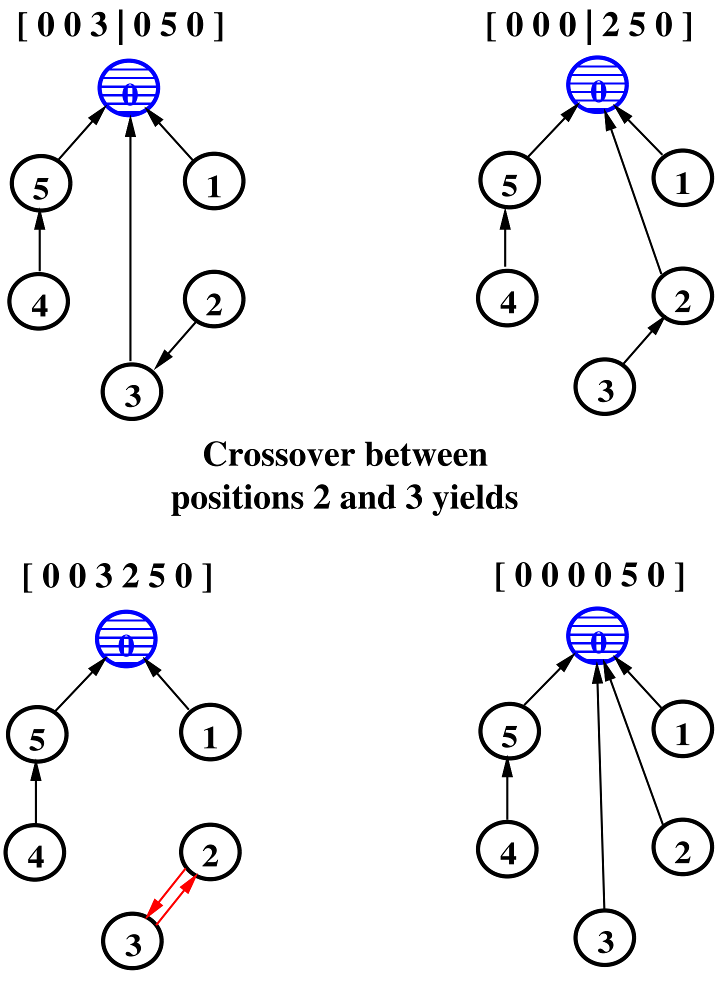


Figure 4.3: Example of simple breeding of predecessor vectors with equal roots producing a bad child.

Algorithm 1 *Convert Tree to Prüfer Number*

1. Let i be the lowest numbered leaf (node of degree 1) in T. Let j be the node which is the predecessor of i. Then j becomes the rightmost digit of $P(T)$. $P(T)$ is built up by appending digits to the right; thus, $P(T)$ is built and read from left to right.
2. Remove i and the edge (i, j) from further consideration. Thus, i is no longer considered at all and if i was the only successor of j, then j has become a leaf.
3. IF only two nodes remain to be considered

   $P(T)$ has been formed - stop.

   else

   return to Step 1.

   end

end

Figure 4.4: Algorithm for converting a tree into its Prüfer number.

defined by the algorithm in figure 4.4.

As an example of how this algorithm works, consider the tree shown in figure 4.5, below. Node 2 is the lowest numbered leaf and node 3 is its predecessor. Therefore, select 3 as the first digit of $P(T)$. Then remove node 2 and the edge $(2, 3)$ from consideration and node 4 becomes the lowest numbered leaf. Now, the next digit of $P(T)$ is also a 3, since 3 is also the predecessor of node 4. Now remove node 4, making node 3 itself become a leaf and, in fact, the lowest numbered leaf. Node 1, its predecessor is thus the next digit of $P(T)$. Node 5, with predecessor 1, becomes the lowest numbered leaf and the next digit of $P(T)$ is again 1. There are now only two nodes left to be considered, and so the algorithm ends with $P(T) = 3311$.

It is also possible to go from a Prüfer number to a unique tree via the algorithm in figure 4.6.

In the example given in figure 4.5, begin with $P(T) = 3311$ and with nodes $2, 4, 5$ and 6 eligible. Node 2 is the lowest numbered eligible node. Node 3 is the leftmost

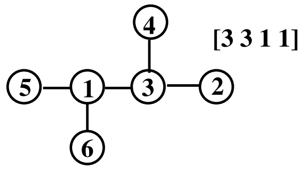


Figure 4.5: A Tree and its Prüfer number.

Algorithm 2 *Convert Prüfer Number to Tree*

1.Let $P(T)$ be the original Prüfer number and let all nodes not part of $P(T)$ be designated as eligible for consideration.

2.IF no digits remain in $P(T)$

  there are exactly two nodes, i and j, still eligible for consideration. (This can be seen by observing that as a digit is removed from $P(T)$ in Step 3 below, exactly one node is removed from consideration and there are $n-2$ digits in the original $P(T)$). Add (i,j) to T and stop.

 end

3.Let i be the lowest numbered eligible node. Let j be the leftmost digit of $P(T)$. Add the edge (i,j) to T. Remove the leftmost digit from $P(T)$. Designate i as not no longer eligible. IF j does not occur anywhere in what remains of $P(T)$

  designate j as eligible.

 end

4.Return to Step 2.

end

Figure 4.6: Algorithm for converting a Prüfer number into a tree.

digit of $P(T)$. Therefore, add edge $(2,3)$ to T, remove 2 from further consideration, and remove the leftmost digit of $P(T)$ leaving $P(T) = 311$. Node 4 is now the lowest eligible node and the leftmost digit of what remains of $P(T)$. As a result, $(4,3)$ is added to T, 4 is removed from further consideration and the second 3 is removed from $P(T)$. Node 3 is now no longer part of $P(T)$ and becomes eligible. Indeed, it is the lowest such number. Therefore, add $(3,1)$ to T, remove the leftmost 1 from $P(T)$, and remove 3 from further consideration. Now $P(T) = 1$ and only nodes 5 and 6 are eligible. Thus, link $(5,1)$ is added to T, the last digit of $P(T)$ is removed, and 5 is designated as no longer eligible. Node 1 is now eligible, since it is no longer part of $P(T)$. $P(T)$ is now empty and only nodes 1 and 6 are eligible. Finally, add $(1,6)$ to T and stop. The tree in figure 4.5 has been formed.

There are $n^{(n-2)}$ Prüfer numbers for a graph with n nodes. This is exactly the number of trees possible in such a graph. There is, in fact, an exact one to one correspondence between trees and Prüfer numbers, the transformation being unique in both directions.

Thus, Prüfer numbers are unbiased (each tree is represented once), they cover the entire space of trees, and they do not represent anything other than trees. The transformations back and forth between edges and Prüfer numbers can be carried out in $O(n \log n)$ with the aid of a heap. The algorithms are, as seen above, somewhat more complex intellectually than those for the two preceding representations, but this is not a serious disadvantage.

The real disadvantage of this representation is that it has relatively little locality. While any offspring formed by taking parts of two Prüfer numbers will indeed be a tree, it need not resemble the parent trees at all. Indeed, changing even one digit of a Prüfer number, can change the tree dramatically. Consider, for example, the six node trees formed from the Prüfer numbers 3241 and 3242 which have only two of their five edges in common. A genetic algorithm for the OCSTP using Prüfer numbers as a tree encoding was built as a part of the research presented in this thesis. The results for problems of 6, 12, and 24 produced by this genetic algorithm were compared to those of a simple heuristic (described in section 5.3). The Prüfer number genetic algorithm's solutions to the three problems were 6%, 22%, and 82%

worse, respectively, than those of the heuristic. This was particularly disturbing since the six-node problem has only 1296 different trees to consider; a simple enumeration would have been faster and would have produced the optimum.

Some other researchers have built genetic algorithms that use these Prüfer numbers as their encoding for tree problems, but they too have had limited success. Recently, Julstrom [28] used them as the encoding for trees in his genetic algorithm for the rectilinear Steiner problem. His results showed that while the genetic algorithm did indeed find valid solutions, it "did not perform as well as other approximation methods." He measured the quality of his genetic algorithm's results against Hwang's [24] lower bound which established that no solution to this problem can be shorter than 67% of the length of a minimal rectilinear spanning tree (MRSPT) over the same points. Another result from Hwang [25] included an algorithm which could find solutions that were 92% as long as a minimal rectilinear spanning tree over the same points. The best result from the genetic algorithm was 99% of the length of the MRSPT over the same points, while the average result was longer than this.

4.3 New Representations for Trees

While each of these encodings met some of the criteria listed above, none were entirely adequate. If tree problems are to be effectively addressed using a genetic algorithm, a more suitable encoding for trees must be found. This became the first goal of this research: to define an encoding for trees that would satisfy most, if not all of these criteria. Several encodings were tried with varying degrees of success before the final one was discovered. Some interesting points along the trail to the final encoding will now be described.

4.3.1 Predecessors with tree grafting

The traditional encodings described in the previous section were all tried and abandoned for the reasons already presented. The first variation that was investigated used predecessor vectors as the encoding and added a special crossover technique to

nodes	Heuristic	GA	GA gain	Pop.Size	Evaluations
6	1.383×10^6	1.383×10^6	$=$	40	500
12	7.135×10^6	6.993×10^6	$+2\%$	$1,000$	$20,000$
24	3.795×10^7	3.875×10^7	-2%	$5,000$	$50,000$
47	1.586×10^8	1.545×10^8	$+3\%$	$10,000$	$100,000$
98	7.801×10^8	8.303×10^8	-6%	$10,000$	$200,000$

Table 4.1: Tree grafting encoding results

address the problems that this encoding has with simple crossover. The algorithm for this "tree grafting" technique is shown in figure 4.7. This algorithm is best described using the examples in figures 4.8 through 4.10.The effect of this algorithm is to graft parts of one tree onto the other, while making sure that no cycles are formed.

The results obtained from this encoding were reasonably good as shown in table 4.1. While these results for the small problems were promising, the results for the larger problems indicated that the genetic algorithm was beginning to lose ground and required a larger population size and a larger number of generations in order to keep up. These larger sizes will slow the convergence of the genetic algorithm, but it was already slow to converge even for the smaller problems. The very large population and number of generations multiplied by the $O(n^2)$ complexity to perform the special crossover just described caused the genetic algorithms for the two larger problems to run for several hours.

4.3.2 Leveled encoding

In order to avoid the creation of cycles, some sort of ordering of the nodes is needed. Given that, a connection rule such as "only connect to higher numbered nodes" could be employed. This idea gave rise to another new representation called the "leveled encoding." Given an n node problem, define a vector, C, of n integers in the range $0 \leq C_i < n^2$ which will encode a displacement into another vector of integers which represent a linearized $n \times n$ array. The chromosome would then consist of the vector C, so that each c_i would correspond to the i'th node's displacement into the $n \times n$

Algorithm 3 *Tree Grafting*

Select the two parents:p1 p2
Randomly choose which root will be dominant
Convert the other parent to have this root
For each of the two children, do:

    1. Initialize the child to have the dominant root
    2. Initialize nodelist for each parent to hold the nodes
    whose predecessor is the root
    3. Build a successor matrix for each parent
    4. Starting with parent p1 when producing the first child
    or with parent p2 when producing the second,
    then alternating until both nodelists are empty, DO
        (a) Pick an entry, e, from the current parent's nodelist
        (b) IF e's position in the child isn't filled yet THEN DO
            add e's predecessor (from the current parent) to e's
            position in the child
        end
        (c) Remove e from current nodelist and add e's successor(s)
        to the current parent's nodelist
        (d) IF e is in the nodelist for the other parent THEN DO
            remove it and add e's successor(s)
            (for the other parent) to the nodelist for the other parent.
        end
    end
end

end

Figure 4.7: Tree grafting algorithm.

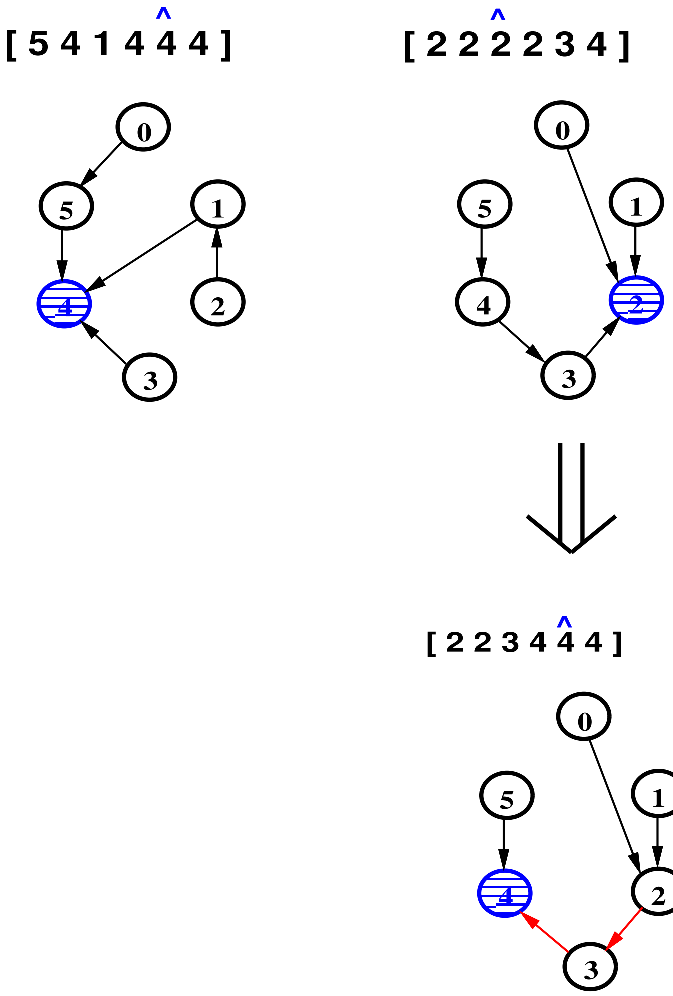
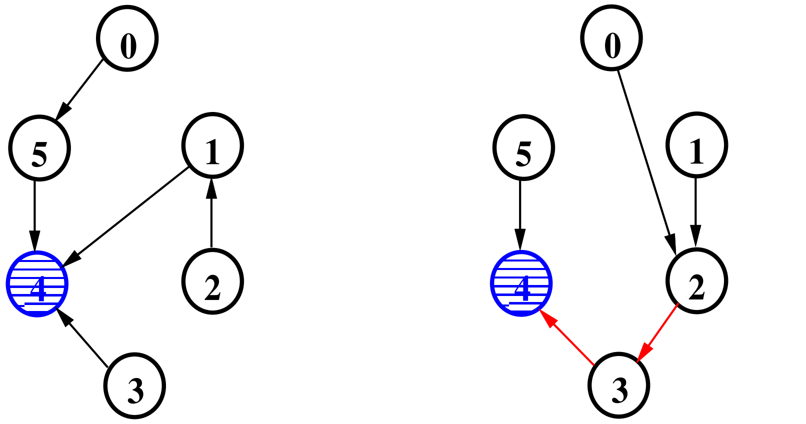


Figure 4.8: Tree grafting algorithm: root selection and conversion.

[5 4 1 4 $\hat{4}$ 4] **[2 2 3 4 $\hat{4}$ 4]**

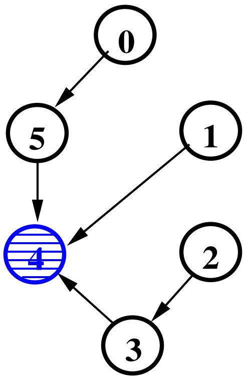


node	succs
0	–
1	2
2	–
3	–
4	1,3,5
5	0

node	succs
0	–
1	–
2	0,1
3	2
4	3,5
5	–

Figure 4.9: Tree grafting algorithm: build successors and nodelists.

step	parent1 nodelist	kid 1	parent2 nodelist
0	{ $\boxed{1}$ 3 5 }	[_4__4_]	{ 3 5 }
1	{ 2 3 5 }	[_4__44]	{ 3 $\boxed{5}$ }
2	{ 2 $\boxed{3}$ 0 }	[_4_444]	{ 3 }
3	{ 2 0 }	[_43444]	{ $\boxed{2}$ }
4	{ $\boxed{0}$ }	[543444]	{ 0 1 }
5	{ }	[543444]	{ $\boxed{1}$ }
6	{ }	[543444]	{ }



step	parent1 nodelist	kid 1	parent2 nodelist
0	{ 1 3 5 }	[_____4]	{ 3 $\boxed{5}$ }
1	{ 1 $\boxed{3}$ 0 }	[____44]	{ 3 }
2	{ 1 0 }	[___444]	{ $\boxed{2}$ }
3	{ 1 $\boxed{0}$ }	[__3444]	{ 0 1 }
4	{ 1 }	[5_3444]	{ $\boxed{1}$ }
5	{ $\boxed{2}$ }	[523444]	{ }
6	{ }	[523444]	{ }

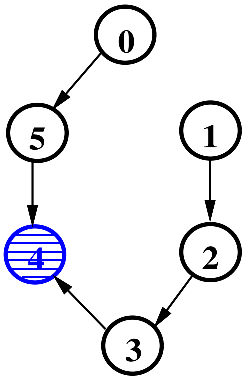


Figure 4.10: Tree grafting algorithm: perform the first graft.

nodes	heuristic	GA	GA gain	Pop. Size	# Gens
6	1.383×10^6	1.383×10^6	0%	65	1,000
12	7.135×10^6	7.402×10^6	−4%	65	1,000
24	3.795×10^7	4.196×10^7	−11%	65	1,000
47	1.586×10^8	2.981×10^8	−94%	130	20,000
98	7.801×10^8	7.356×10^9	−878%	130	20,000

Table 4.2: Leveled encoding results comparison.

array. For example, with $n = 5$, a chromosome of $[3, 11, 22, 23, 13]$ yields the array

$$\begin{bmatrix} - & - & - & 0 & - \\ - & - & - & - & - \\ - & 1 & - & 4 & - \\ - & - & - & - & - \\ - & - & 2 & 3 & - \end{bmatrix}$$

This array is then input to the algorithm in figure 4.11 to produce the tree in figure 4.12.

The crossover operator for this encoding presents several possibilities. The simplest of these is to limit the crossover points within the chromosome to whole integer boundaries as shown in figure 4.13. If traditional crossover is allowed between any pair of bits, valid trees will be produced but they may differ considerably from their parents. For example, given the previous two chromosomes for $n = 5$, with each integer needing 5 bits, the crossover point might occur between bit positions 17 and 18, with the results shown in figure 4.14. Note that a tie had to be broken for the child $c1$; the alternative to the edge $(1, 3)$ is shown as a dotted line.

This encoding was implemented in a genetic algorithm and tested. An $O(n^2)$ algorithm was used to convert from the $O(n \log_2 n^2)$ bit chromosome back into a tree. While it worked as well as a simple heuristic for the small problem of 6 nodes, it performed increasingly poorly for larger problems as shown in table 4.2. Larger populations, as high as one thousand, and much longer running times, as high as one hundred generations, were required in the larger problems to obtain results comparable to those from the simple heuristic.

Even if these shortcomings were tolerable, it was determined that the encoding

Algorithm 4 *Leveled Encoding*

```
    Let the upper left corner be the origin, represented as (0,0), with the
    row numbers increasing downward and the column numbers increasing
    from the left to the right

    Proceed down the array, from 0 to row n - 1

        scan across the row for a non-empty cell
        IF not found
            continue with next row
        else
            Let Start = the node number in this cell.
            Starting with the next row, look for another node that
            1. appears in a higher-numbered row in the table, and
            2. is closest to this cell
            3. in the case of ties, use a scheme like "leftmost" or random
            to choose
            IF such a cell is found
                add a link from Start to the node number in this newly
                found cell
            end
        end
    end
    IF the last non-empty row contains more than one non-empty cell,
        one link is added between pairs of nodes as the scan proceeds from
        left to right along that row
    end

end
```

Figure 4.11: Leveled encoding algorithm.

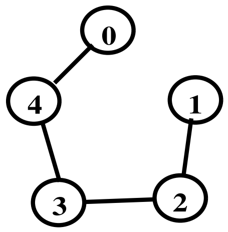


Figure 4.12: Leveled encoding example tree.

P1: 03 11 08 23 13
P2: 16 15 10 17 21
x------------
C1: 03 11 08 17 21
C2: 16 15 10 23 13

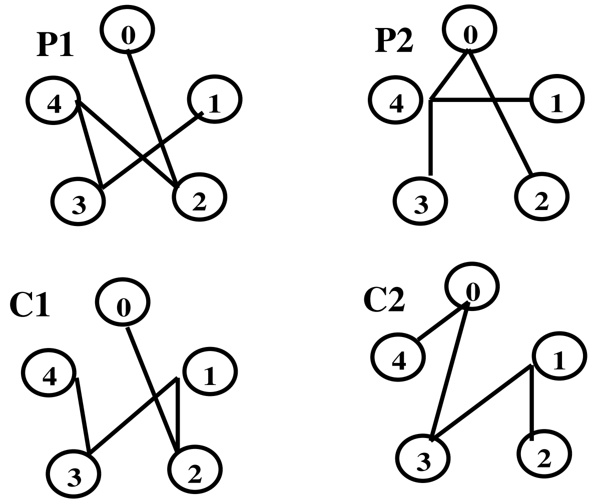


Figure 4.13: Integral boundary crossover for leveled encoding.

P1: 00011 01011 01000 10111 01101 (3 11 8 23 13)
P2: 10000 01111 01010 10001 10101 (16 15 10 17 21)
x--
C1: 00011 01011 01000 10101 10101 (3 11 8 21 21)
C2: 10000 01111 01010 10011 01101 (16 15 10 19 13)

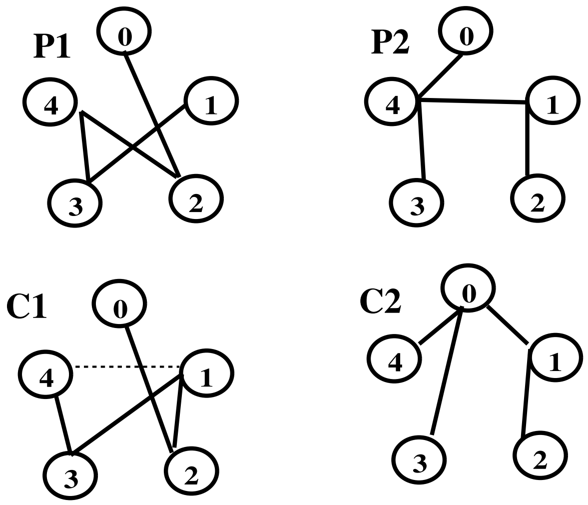


Figure 4.14: Bit-wise crossover for the leveled encoding.

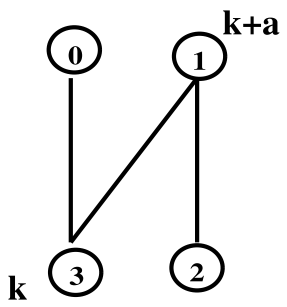


Figure 4.15: Example of a tree the leveled encoding cannot represent.

did not cover the desired space of trees[2]. Suppose the tree in figure 4.15 is to be represented using the leveled encoding. Using l_n to represent the level of node n, the figure shows $l_3 = k$ and $l_1 = k + a$. There are two cases:

Case 1: If $a > 1$, then it may be that $k < l_2 < k+a$. If this is the case, then given that the connection rule is "connect to the *nearest* node with a higher level number", link $(1, 2)$ would be selected because node 1 is the nearest higher-leveled node to node 2. Further, link $(2, 3)$ would also be selected because $l_2 > l_3$ and node 2 is nearer to node 3 than node 1. In this situation, the tree in figure 4.15 cannot be reached.

Case 2: If $a \leq 1$, when $l_c > k$ the link $(1, 3)$ is replaced by the link $(2, 3)$ and when $l_c < k$ the link $(1, 2)$ link is replaced by the link $(2, 3)$. In this situation, the tree in figure 4.15 cannot be reached.

These arguments assume, of course, that the triangle inequality holds for these distances. If it does not hold, a similar argument could be made that also leads to the leveled-encoding not covering the desired space of trees.

Although the encoding performed well for the six node problem, it apparently did so without being able to search the whole space of trees. This limitation became more

[2]For this example, a graphical proof was evident. Another convenient coverage test is to see if an encoding can produce all of the n-2 digit Prufer numbers.

harmful as the size of the problem increased. Due to this fact and the very long run times, this encoding was abandoned.

4.3.3 Permuted predecessors encoding

The predecessor representation of rooted trees has many desirable properties. It is reasonably compact. There is a natural transformation between the encoding and the representation of the tree as a set of edges. It covers the entire space of solutions in an unbiased way. It is reasonably local in that changing one predecessor changes just one edge in the tree. Its major drawback is that it is capable of representing non-trees. As was mentioned earlier, a random n digit number has a probability of only $\frac{1}{n}$ of representing a tree.

Using such a representation in a genetic algorithm decreases the effectiveness of the genetic algorithm dramatically. To make use of it one must either accept a large percentage of non-viable members in the population or else come up with rules and constraints which make the production of non-trees less likely. The former causes a genetic algorithm to drift and seldom converge to good solutions due to the high probability ($\frac{n-1}{n}$) of standard crossover producing non-trees. The latter do not just complicate the implementation of the genetic algorithm; they undermine its validity by potentially introducing a bias into the breeding process. This can result in an incomplete search of the solution space and the loss of one of the major strengths of the genetic algorithm approach.

It is possible, however, to modify the predecessor representation to completely eliminate non-trees without destroying any of its other desirable properties. By doing so, a representation is obtained which satisfies all of the criteria given in section 4.1 except for, as is shown later, the locality criteria.

The key to improving the predecessor representation is to see what can go wrong with the original one. Any tree, T, can be represented by a vector of predecessors, $P(T)$, which can also be thought of as an n digit number. Again, each digit is a number between 1 and n. Now, however, there are n digits instead of $n-2$. The rule used for transforming a predecessor "number" to a tree is much simpler than the rule for Prüfer numbers: if the i-th digit of $P(T)$ is j, add edge (i, j) to T.

By convention, the r-th digit of $P(T)$ is set to r when the root of the tree is r and this digit is ignored in adding edges to T. Thus, any tree can be represented by n possible $P(T)$'s by selecting each of the n nodes as the root. Three of the possible $P(T)$'s for the tree in figure 4.5 are 131311, 322311, and 631316, which correspond to making the root nodes 1, 2, and 6, respectively.

There are, however, n digit numbers which do not correspond to any tree. Consider, for example, the number 114635. This corresponds to a graph with the cycle 34653 in it and could be formed by initially selecting random digits or by breeding valid trees (e.g., by taking the leftmost five digits of tree 114631 and the rightmost one digit of tree 555555).

So, the problem is that it is possible for the predecessors to include one or more cycles. This problem may be avoided, however, by ordering the nodes and then only allowing a node to choose its predecessors from among the nodes to its right. By restricting the selection of predecessors in this manner, all edges will "point to the right" and no cycles can be formed.

Any tree, and only trees, can be characterized by the following rules:

1. Let $S_1(T)$ be an ordering of the nodes; i.e., a permutation of the integers from 1 to n.

2. Let $S_2(T)$ be an $n-1$ digit number such that all the digits are between 2 and n and the i-th digit is strictly greater than i. The i-th digit of $S_2(T)$ gives the position (in the ordering given by $S_1(T)$) of the node which is the predecessor of i in T.

For example, one possible representation of the tree in figure 4.5 is $S_1(T) = 651432$ and $S_2(T) = 33556$. This encoding is interpreted as follows:

- Node 6 is the first node in the ordering and its predecessor is the third node in the ordering (node 1),

- Node 5 is the second node in the ordering and its predecessor is the third node in the ordering (node 1), etc.

Any tree is representable in this way. In fact, any tree is representable in many ways. First, the above representation is for rooted trees, so any tree can be represented in at least n ways, considering each of the nodes as the root. But this is not all. $S_1(T)$ is a permutation of n numbers. Thus, $S_1(T)$ can take $n!$ values. The leftmost digit of $S_2(T)$ can take $n-1$ values; The next leftmost digit can take $n-2$ values; etc. So, $S_2(T)$ can take $(n-1)!$ values. Each combination of S_1 and S_2 gives rise to a tree, and no combination of S_1 and S_2 gives rise to anything other than a tree since the interpretation of S_2 guarantees that there are no cycles.

Given a specific tree, T, a node may be selected as the root and then a postordering of the nodes may be built[3]. This postordering leads to $S_1(T)$. A given tree, T, and a postordering, $S_1(T)$ uniquely specifies $S_2(T)$. Conversely, given any S_1 and S_2 it is easy to form a specific tree T.

Unfortunately, different rooted trees have different numbers of postorderings associated with them. A chain rooted at one end has only one. A star rooted at the center has $(n-1)!$. Indeed, different rooted trees corresponding to the same tree can have a different number of postorderings associated with them. The same chain mentioned above, this time rooted at a node at the center of the chain, has

$$\frac{(n-1!)}{\left(\left(\frac{(n-1)}{2}\right)!\right)^2} = \binom{n-1}{\frac{(n-1)}{2}}$$

postorderings associated with it, for n odd.

Since there are $n^{(n-2)}$ possible trees and $n!(n-1)!$ possible values for S_1 and S_2, the average number of times a tree is represented by an S_1,S_2 combination is

$$\frac{n!\,(n-1)!}{n^{(n-2)}}$$

Note that this quantity is not, in general, an integer.

So, this representation is biased in that some trees are represented more times than others. Fortunately, the number of times each tree is represented may be determined and, thus, a compensation for the bias may be made.

[3]A postorder of the nodes is any ordering which places predecessors to the right of their successors. In general, there are many postordering corresponding to a given rooted tree.

One way of doing this is to observe that a given tree T can not be obtained from just any permutation. It must be a permutation which represents a postordering of the nodes of T. Since $S_1(T)$ uniquely specifies $S_2(T)$ for a given T, the number of times T is representable by an S_1,S_2 combination is exactly equal to the number of permutations of the numbers from 1 to n which represent postorderings of T.

Theorem 1 *Let T be a rooted tree on n nodes. Let n_i be the number of nodes in the subtree rooted at node i, where i is any node in T. Then the number of postorderings of the nodes in T is given by:*

$$\frac{n!}{\prod_j n_j}$$

Proof. The proof is by induction on the number of nodes in the tree. There is only one tree on two nodes and this tree satisfies the conditions of the theorem. Suppose that all trees with fewer than n nodes satisfy the conditions of the theorem. Now consider a tree, T, with n nodes. There are two cases to consider.

Case 1: The root of T has only one subtree, T'. In this case, every postordering of T is just a postordering of T' with the root of T appended to the end of the postordering, and the number of postorderings of the nodes of T is the same as the number of postorderings of the nodes of T'. This satisfies the conditions of the theorem since, in going from T' to T, the numerator is multiplied by n (going from $(n-1)!$ to $n!$) and the denominator is also multiplied by n (as an n is added to the product to account for the new (sub)tree, T, with n nodes in it). Thus, the number of subtrees in T satisfies the theorem.

Case 2: The root of T has two or more subtrees. For simplicity, first consider the case of exactly two subtrees. Let T_1 be one of these subtrees and let T_2 be the other subtree. Both T_1 and T_2 have fewer than n nodes and so, by the induction hypothesis, both satisfy the theorem. Let $N(A)$ be the number of nodes in an arbitrary tree A and let $NP(A)$ be the number of postorderings of the nodes of A. Any postordering of the nodes of T can be thought of as a postordering of the nodes of T_1 interleaved with a postordering of the nodes of T_2. The number of ways of interleaving nodes of T_1 and T_2 is equal to:

$$\frac{(N(T_1) + N(T_2))!}{(N(T_1))!\ (N(T_2))!}$$

Since $N(T_1) + N(T_2) = n - 1$, this is $\binom{n-1}{N(T_1)}$, or

$$\frac{(n-1)!}{(N(T_1))!\ (N(T_2))!}$$

Within each interleaving, there are $NP(T_1)$ ways of arranging the nodes of T_1 and $NP(T_2)$ ways of arranging the nodes of T_2. By the induction hypothesis,

$$NP(T_i) = \frac{(N(T_i))!}{\prod_{j \in T i} n_i}$$

where the product is taken over all subtrees in T_i, for $i = 1, 2$. Thus the total number of postorderings of the nodes of T is the product of these three factors

$$\frac{(n-1)!}{(N(T_1))!\ (N(T_2))!} \times \frac{(N(T_1))!}{\prod_{j \in T_1} n_j} \times \frac{(N(T_2))!}{\prod_{j \in T_2} n_j} = \frac{n!}{\prod_{j \in T} n_j}$$

The denominator in the last expression is the product taken over all subtrees in T and is obtained by observing that the subtrees of T are precisely those of T_1 plus those of T_2 plus T itself (with n nodes).

A similar argument can be made when the number of subtrees is more than two. This case has corresponding products and terms for each subtree, T_i, of T for an arbitrary i. □

By knowing the number of postorderings, the number of representations for each tree is also known. It is thus possible to compensate for the different number of times each tree is represented (e.g., by keeping a tree with probability equal to the reciprocal of this number), and thus obtain an unbiased set of trees. The effort in computing this factor is $O(n)$.

Next there is the problem of generating the S_1,S_2 combination in a way that will encourage short, low-order schemata in the chromosome. One way of doing this is to assign an arbitrary numbering of the nodes. If these numbers are sorted, say smallest to largest, a permutation is made. If two numbers are associated with each node, two

Algorithm 5 S_2

1. Let X be the identity permutation on the integers from 1 to $N-1$. Let P be an arbitrary permutation of the integers from 1 to $N-1$.
2. FOR $i = 1$ to $N-1$ DO Step 3.
3. Let j be $P[i]$, the number in position i of P. Let k be position that j currently occupies in X. Set $S_2[i] = i + k$. Exchange j and $X[N-i]$ in X.

end

Figure 4.16: S_2 algorithm.

permutations are obtained. The first permutation is used directly to obtain S_1. The second is used to obtain S_2 using the algorithm in figure 4.16.

As an example, suppose that n is 6 and these 6 random numbers are generated: 37, 90, 78, 61, 32, 19. These are sorted from smallest to largest, yielding 19, 32, 37, 61, 78, 90 and the permutation 651432 (from the original positions of the numbers currently occupying each of the positions in the sorted sequence). Thus, $S_1 = 651432$. Next, 5 random numbers are generated: 40, 29, 86, 68, 67. Sorting these smallest to largest yields the permutation $P = 21543$ and S_2 is generated using the above algorithm as shown in the execution trace in figure 4.17. Thus, the permutation $P = 21543$ yields $S_2 = 33556$.

If the algorithm to generate S_2 is carefully implemented, keeping the inverse permutation of X (i.e., keeping track of which number is in each position and in what position each number is), the algorithm is $O(n)$. Thus, transforming back and forth between the tree and the encoding is $O(n)$, which is the best complexity possible. Another feature of this encoding is that the higher values of S_2, i.e. having schemata with higher-order bits set, correspond to "star-like" trees while the lower values correspond to "chain-like" trees.

Unfortunately, this encoding proved only productive for small numbers of nodes. Using several sets of genetic algorithm control parameters, experiments using this

```
i = 1; j = P[i] = 2; k = 2 since X[2] = j; S2[1] = i + k = 3
  Exchange X[2] and X[5]: X = 15342
i = 2; j = P[i] = 1; k = 1 since X[1] = j; S2[2] = i + k = 3
  Exchange X[1] and X[4]: X = 45312
i = 3; j = P[i] = 5; k = 2 since X[2] = j; S2[3] = i + k = 5
  Exchange X[2] and X[3]: X = 43512
i = 4; j = P[i] = 4; k = 1 since X[1] = j; S2[3] = i + k = 5
  Exchange X[1] and X[2]: X = 34512
i = 5; j = P[i] = 3; k = 1 since X[1] = j; S2[3] = i + k = 6
  Exchange X[1] and X[1]: X = 34512
```

Figure 4.17: Trace of algorithm S_2.

technique	6	12	24	47	98
GA	1,386,360	7,035,895	45,712,100	192,500,000	1,800,810,000
Heuristic	1,386,360	7,134,530	37,952,000	158,612,156	780,999,474

Table 4.3: Comparison of "right hand rule" GA and Heuristic results

encoding scheme produced results that did not compare well to those provided by a simple heuristic[4]. As shown in table 4.3 , for problems larger than 12 nodes the genetic algorithms performed increasingly poorly. Upon further examination, it became clear that the encoding wasn't able to maintain good locality. The wide separation of the predecessors on the left of the chromosome and the permutation on the right led to schemata of rather high length. As discussed earlier, dependence upon long schemata has a negative effect on the performance of a genetic algorithm and can produce the kind of mediocre results that were encountered in spite of the fact that the encoding possessed many other desirable qualities.

4.4 The Node and Link Biased Encoding

The search for a new encoding began by noting that experience has shown that for a given problem (nodes, requirements, and costs), certain nodes should be interior

[4]The nature of this and another comparison heuristic will be discussed in the next chapter.

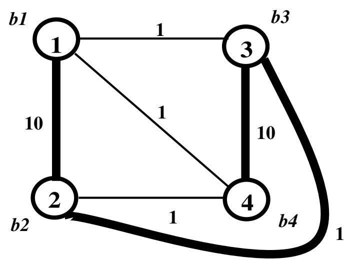


Figure 4.18: Example of a tree not representable using only node biases.

nodes and others should be leaf nodes. With this in mind, the new encoding was designed so that the genetic algorithm would be allowed to search for nodes with these tendencies while looking for solutions to the OCSTP [33]. In the first version of this encoding, the chromosome holds a *bias* value for each node. For example, in a four node problem the chromosome would contain four biases $[b_1 b_2 b_3 b_4]$. Node values are multiplied by a control parameter, P, and by the maximum link cost, $C_{\max}$, and are added to the cost of all links that have the node as an endpoint. The cost matrix would be biased by these values using

$$C'_{ij} = C_{ij} + P\,(b_i + b_j)\,C_{\max}$$

The tree that the chromosome represents is then found by applying Prim's algorithm [13] to find a minimal spanning tree (MST) over the nodes using the biased cost matrix. Finally, this MST is evaluated using the original cost matrix to determine the tree's fitness for the OCSTP.

This seemed sufficient at first, but it was later found that it didn't cover the space of all trees in certain cases. To see this, consider the network shown in figure 4.18. Suppose the encoding for the tree having links $(1,2)$ $(2,3)$ $(3,4)$ is sought. It is well known that a MST cannot contain the longest edge in any cycle. Thus, if $(1,2)$ is to be part of the MST, it must be true that

$$C'_{12} \leq C'_{13} \tag{4.1}$$

$$C'_{12} \leq C'_{23} \tag{4.2}$$

since $(1,2)$ $(2,3)$ $(1,3)$ form a cycle. Similarly,

$$C'_{12} \leq C'_{14} \tag{4.3}$$

$$C'_{12} \leq C'_{24} \tag{4.4}$$

Also, if $(3,4)$ is to be part of the MST, it must be true that

$$C'_{34} \leq C'_{13} \tag{4.5}$$

$$C'_{34} \leq C'_{14} \tag{4.6}$$

$$C'_{34} \leq C'_{23} \tag{4.7}$$

$$C'_{34} \leq C'_{24} \tag{4.8}$$

But, these inequalities contain four contradictory pairs. For example,

$$\begin{aligned} (4.1) \Rightarrow \qquad\qquad C'_{12} &\leq C'_{13} \\ 10 + b_1 + b_2 &\leq 1 + b_1 + b_3 \\ b_2 + 9 &\leq b_3 \end{aligned} \tag{4.9}$$

$$\begin{aligned} (4.8) \Rightarrow \qquad\qquad C'_{34} &\leq C'_{24} \\ 10 + b_3 + b_4 &\leq 1 + b_2 + b_4 \\ b_3 + 9 &\leq b_2 \end{aligned} \tag{4.10}$$

Thus, there is no choice of $[b_1 b_2 b_3 b_4]$ that would result in the links $(1,2)$ $(2,3)$ $(3,4)$ being selected as the MST using the given link costs. Such a tree is clearly not useful as a solution to the OCSTP, but in the interest of producing a generally useful tree representation the representation was extended to include link biases as well.

In this second version of the representation, the chromosome has biases for the n nodes and each of the $\frac{n(n-1)}{2}$ links, for a total of $\frac{n(n+1)}{2}$ biases. The genetic algorithm itself now has two additional parameters, P_1 and P_2, for use as multipliers (along with $C_{\max}$) on the link and node biases, respectively. The cost matrix is then biased by both of these values using

$$C'_{ij} = C_{ij} + P_1 b_{ij} C_{\max} + P_2 \left(b_i + b_j\right) C_{\max} \tag{4.11}$$

This version of the representation can encode any tree, T, given appropriate values the b_i,b_j, and b_{ij}. This may be done by setting $b_i = 0$, for all i, and setting

$$b_{ij} = \begin{array}{l} 0 \; for \; (i,j) \in T \\ M \; otherwise \end{array}$$

where M is larger than the maximum value of the C_{ij}. For the experiments described here, the b_i, b_j, and b_{ij} were allowed to take on values in the range $[0, 255]$, stored as 8 bits, and normalized to the range $[0, 1)$ before use in Equation 4.11. The parameters P_1 and P_2 are fixed for a single genetic algorithm experiment. For the OCSTP, the P_1 value, which is a parameter controlling the effect of link biases on the solution, was set to zero, thereby disabling the link bias' effect. This followed the intuition that in the OCSTP the primary question is whether a node should be an interior or exterior node. Several of the link biases would have to work together to cause a given node to be forced into being an interior or exterior node. This could require long schemata since the positions of the relevant b_{ij} could be as far as $(n^2 - 1)$ bias positions apart on the chromosome. The node biases, however, succinctly describe this interior versus exterior behavior within their 8 consecutive bits. As a result, the early OCSTP experiments were all made with $P_1 = 0$ and $P_2 = 1$.

The complexity of the decoding algorithm for this encoding is $O(n^2)$. After several runs of genetic algorithms employing this new representation, it appeared to be quite promising. With only a little experimentation, a set of genetic algorithm control parameters and $P1$ and $P2$ encoding control parameters were found that consistently found solutions that seemed as good or better than simple heuristics for small problems. The parameters chosen were:

Population	100	σ *Scaling*	1.0
Crossover	0.6	P_1	0.0
Mutation	0.01	P_2	1.0
Gen Gap	1.0		

These parameters, with the exception of P_1 and P_2, which are specific to this new encoding, closely matched those generally held to be good choices for by other researchers [17].

The encoding also appears to encourage the evolution of short schemata since the importance of a node being an interior or exterior node can be expressed in that node's b_i value: a large value would considerably increase the costs of all links that might be incident to it, thereby discouraging such a link, whereas a small value would increase the costs of incident links only a little, thereby encouraging the inclusion of that link by Prim's algorithm. The order of the schemata can also be low, since just a few of the more significant bits in a b_i would determine whether or not it is a large or small value.

Given what appeared to be a very good encoding for trees along with apparently good genetic algorithm and encoding control parameter values, the next step was to find out just how well it worked on real problems as compared to random selections, strong heuristics, and simulated annealing.

Chapter 5

Experimental Results

Once the link and node biased (LNB) genetic algorithm for the OCSTP was assembled, several runs of the genetic algorithm were made for each of five problems to evaluate its performance. These problems consisted of 6, 12, 24, 47, and 98 city (node) networks, with inter-city traffic requirements inversely proportional to the distance between the cities, and with link costs obtained from a tariff database. Since genetic algorithms are randomized procedures, all of the results reported in this thesis are based upon averages of five or more runs for each problem. For all of the experiments, the control parameters were set to the values identified in section 4.3. Of course, halting a genetic algorithm after some arbitrary number of evaluations may not allow the genetic algorithm to complete its work. Outside of a strictly controlled experiment such as is described in this thesis, a genetic algorithm could be allowed to work until one or more termination conditions are met. Examples of such termination conditions are:

- the number of chromosome positions that must converge to a single value. These might be parts of the same gene, or feature, or the whole gene. For example, if there are eight bits that represent some integer valued feature of the problem, if the most significant bit is set to 1 in every chromosome in the population, that is one chromosomal position that has converged. This bit will never change in future generations, unless it does so as a result of mutation.

- the number of consecutive generations wherein no new evaluations were required. If a population has evolved to the point that, except for mutation, there is no combination of two parent chromosomes and a crossover point that would produce an offspring not already represented in the population, then there is little reason to continue the experiment. If the mutation rate was sufficiently high so as to provide a reasonable chance of a change resulting in a chromosome fit enough to reproduce, it may prove useful to run the experiment longer.

- the average percentage of the most common value in each position is above a specified threshold, called the gene bias. This means that over the whole population, this percentage of the chromosome positions, or genes, have converged to a specific value. For example, in a binary encoding case with a bias level of 80% this means that, on average, each position on the chromosome has evolved to 80% 0's or 80% 1's throughout the population.

In these experiments, the LNB genetic algorithm encoding was employed and was given a budget of $10,000$ evaluations. As shown later in this chapter, the LNB genetic algorithm provided solutions comparable to, if not better than, those of some standard heuristics and a simulated annealing approach. For some of the experiments, the optimal solution was previously known and the genetic algorithm always found it. In some cases, it appeared that a larger population size or an increased number of trials would have allowed the genetic algorithm to continue to improve its solutions. However, the additional improvement was typically minimal and, therefore, did not justify the additional computer time.

The effectiveness of the LNB genetic algorithm was evaluated in several ways. The genetic algorithm's results were first compared to those from a purely random search. Then, since optimal solutions to the OCSTP are not generally known, the trees produced by the LNB genetic algorithm were compared to those produced by two good heuristics: star-search, described in section 5.3, and local-exchange, described in section 5.4. Next, the characteristics of the underlying network were changed so as to strongly encourage a two-level, multiple-star network in order to see how well the genetic algorithm and the two heuristics would adapt. Finally, as a further test of the LNB genetic algorithm's adaptability, the same LNB encoding was used in a genetic algorithm to search for solutions to a delay analysis problem quite different from the OCSTP but whose solutions were still required to be trees. For this new problem, the genetic algorithm results were compared against those obtained by simulated annealing.

5.1 Experiment Environment

All of the experiments described in this thesis were carried out using IBM RiscSystem/6000 model 550 and 560 workstations having performance ratings of 40 and 50 MIPS. As described earlier, some of these experiments ran for minutes and others for up to twenty-five days. The results reported in this thesis are the result of runs totalling more than 90 CPU-days on these workstations.

A generally available genetic algorithm experimentation tool, GAucsd 1.4[37], was ported to the IBM AIX operating system and the above workstation for this research. The original version of this tool is available via anonymous ftp from *cs.ucsd.edu (132.239.51.3)* in the subdirectory pub/GAucsd/GAucsd14.sh.Z, or requested via email to *nici@cs.ucsd.edu*. This tool is supplied in C source code form and has been ported to most of the popular platforms. Well-suited for genetic algorithm research, the GAucsd tool, and its ancestor Genesis [19], have been used by researchers around the world. Since the tool is supplied in source form the user is able to change any aspect of the genetic algorithm process, such as crossover operation, mutation, or selection. The tool is particularly useful for developing and testing new encodings and fitness functions.

5.2 Random search comparison

We compared the distribution of solutions found by a purely random search to the distribution of the solutions to a 24-node problem found by the genetic algorithm. As shown in figure 5.1, the genetic algorithm using $10,000$ trials consistently found

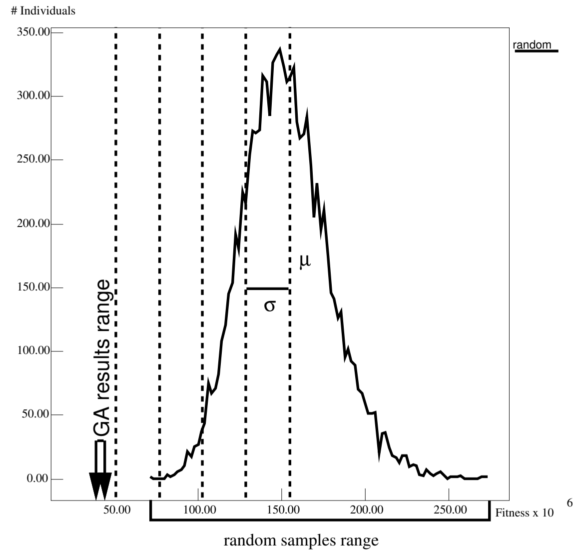


Figure 5.1: GA comparison with one million random samples for N=24.

solutions superior to a random search of one million solutions by more than 4 standard deviations.

5.3 Star-Search Heuristic Comparison

The first heuristic [34] is shown in figure 5.2. It returns the best tree among stars on one node, connected pairs of stars, and trees based upon a minimal spanning tree (MST) with interior nodes reduced. A comparison of the costs of the trees found by the LNB genetic algorithm to the trees found by the heuristic shows that the genetic algorithm consistently found solutions as good as or better than those found by the heuristic as shown in table 5.1.

The results for the $N = 6$ case, which has a search space of only $6^4 = 1296$, show that the genetic algorithm is somewhat unstable for really easy problems. While the genetic algorithm found the optimum, it did not fully converge to that value as it did for the $N = 12$ and $N = 24$ cases. The best trees found by the LNB genetic algorithm and the heuristic for $N = 98$ are shown in figures 5.3 and 5.4, respectively.

Algorithm 6 *Star-Search*

1. Evaluate all of the n trees that are stars around one node.
2. Save the best single-star tree.
3. Evaluate all of the n^2 trees that are two stars around two connected nodes.
4. Save the best double-star tree.
5. Choose a center M. This might be, for example, the center of the best star found in the previous four steps.
6. Find an MST, T, starting from the center over the set of nodes.
7. Compute $S(T)$, the weighted (by the size of the requirements) sum of the lengths of the paths between all pairs of nodes.
8. Working inward towards M, for each node i where $i \neq M$ and $P[i]$, the predecessor of i in the current tree, is not M, DO
   - (a) Let PP be the predecessor of $P[i]$
   - (b) Try making PP the predecessor of i in a temporary tree T'.
   - (c) Evaluate the fitness of T'.
   - (d) IF $S(T') < S(T)$
     - set $T = T'$ and $P[i] = PP$

     end
   - (e) IF $PP \neq M$ THEN goto step (a)

   end
9. Save the final improved MST tree.
10. Return the minimum of the best single-star tree, best double-star tree, and improved MST.

end

Figure 5.2: Star-search algorithm.

	Star-	**LNB Genetic Algorithm**			
n	**Search**	**minimum**	**average**	**maximum**	**% gain**
6	1.386×10^6	1.386×10^6	1.413×10^6	1.420×10^6	0
12	6.857×10^6	6.857×10^6	6.857×10^6	6.857×10^6	3.9
24	3.664×10^7	3.603×10^7	3.603×10^7	3.603×10^7	5.4
47	1.478×10^8	1.426×10^8	1.434×10^8	1.444×10^8	5.9-7.2
98	7.331×10^8	7.038×10^8	7.119×10^8	7.184×10^8	4.5-6.4

Table 5.1: Star-search heuristic results comparison.

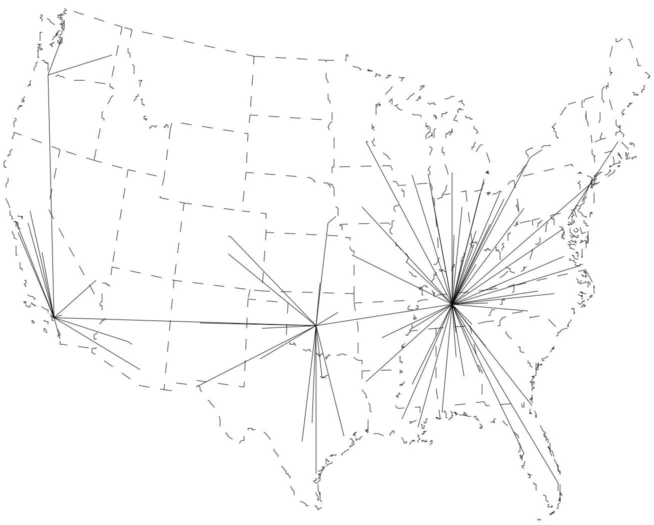


Figure 5.3: LNB GA network for $N = 98$.

These figures provide an excellent example of when the genetic algorithm results could be used by a heuristic designer for examples of what good solutions look like. It is not uncommon for the heuristic designer to have little idea what “good” solutions will look like. In this case, none of the three parts of the heuristic have any chance of finding a tree having six stars as in figure 5.3. Armed with this knowledge, the heuristic designer could set out to build a heuristic that could find a solution having a larger number of stars.

Figure 5.4: Star-search heuristic network for $N = 98$.

	Local Exchange		**LNB Genetic Algorithm**		
n	**time**	**minimum**	**time**	**minimum**	**% gain**
6	1 min.	1.386×10^6	1 min.	1.420×10^6	0
12	3 min.	6.857×10^6	3 min.	6.857×10^6	0
24	12 min.	3.664×10^7	11 min.	3.603×10^7	1.7
47	1.2 days	1.478×10^8	58 min.	1.426×10^8	3.7
98	> 4 days	7.331×10^8	347 min.	7.038×10^8	4.2

Table 5.2: Local-exchange heuristic results comparison.

5.4 Local-Exchange Heuristic Comparison

The second algorithm, a standard local-exchange algorithm, is given in figure 5.5. Because this heuristic uses an open-ended iterative approach, its runtime had to be bounded in some way. For comparison with the genetic algorithm results, this heuristic was allowed to run at least as long as the genetic algorithm and, in the larger problems, much longer. Once again, the genetic algorithm was able to find solutions as good as or better than the heuristic in all five problems as shown in table 5.2. Each execution of this algorithm involves between $O(n^5)$ or $O(n^6)$ operations, as listed in table 5.3, multiplied by the number of samples requested. As the results

Algorithm 7 *Local-Exchange*

```
DO for a input number of passes
      1.Randomly produce a tree
      2.FOR each link, DO
            (a)Remove the link and find the set of
               links crossing the cut
            (b)FOR each link in this cut-set, DO
                    i. add the link to the cut tree
                    ii. evaluate the cost of this new
                    tree
                    iii. remember the new tree with
                    the lowest cost

               end
        end
      3.IF the best new tree found is an improvement
        over the previous best
            keep the tree as the best and go to step
            2
        end
end

end
```

Figure 5.5: Local-Exchange OCSTP Algorithm.

step	**complexity**
2	$O(n)$
2.(b)	$O(n)$ to $O(n^2)$
2.(b).ii	$O(n^2)$
3	$O(n)$
overall	$O(n^5)$ to $O(n^6)$

Table 5.3: Computational complexity of Local-Exchange.

in table 5.2 show, as the problem size (n) doubled from 6, to 12, and then 24, the running times were similar. After $N = 24$, the heuristic had to run for more than a factor of 2^6 times as many evaluations than the $O(n^2)$ genetic algorithm in order to find solutions comparable to that of the genetic algorithm. For the $N = 98$ case, this heuristic was able to improve upon the previous one's result by 3.5%, but still lags behind the genetic algorithm by 4.5 to 6.4%. Once again, the goal here is not to try and beat the heuristics with a genetic algorithm, but rather to find a genetic algorithm that can find reliably good solutions from which existing heuristics may be improved or new ones designed.

5.5 Tuning control parameters with a Meta-GA

When a genetic algorithm approach is chosen for a problem, the primary initial work is to design a suitable encoding, specialized crossover and mutation operators, if necessary, and a fitness function that accurately represents the "goodness" of a solution. In addition, the genetic algorithm control parameters described in chapter 3 must be set. Prior research by DeJong [5] and Grefenstette [17] has provided guidelines for setting the control parameters for a typical problem, but there is always some doubt as to how typical one's problem really is. Grefenstette's parameters were determined through the use of a meta-genetic algorithm (meta-GA). A meta-GA is a genetic algorithm that has as its fitness function another genetic algorithm called the objective genetic algorithm. The parameter space (i.e. chromosomes) that the meta-GA searches is the space of genetic algorithm control parameter combinations. The evaluation of a meta-GA chromosome is done by running the fitness function, i.e. the objective genetic algorithm, using the control parameters specified in the meta-GA chromosome. The fitness of the best solution found by the objective genetic algorithm is returned to the meta-GA as the fitness of that chromosome (set of control parameters).

A series of meta-GA's was run using the LNB genetic algorithm for the OCSTP where $N = 24$ in order to see what the best genetic algorithm control parameters really are and to verify the intuition about the unimportance of the b_{ij} values to

parameter	**low**	**high**	**step**
population	50	5050	50
p_c	0.4	1.4	0.01
p_m	0.0	0.2	0.002
gap	0.3	1.0	0.007
σ scaling	1.0	5.0	0.04
P_1	0.0	1.0	0.05
P_2	0.0	1.0	0.05

Table 5.4: Meta-GA parameter ranges.

the OCSTP. The meta-GA's were set up to vary the control parameters through the ranges shown in table 5.4. The meta-GA fitness functions used execution time to break ties between parameter sets that resulted in equal fitnesses, a decision that had a direct effect upon some of the results.

The primary difficulty with meta-GA's is that they tend to have extremely long running times. For example, the search space defined by the above ranges is 10^{14} possible solutions is modest when compared to the size of the space for 98 node trees (98^{96}). However, each one of these possible solutions takes from a few minutes to a few hours of running time to evaluate. Each run involves running a genetic algorithm with those parameters not once, but several times, due to genetic algorithms being randomized procedures. Further, since the meta-GA itself is also a randomized procedure, it too should be run several times and the results averaged. For these meta-GA's, approximately sixty days of workstation time was consumed. The average of the meta-GAs' parameter choices varied little from those parameters chosen for the experiments described in this thesis. The general recommendations from DeJong [5] and Grefenstette [17] are also included in table 5.5 for comparison. The pairs of values in the last column reflect Grefenstette's separate study of which parameter values would optimize DeJong's two kinds of effectiveness measures for a genetic algorithm: off-line and on-line performance. DeJong defined on-line performance as an average of all objective function evaluations so far, and off-line performance as a running average of the best performance values so far.

The values used in the experiments being reported in this thesis differ somewhat

parameter	**thesis**	**Meta-GA**	[5]	[17]
population	100	80	50	30/80
p_c	0.6	1.0	0.6	0.95/0.45
p_m	0.01	0.124	0.001	0.01
gap	1.0	0.74	1.0	1.0
σ -scaling	1.0	1.37	1.0	0.9
P_1	0.0	0.044	n.a.	n.a.
P_2	1.0	0.95	n.a.	n.a.

Table 5.5: Meta-GA results comparison.

from those of the meta-GA and others. The experiment value used for population size was larger than all the others. Larger populations can increase the number of schemata available for the search process, but they do so at the expense of slower convergence. The difference in population sizes here appears inconsequential. The crossover rates seem to fall into two groups: the value used by these experiments, DeJong, and Grefenstette (on-line), versus those used by Grefenstette (off-line) and the meta-GA. DeJong felt that the use of a crossover rate of 0.6 was a reasonable compromise between the competing forces of off-line and on-line performance. The higher crossover rate indicated by the meta-GA can be attributed to higher rates leading to better off-line performance which, in turn, implies faster population convergence and, thus, a shorter execution time. The meta-GA value for the gap parameter appears to have been linked with the meta-GA value for mutation rate. A gap setting of 1.0 is generally thought to be best for optimization problems. A value lower than 1.0 means that $gap \times population\_size$ chromosomes will undergo selection, crossover, and mutation, with the offspring produced randomly inserted into a copy of the parent's generation, without replacement. Higher mutation rates are used to combat the premature loss of useful, but perhaps low fitness, genetic material. Together, these two parameters can work towards faster convergence by offering a better chance of survival to the more fit chromosomes. Of course, this is at the expense of off-line and on-line performance, the maintainence of diversity, and can lead to convergence to local minima. The meta-GA values for these parameters warrants further study. The

σ-scaling value difference between the meta-GA and the others appears to be insignificant given its small role. Finally, the meta-GA values for P_1 and P_2 are quite close to those used in these experiments and such small differences would have a minimal effect.

Given the excellent performance of the LNB genetic algorithm, these differences between the parameter settings used in this experiment and those provided by the meta-GA and the previous research were thought to have minimal impact.

5.6 Adaptation to modified problems

One of the goals of this thesis was to design a genetic algorithm that could be relied upon to produce very good solutions for the OCSTP even when faced with the kinds of changes to the parameters of the problem that might cause a heuristic to break down. When someone designs a new heuristic, a reliable yardstick is needed in order to evaluate the heuristic. Older heuristics might be employed for this purpose, but they too are subject to changes in the problem definition and it may not be known whether they are finding really good solutions or just the best ones known. In fact, it may be the case that the whole motivation for producing a new heuristic is because the old ones can no longer find good solutions because the problem has changed since they were first written [38]. A genetic algorithm that can adapt to changing problem parameters and still produce reliably good groups of solutions would be preferable. There are two general kinds of changes to a problem: changes to the data and changes to the problem goals. Genetic algorithms can be built that adapt well to both kinds of changes. The key to this success is that the underlying encoding remains the same. The fitness functions may need to be changed in the second case, but much of the work done to "tune" the genetic algorithm can be retained. It may be possible to adjust the new genetic algorithm's control parameters in order to improve its performance, but this may not be necessary. This section discusses two problems that have a common goal of finding a tree over the nodes of a graph and use the same biased-tree encoding. However, the first problem has very different input data from the previous experiments and the second problem has a completely different interpretation of a

technique	best solution
Star-Search heurustic	3.210×10^6
Local-Exchange heuristic	2.173×10^6
LNB genetic algorithm	2.183×10^6

Table 5.6: Adaptability to the distribution network problem.

tree solution.

5.6.1 Distribution network problem

When the problem parameters change, a heuristic must be reexamined to see if it will continue to produce good solutions. In the star-search heuristic, the quite reasonable assumption "one or two stars are good" was one of the rules used. Suppose that a different problem is presented: there is a set of special cities called "distribution points" scattered across the country. Other cities around them only have traffic requirements to and from their one distribution point city. These distribution point cities themselves have traffic with only one central city. Thus, if an OCSTP-like optimization is done given this constrained traffic matrix, the expected result would be a cluster of cities around each of the distribution points and the distribution points themselves connected to the center. In this new problem, the data has changed so that link costs encourage the use of several stars. Due to this change, the heuristic is no longer able to produce good solutions. The genetic algorithm, which makes use of no problem-specific knowledge, adapted well to the problem and continued to produce good solutions. The results from running both heuristics and the genetic algorithm on the modified problem are shown in table 5.6. The trees found by the LNB genetic algorithm, the star-search heuristic, and the local-exchange heuristic for the modified problem with $N = 24$ are shown in figures 5.6, 5.7, and 5.8, respectively.

As the results show, the genetic algorithm was able to adapt to the new problem and produce solutions 30% better than the star-search heuristic and within 1% of the solution found by the local-exchange heuristic which in this case can be proven to be optimal as follows:

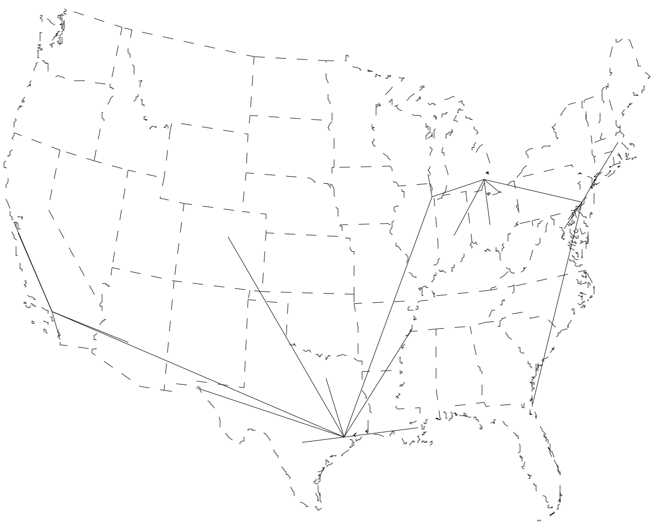


Figure 5.6: LNB GA network for modified problem with $N = 24$.

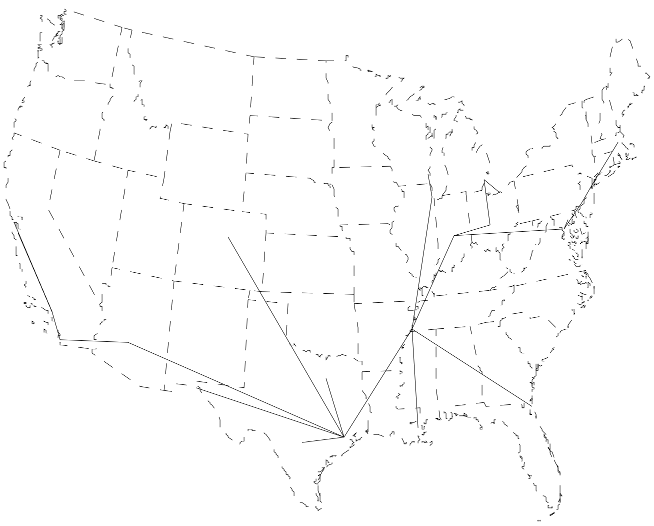


Figure 5.7: Star-search heuristic network for modified problem with $N = 24$.

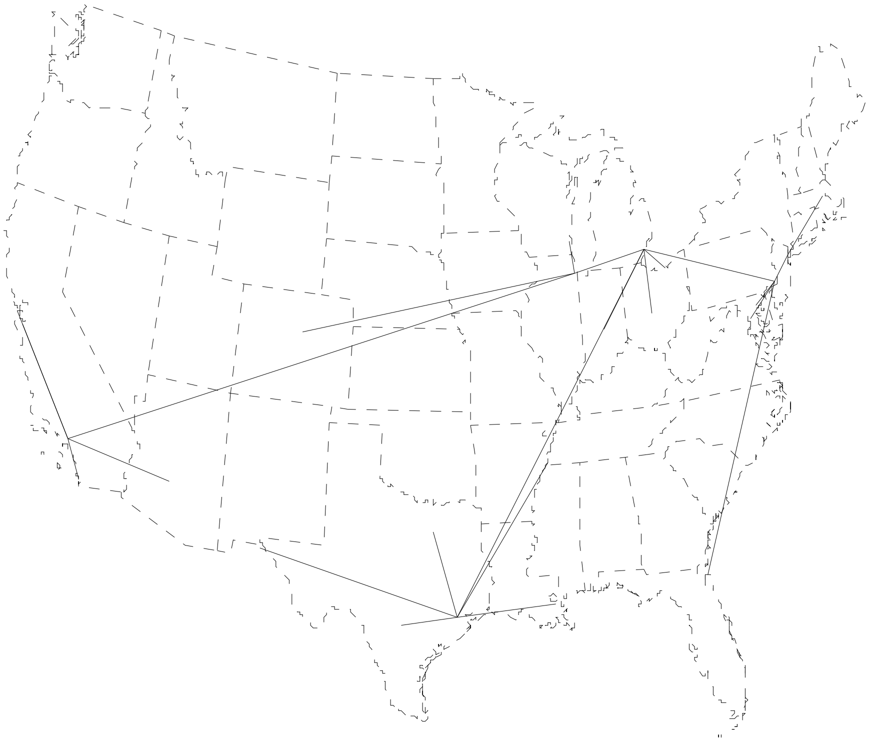


Figure 5.8: Local-exchange heuristic network (optimal) for modified problem with $N = 24$.

- Since all of the link costs in this experiment obey the triangle inequality and the "leaf" cities have traffic to exactly one distribution point city, their traffic must flow along the shortest path to its destination. The destination, in turn, must be the direct link between this leaf its distribution point city.

- The distribution point cities only have traffic to one other distribution point city. So, for the same reason as above, the direct link to that other city will be the shortest.

5.6.2 Minimum delay spanning tree problem

It would be convenient if a heuristic could be used to address problems other than the specific one for which it was designed. This is not a simple situation of changing the parameters of the problem. It is a more basic change to the goals of the problem and meaning of the solutions. The difficulty here is that good heuristics usually take advantage of one or more features of the problem itself in order to efficiently solve the problem. A carefully designed genetic algorithm, however, can sometimes adapt to such basic changes.

Problems whose solutions are trees are good candidates for testing this idea of problem adaptation with the genetic algorithm described in this thesis. One of these is the design of minimum delay spanning tree topologies for the interconnection of local area networks (LANs). As described earlier, a spanning tree is a tree that connects all the nodes of a graph (or network). Because it is a tree, there is exactly one path between each pair of nodes. In the original problem, the goal was to minimize a cost defined as the sum of the lengths of each path multiplied by the traffic requirement between the two nodes on either end of the path. In the problem described here, a collection of LANs is given, along with the traffic requirements between all pairs of LANs including traffic within a LAN. The goal is to find a spanning tree over these nodes that minimizes the average network delay for all of the traffic requirements. These LANs are connected with devices called bridges that allow traffic to flow from one LAN to another, and so the problem is to decide where to place the bridges subject to the minimization stated before. Both the LANs and the bridges have

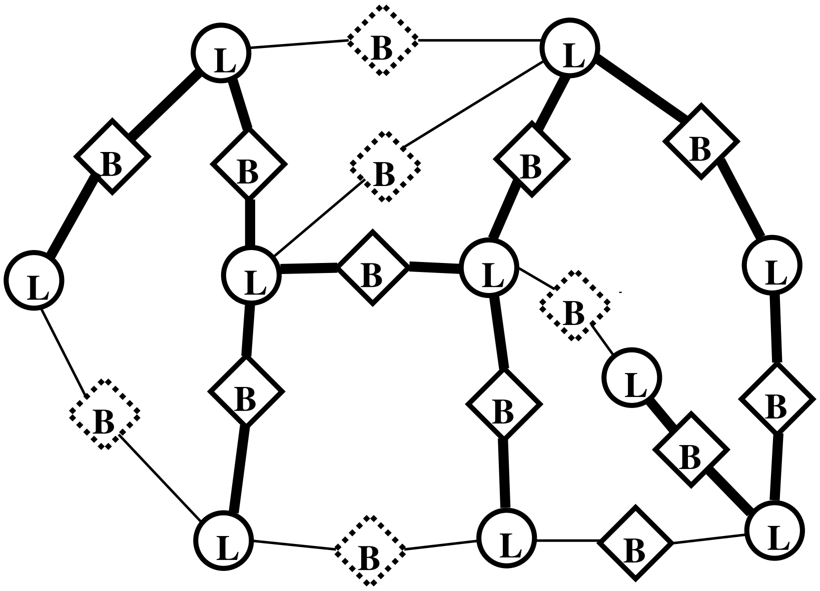


Figure 5.9: Typical LAN interconnection topology.

capacity limits. In the case of bridges, this limit governs how much traffic can pass from one LAN to another. In the case of the LANs, this limit determines how much traffic can come into and go out of the LAN. Similarly, both the LANs and the bridges introduce some delay into the path. Figure 5.9 shows a typical interconnected LAN. The capacities of the various components can be quickly overloaded if traffic from one node to another is allowed to travel over more than one path. To prevent this, a spanning tree of bridges between the LANs is usually employed. In figure 5.9 the bold links represent a spanning tree over the network. The dashed boxes represent bridges that are not participating in the spanning tree. They typically act as backup paths that can be used if something goes wrong with one of the active links. The goal is to select a spanning tree such that the capacities of LANs and bridges used by the tree are not exceeded and such that the total end-to-end delay for the traffic between all pairs of nodes is minimized.

In his dissertation [8], Ersoy investigated this problem using the simulated annealing local search technique [29]. He approximated the average network delay by limiting his model to delays due to LANs and bridges only. The objective function he used was

$$D = \frac{1}{\gamma}\left(\sum_{i=1}^{n}\frac{X\rho_i}{1-\rho_i} + \sum_{i=1}^{n}\sum_{j=1}^{n} x_{ij}\frac{X\rho_{ij}}{1-\rho_{ij}}\right)$$

where D is the average network delay, ρ_i is the utilization on LAN i, ρ_{ij} is the utilization on the bridge port from LAN i to LAN j, X is the mean number of packets in a batch, and γ is the total input flow into the network given by

$$\gamma = X\sum_{i=1}^{n}\sum_{j=1}^{n} t_{ij}$$

where the t_{ij} is the traffic requirement between LAN i and LAN j. In general, a specific simulated annealing process may be described by its neighborhood structure and its cooling schedule. In this case, the neighborhood structure used by the simulated annealing was defined as any two spanning tree topologies which have all the branches except one in common. The cooling schedule chosen was similar to one described in [1]. The overall complexity of the simulated annealing process was reported as $O(n^5)$.

Comparisons with Ersoy's work are made for several reasons:

- simulated annealing is another randomized search process like genetic algorithms,
- it demonstrates an adaptation of the genetic algorithm to a problem with both different data and a different fitness function,
- the problem provides an example of when the link biases of the chromosome should be used (i.e., $P_1 > 0$), and
- his work provided lower bounds for the average network delay.

For application to this problem, the fitness function of the LNB genetic algorithm was replaced by one that would evaluate the average network delay in the tree represented by the chromosome. The LNB encoding was used without modification, although the fitness function had to be modified to add up the delays in a given network. Due to the nature of this new problem, wherein the links themselves represent bridges which have capacity and delay, it became clear that link biases would

n	D_{LB}	D_{SA}	$\frac{D_{SA}-D_{LB}}{D_{LB}}$	D_{GA}	$\frac{D_{GA}-D_{LB}}{D_{LB}}$	$D_{H2'}$	$\frac{D_{H2'}-D_{LB}}{D_{LB}}$
6	5.42	6.41	18.3%	6.41	18.3%	6.41	18.3%
7	5.72	7.52	31.5%	7.52	31.5%	7.52	31.5%
10	6.20	7.79	25.7%	7.42	19.7%	7.60	22.6%
15	7.17	10.5	46.4%	10.6	47.8%	10.9	52.0%
20	8.90	13.7	53.8%	14.5	62.9%	13.8	55.1%
30	10.7	16.1	50.5%	18.0	68.2%	15.6	45.8%

Table 5.7: Minimum delay spanning tree problem lower bound comparisons.

n	D_{SA}	D_{GA}	$\frac{D_{GA}-D_{SA}}{D_{SA}}$	$D_{H2'}$	$\frac{D_{H2'}-D_{SA}}{D_{SA}}$
6	6.41	6.41	0%	6.41	0%
7	7.52	7.52	0%	7.52	0%
10	7.79	7.42	−4.75%	7.60	−2.4%
15	10.5	10.6	0.95%	10.9	3.8%
20	13.7	14.5	5.84%	13.8	0.7%
30	16.1	18.0	11.8%	15.6	−3.1%

Table 5.8: Direct comparison of GA and $H2'$ results to SA results.

be important. To enable the link biases, the P_1 parameter was set to 1.0 rather than the zero value it had for the OCSTP version of the genetic algorithm. All of the other parameters, population size, number of generations, etc., were kept at the values used for the OCSTP. In table 5.7, the average genetic algorithm results, D_{GA}, are shown with results from Ersoy's simulated annealing approach, D_{SA}, and results from a version of the local-exchange heuristic, $D_{H2'}$, of section 5.4 modified to find solutions for this problem. The lower bounds, D_{LB}, determined by Ersoy are also shown in the table. The values shown for $N = 6$ and $N = 7$ were proven by Ersoy to be the optimum through enumeration[1]. Based upon the large differences between the optimal results for $N = 6$ and $N = 7$, it would appear that the bounds are rather loose. Given this fact, table 5.8 contains another comparison of the genetic algorithm and local-exchange heuristic with the results of the simulated annealing approach.

The quality of the genetic algorithm results show that it was indeed able to adapt well to this quite different problem. More importantly, the genetic algorithm was

[1]The results for n=10, 20, and 30 were revised by Dr. Ersoy since completing his dissertation and were privately communicated to the author.

n	30,000 trials			50,000 trials		
	D_{GA}^{30K}	$\frac{D_{GA}^{30K}-D_{SA}}{D_{SA}}$	**gene bias**	D_{GA}^{50K}	$\frac{D_{GA}^{50K}-D_{SA}}{D_{SA}}$	**gene bias**
20	14.1	3%	66%	14.0	2%	66%
30	17.4	8%	72%	17.2	7%	73%

Table 5.9: Extended delay analysis runs for $n = 20$ and $n = 30$.

able to adapt to this problem without any retuning of its control parameters. It was observed that for the cases of $N = 20$ and 30, the genetic algorithm was still evolving improved solutions when its budget of $10,000$ trials was exhausted. Additional runs were made for these two cases with increased budgets with the average results and final population gene bias values shown in table 5.9. These extended runs resulted in improvements of only 1% and 2% for the 20 and 30 node problems, respectively. From this lack of significant improvement and the bias values of the final populations, it appears that larger numbers of trials would be of little interest.

Chapter 6

Conclusions and Future Work

The application of a genetic algorithm to a new problem is always a trailblazing effort. Whether the problem requires specialized crossover operators, non-traditional encoding alphabets, or tailored fitness scaling procedures, a reapplication of old ideas in a new way or some new innovation will be the key to success. Tree problems have been almost totally ignored by genetic algorithm researchers for lack of an effective chromosome encoding for trees. It is certainly not for want of an application that an effective tree encoding has not been found. The series of trials and errors that designing such an encoding requires that was described in this thesis are evidence of the fact that this is a hard problem. The identification of the key attributes that a good tree encoding should possess was an important beginning. An encoding that combines the standard technique of finding the minimal spanning tree with the new idea of having genes represent the desirability of a node or a link in the tree is a major contribution of this work. The new Link and Node Bias (LNB) encoding was shown to have all the desirable properties that were identified in this thesis with an important addition: it was completely general. Any problem involving trees can apply a genetic algorithm using the same LNB encoding with no changes because the genetic algorithm, by its very nature, needs little, if any, information about its search space to successfully perform its search. This capability was demonstrated through the application of the LNB genetic algorithm to the average network delay problem. Although quite different from the OCSTP, this delay problem was successfully addressed by a genetic algorithm using the LNB encoding. The results from this unchanged genetic algorithm compared quite favorably with those obtained using simulated annealing and with results produced by a modified version of one of the heuristics for the OCSTP. This adaptability is one of the key reasons that genetic algorithms can be so effective once they are designed and tested: genetic algorithms don't really care about the problem they're assigned.

Once an encoding for trees was designed, it had to be tested. For this, the Optimal Communications Spanning Tree Problem (OCSTP) was chosen. Since its origins with T. C. Hu in 1974 [23], good heuristics for the NP-complete OCSTP have eluded researchers. Although the literature is almost devoid of other than passing references to it, some existing heuristics have been found to perform reasonably well. After easily showing that the genetic algorithm was finding solutions to the OCSTP that were superior to a random search, head-to-head comparisons with existing OCSTP heuristics were performed. The goal of these comparisons was not so much to compete as to probe the consistency and reliability of the genetic algorithm's solutions. As described in chapter 5, the genetic algorithm reliably found solutions as good as or as much as 4.2% better than the heuristics, even when the heuristics were given more than an order of magnitude additional computation time. This ability to find reliably very good solutions to the OCSTP was another major contribution of this work.

Several times in this thesis it was pointed out that the overall goal was not to build a genetic algorithm that would compete with heuristics. Instead, the genetic algorithm approach can be best used in two ways:

- First, when no heuristics for a problem exist, a genetic algorithm, which needs only limited information about the problem space in its fitness function, can find solutions that would be unknown otherwise. Once some solutions are found, the heuristic designer can examine the genetic algorithm solutions in order to see what good solutions look like and then proceed to design a better heuristic.

- Second, when there are existing heuristics for a problem, but the rules of that problem have changed slightly, a genetic algorithm, which has little interest in the specific problem being addressed, can be used as a scale against which to measure the continuing effectiveness of the old heuristic when faced with the new data. Without a genetic algorithm, verification of heuristics can be as challenging as the designing of the heuristics themselves.

An encoding that can be generally applied to problems across several domains is highly desirable. Another contribution of this work is that the LNB encoding is such an encoding.

6.1 Future Work

Finding an encoding for trees and an effective genetic algorithm for the OCSTP and other problems was just the beginning of several avenues of investigation. One area that is quite appealing is to study the schemata at work in this encoding. The encoding was designed around the idea of the schemata arising within the bias values, but there may be other schemata that are responsible for the encoding's very good performance. A related question is how the values of P_1 and P_2 should be chosen for a given problem. For example, how will this encoding fare when faced with an OCSTP problem in which severe degree constraints (i.e. only degrees of 1 or 2) are placed on the nodes?

Another interesting effort would be to attempt to characterize the schemata that arise when using the LNB encoding. Given the good performance of the LNB genetic algorithm, short, low-order schemata are predicted by the theory. However, it is not known whether the individual b_i, b_j, and b_{ij} genes on the chromosome represent the only places where schemata arise.

Recent work by Orvosh and Davis [32] indicates that when an encoding allows invalid chromosomes to be produced by mating, it isn't always best to "repair" them. They suggest that this reparation process can result in the loss of important genetic material and, thus, reduce the efficiency of the search. Their proposed alternative is another genetic algorithm parameter which specifies the probability that an invalid chromosome is repaired. Their research has shown good success with $P_{repair} \approx 5\%$. This new idea might allow more effective use of one of the other, more fragile, encodings which might lead to encodings more effective than the LNB.

Appendix A

GA Fitness Function for the OCSTP

```
#include <stdio.h>
#include <stdlib.h>
#include <values.h>
#include <assert.h>
#include <math.h>

/*******************************************************************

   The chromosome has two sections:
     b(ij)     - biases on the link costs
     b(k)      - biases on the endpoints of a link
   The size in bits of these biases must be the same. There will be
      a total of N(N-1)/2+N of these biases.
   The GA takes four parameters:
        OutputFileExtension
        InputFile
        P1
        P2
```

```
  The distances (costs) used are biased by the chromosome contents in
  this way:
      bdist(i,j) = dist(i,j) + P1*b(ij) + P2*(b(i)+b(j))
  These biased distances are passed to Prim's algorithm which returns
  a pred vector. This pred vector is in turn passed to the standard
  evaluator to determine the fitness.

 ********************************************************************/

#define   N_MAX   24
#define   K_BITS  8

double    bga24();

/* Here's the special string for the Genesis control parameters */
/* GAeval bga24     8uib300      */
/*
 * where:
 *      bga24   is the name of the evaluation function
 *      8              is the size in bits of each parameter (must match K_BITS)
 *      u              indicates parameters are unsigned
 *      i              indicates parameters are integers
 *      b          encode parameters as binary (not grey-code)
 *      300      N(N-1)/2 + N parameters
 ********************************************************************/

void GenerateNet(int);
void Prim( int , int* );

int     preds[N_MAX];
int     req[N_MAX][N_MAX];
```

```
int     costs[N_MAX][N_MAX];
double  dists[N_MAX][N_MAX],
        bdists[N_MAX][N_MAX];
int     nodes[N_MAX][2];
int     link_indices[N_MAX][N_MAX];

/*------ evaluation function ------*/

double bga24 (  unsigned int *chromosome )
{
   register int    i, j
          int    k, p, maxcost;
   static   double sum=0.0;
   static   int    InTree[N_MAX];
   static   int    nTreeNodes, TreeNodes[N_MAX];
   static   unsigned long  numcalls=0;
   static   int    n, numlinks;
   unsigned int *pbij, *pbi;
   static double   Bij_factor, Bi_factor, Bj_factor, p1, p2;

   /* If this is the first entry, we have to generate   */
   /* (x,y) pairs for the nodes, and calc the distances */
   /* between them.                                     */

   if (numcalls++ == 0L)
   {
      n = N_MAX;
      GenerateNet(n);

      /* find the max link costs to precalculate the */
      /* bias constants and extract P1 and P2 */
```

```
    maxcost = 0;
    for (i=0; i<n; ++i)
        for (j=0; j<n; ++j)
        {
            if (costs[i][j] > maxcost)
                maxcost = costs[i][j];
        }

    p1 = atof(GArgv[2]);
    p2 = atof(GArgv[3]);
    numlinks = (n*(n-1))/2;

    /* the Bij factor is p1 times max_link_cost / 2**k */
    Bij_factor = p1 * maxcost / (1 << K_BITS);

    /* the Bi & Bj factors are p2 times max_link_cost / 2**k */
    Bi_factor = Bj_factor = p2 * maxcost / (1 << K_BITS);

    /* precalculate the mapping from ij into the pbij part  */
    /* of the chromosome (can't think of a formula to do it)*/
    k=0;
    for (i=0; i<n; ++i)
        for (j=i+1; j<n; ++j)
            link_indices[i][j] = k++;
}

/* produce the biased distance (cost) matrix */
/* using the link and node biases from the   */
/* chromosome and parameters P1 and P2.      */
pbij = chromosome;
```

```
pbi  = chromosome + numlinks;

for (i=0; i<n; ++i)
    for (j=i; j<n; ++j)
    {
        if (i==j)
            bdists[i][j] = bdists[j][i] = 0;
        else
            bdists[i][j] =
                bdists[j][i] =
                    costs[i][j] + pbij[link_indices[i][j]]*Bij_factor
                                + pbi[i]*Bi_factor
                                + pbi[j]*Bj_factor;
    }

/* recover the tree by running Prim to produce preds */
Prim(n, preds);

/* calculate the distance (cost) of     */
/* the path between all pairs of nodes. */
for ( i = 0 ; i < N_MAX ; i++ )
{
    if ( i == preds[i] )
    {
        InTree[i] = 1;
        TreeNodes[0] = i;
        dists[i][i] = 0L;
    }
    else
        InTree[i] = 0;
}
```

```
    nTreeNodes = 1;
    while( nTreeNodes < N_MAX )
    {
        for ( i = 0 ; i < N_MAX ; i++ )
        {
            p = preds[i];
            if ( InTree[i]==0 && InTree[p]==1 )
            {
                InTree[i] = 1;
                for ( j = 0 ; j < nTreeNodes ; j++ )
                {
                    k = TreeNodes[j];
                    dists[i][k] = dists[k][i] = costs[i][p] + dists[p][k];
                }
                dists[i][i] = 0L;
                TreeNodes[nTreeNodes++] = i;
            }
        }
    }

    /* finally, find  sum of dists (costs/capacity) times requirements */
    sum = 0L;
    for ( i = 0 ; i < N_MAX ; i++ )
        for ( j = 0 ; j < N_MAX ; j++ )
            sum += (dists[i][j] * req[i][j]);

    return( sum );
}
```

Appendix B

Input Data for OCSTP Experiments

The input data for these experiments was generated using the tools described in [2]. While there always seems to be a source of inter-city link costs or city locations, obtaining requirements matrices is always difficult. The referenced report describes a tool that will produce requirements parametrically. Specifically, the tool takes as input a list of up to 100 cities in the U.S. that are to be in the network, a specified link type (e.g. T1), a table of the cost of such a link between all pairs of cities, and some parameters which will guide the generation of the requirements. The requirements are based upon such information as the population of each city, the distance between cities, and other factors, all of which have control parameters which determine the influence of each kind of information. For the experiments described in this thesis, the inter-city requirements were generated as being inversely proportional to the inter-city distances. The link costs used were from a tariff database.

The input files that are provided here are for the OCSTP problem with $N = 6$ and $N = 12$, and for the distributed network problem. Due to their bulk, the input files for the 24, 47, and 98 node OCSTP problems are not included here. They may be obtained by sending a request via email to *cpalmer@watson.ibm.com*. The input files for the 6 and 12 node OCSTP and 24 node distributed network problems are shown on the following pages.

B.1 6 Node OCSTP Input Data

B.1.1 Cost matrix

	L.A.	CHI	DET	N.Y.	PHIL	HOU
L.A.	0	16661	18083	21561	21099	13461
CHI	16661	0	5658	9194	8797	10440
DET	18083	5658	0	7230	6899	11340
N.Y.	21561	9194	7230	0	4300	13730
PHIL	21099	8797	6899	4300	0	13130
HOU	13461	10440	11340	13730	13130	0

B.1.2 Requirements matrix

	L.A.	CHI	DET	N.Y.	PHIL	HOU
L.A.	0	1	1	1	1	2
CHI	1	0	10	3	4	3
DET	1	10	0	5	6	2
N.Y.	1	3	5	0	31	2
PHIL	1	4	6	31	0	2
HOU	2	3	2	2	2	0

B.2 12 Node OCSTP Input Data

B.2.1 Cost matrix

	PHNX	L.A.	S.D.	S.F.	CHI	BALT	DET	N.Y.	PHIL	DAL	HOU	SANAN
PHNX	0	6490	5903	8484	14561	18359	15976	19360	18867	10090	10883	9665
L.A.	6490	0	4523	6256	16661	20618	18083	21561	21099	12639	13461	12236
S.D.	5903	4523	0	6908	16414	20292	17829	21263	20787	12073	12802	11540
S.F.	8484	6256	6908	0	17328	21452	18714	22286	21874	14234	15259	14136
CHI	14561	16661	16414	17328	0	8425	5658	9194	8797	9603	10440	11237
BALT	18359	20618	20292	21452	8425	0	6621	5067	4439	12385	12526	13722
DET	15976	18083	17829	18714	5658	6621	0	7230	6899	10720	11340	12297
N.Y.	19360	21561	21263	22286	9194	5067	7230	0	4300	13531	13730	14912
PHIL	18867	21099	20787	21874	8797	4439	6899	4300	0	12967	13130	14319
DAL	10090	12639	12073	14234	9603	12385	10720	13531	12967	0	4888	5076
HOU	10883	13461	12802	15259	10440	12526	11340	13730	13130	4888	0	4478
SANAN	9665	12236	11540	14136	11237	13722	12297	14912	14319	5076	4478	0

B.2.2 Requirements matrix

	PHNX	L.A.	S.D.	S.F.	CHI	BALT	DET	N.Y.	PHIL	DAL	HOU	SANAN
PHNX	0	7	8	4	2	1	2	1	1	3	3	3
L.A.	7	0	25	7	1	1	1	1	1	2	2	2
S.D.	8	25	0	6	1	1	1	1	1	2	2	2
S.F.	4	7	6	0	1	1	1	1	1	2	2	2
CHI	2	1	1	1	0	4	11	4	4	3	3	2
BALT	1	1	1	1	4	0	6	15	29	2	2	2
DET	2	1	1	1	11	6	0	5	6	3	2	2
N.Y.	1	1	1	1	4	15	5	0	33	2	2	2
PHIL	1	1	1	1	4	29	6	33	0	2	2	2
DAL	3	2	2	2	3	2	3	2	2	0	11	10
HOU	3	2	2	2	3	2	2	2	2	11	0	14
SANAN	3	2	2	2	2	2	2	2	2	10	14	0

B.3 24 Node Distribution Network Problem Input Data

B.3.1 Cost matrix

part 1

	PHNX	L.A.	S.D.	S.F.	S.J.	DEN	WDC	JACK	CHI	IND	N.O.	BOS
PHNX	0	649	590	848	824	808	1810	1700	1456	1489	1336	2059
L.A.	649	0	453	626	599	980	2034	1955	1661	1717	1595	2275
S.D.	590	453	0	698	664	967	2003	1897	1644	1688	1539	2241
S.F.	848	626	698	0	282	1056	2126	2106	1738	1795	1758	2339
S.J.	824	599	664	282	0	1046	2114	2088	1720	1782	1742	2323
DEN	808	980	967	1056	1046	0	1451	1456	1068	1129	1154	1674
WDC	1810	2034	2003	2126	2114	1451	0	855	829	754	1064	675
JACK	1700	1955	1897	2106	2088	1456	855	0	1036	929	752	1144
CHI	1456	1661	1644	1738	1720	1068	829	1036	0	549	995	1030
IND	1489	1717	1688	1795	1782	1129	754	929	549	0	919	993
N.O.	1336	1595	1539	1758	1742	1154	1064	752	995	919	0	1373
BOS	2059	2275	2241	2339	2323	1674	675	1144	1030	993	1373	0

part 2

	PHNX	L.A.	S.D.	S.F.	S.J.	DEN	WDC	JACK	CHI	IND	N.O.	BOS
BALT	1835	2068	2022	2142	2136	1471	387	888	845	771	1100	655
DET	1597	1803	1789	1874	1860	1203	657	983	568	567	1047	823
N.Y.	1936	2151	2123	2226	2228	1561	524	1005	914	875	1226	530
CLEVE	1674	1890	1864	1963	1955	1295	615	979	651	618	1074	813
COL	1588	1813	1784	1892	1887	1223	605	870	600	528	958	8571
PHIL	1886	2109	2077	2184	2176	1515	463	942	877	821	1161	585
MEMP	1289	1537	1483	1663	1643	1005	914	817	736	667	621	1203
DAL	1009	1269	1203	1424	1404	833	1206	1037	963	932	675	1494
ELPAS	602	852	791	1051	1031	741	1582	1424	1264	1280	1058	1851
HOU	1088	1341	1282	1529	1508	975	1225	954	1040	995	577	1511
SANAN	966	1226	1150	1416	1390	920	1348	1092	1127	1089	714	1631
MILW	1461	1664	1644	1715	1719	1065	853	1097	475	592	1056	1038

Cost matrix, part 3

	BALT	DET	N.Y.	CLEVE	COL	PHIL	MEMP	DAL	ELPAS	HOU	SANAN	MILW
PHNX	1835	1597	1936	1674	1588	1886	1289	1009	602	1088	966	1461
L.A.	2068	1803	2151	1890	1813	2109	1537	1269	852	1341	1226	1664
S.D.	2022	1789	2123	1864	1784	2077	1483	1203	791	1282	1150	1644
S.F.	2142	1874	2226	1963	1892	2184	1663	1424	1051	1529	1416	1715
S.J.	2136	1860	2228	1955	1887	2176	1643	1404	1031	1508	1390	1719
DEN	1471	1203	1561	1295	1223	1515	1005	833	741	975	920	1065
WDC	387	657	524	615	605	463	914	1206	1582	1225	1348	853
JACK	888	983	1005	979	870	942	817	1037	1424	954	1092	1097
CHI	845	568	914	651	600	877	736	963	1264	1040	1127	475
IND	771	567	875	618	528	821	667	932	1280	995	1089	592
N.O.	1100	1047	1226	1074	958	1161	621	675	1058	577	714	1056
BOS	655	823	530	813	8571	585	1203	1494	1851	1511	1631	1038

part 4

	BALT	DET	N.Y.	CLEVE	COL	PHIL	MEMP	DAL	ELPAS	HOU	SANAN	MILW
BALT	0	661	507	633	627	449	940	1235	1617	1256	1372	862
DET	661	0	720	463	491	689	812	1070	1404	1130	1227	576
N.Y.	507	720	0	702	722	430	1068	1351	1727	1370	1492	934
CLEVE	633	463	702	0	493	660	858	1123	1472	1170	1279	663
COL	627	491	722	493	0	671	739	1019	1377	1060	1162	635
PHIL	449	689	430	660	671	0	1000	1297	1663	1310	1439	896
MEMP	940	812	1068	858	739	1000	0	652	1044	680	795	792
DAL	1235	1070	1351	1123	1019	1297	652	0	744	488	506	990
ELPAS	1617	1404	1727	1472	1377	1663	1044	744	0	809	674	1280
HOU	1256	1130	1370	1170	1060	1310	680	488	809	0	448	1084
SANAN	1372	1227	1492	1279	1162	1439	795	506	674	448	0	1166
MILW	862	576	934	663	635	896	792	990	1280	1084	1166	0

B.3.2 Requirements matrix

part 1

	PHNX	L.A.	S.D.	S.F.	S.J.	DEN	WDC	JACK	CHI	IND	N.O.	BOS
PHNX	0	100	0	0	0	0	0	0	0	0	0	0
L.A.	100	0	100	100	100	0	0	0	2	0	0	0
S.D.	0	100	0	0	0	0	0	0	0	0	0	0
S.F.	0	100	0	0	0	0	0	0	0	0	0	0
S.J.	0	100	0	0	0	0	0	0	0	0	0	0
DEN	0	0	0	0	0	0	0	0	100	0	0	0
WDC	0	0	0	0	0	0	0	0	0	0	0	0
JACK	0	0	0	0	0	0	0	0	0	0	0	0
CHI	0	2	0	0	0	100	0	0	0	0	0	0
IND	0	0	0	0	0	0	0	0	0	0	0	0
N.O.	0	0	0	0	0	0	0	0	0	0	0	0
BOS	0	0	0	0	0	0	0	0	0	0	0	0

part 2

	PHNX	L.A.	S.D.	S.F.	S.J.	DEN	WDC	JACK	CHI	IND	N.O.	BOS
BALT	0	0	0	0	0	0	0	0	0	0	0	0
DET	0	0	0	0	0	0	0	0	2	100	0	0
N.Y.	0	0	0	0	0	0	0	0	0	0	0	100
CLEVE	0	0	0	0	0	0	0	0	0	0	0	0
COL	0	0	0	0	0	0	0	0	0	0	0	0
PHIL	0	0	0	0	0	0	100	100	0	0	0	0
MEMP	0	0	0	0	0	0	0	0	0	0	0	0
DAL	0	0	0	0	0	0	0	0	0	0	0	0
ELPAS	0	0	0	0	0	0	0	0	0	0	0	0
HOU	0	0	0	0	0	0	0	0	0	0	100	0
SANAN	0	0	0	0	0	0	0	0	0	0	0	0
MILW	0	0	0	0	0	0	0	0	100	0	0	0

Requirements matrix, part 3

	BALT	DET	N.Y.	CLEVE	COL	PHIL	MEMP	DAL	ELPAS	HOU	SANAN	MILW
PHNX	0	0	0	0	0	0	0	0	0	0	0	0
L.A.	0	0	0	0	0	0	0	0	0	0	0	0
S.D.	0	0	0	0	0	0	0	0	0	0	0	0
S.F.	0	0	0	0	0	0	0	0	0	0	0	0
S.J.	0	0	0	0	0	0	0	0	0	0	0	0
DEN	0	0	0	0	0	0	0	0	0	0	0	0
WDC	0	0	0	0	0	100	0	0	0	0	0	0
JACK	0	0	0	0	0	100	0	0	0	0	0	0
CHI	0	2	0	0	0	0	0	0	0	0	0	100
IND	0	100	0	0	0	0	0	0	0	0	0	0
N.O.	0	0	0	0	0	0	0	0	0	100	0	0
BOS	0	0	100	0	0	0	0	0	0	0	0	0

part 4

	BALT	DET	N.Y.	CLEVE	COL	PHIL	MEMP	DAL	ELPAS	HOU	SANAN	MILW
BALT	0	0	0	0	0	100	0	0	0	0	0	0
DET	0	0	0	100	100	2	0	0	0	2	0	0
N.Y.	0	0	0	0	0	2	0	0	0	0	0	0
CLEVE	0	100	0	0	0	0	0	0	0	0	0	0
COL	0	100	0	0	0	0	0	0	0	0	0	0
PHIL	100	2	2	0	0	0	0	0	0	0	0	0
MEMP	0	0	0	0	0	0	0	0	0	100	0	0
DAL	0	0	0	0	0	0	0	0	0	100	0	0
ELPAS	0	0	0	0	0	0	0	0	0	100	0	0
HOU	0	2	0	0	0	0	100	100	100	0	100	0
SANAN	0	0	0	0	0	0	0	0	0	100	0	0
MILW	0	0	0	0	0	0	0	0	0	0	0	0

Appendix C

GA Fitness Function for Delay Analysis

```
#include <stdio.h>
#include <stdlib.h>
#include <values.h>
#include <assert.h>
#include <math.h>

/*************************************************************************

  The chromosome has two sections:
   b(ij)     - biases on the link costs
   b(k)      - biases on the endpoints of a link
  The size in bits of these biases must be the same. There will be
    a total of N(N-1)/2+N of these biases.
  The GA takes four parameters:
      OutputFileExtension
      InputFile
      P1
      P2
```

```
  The distances (costs) used are biased by the chromosome contents in
  this way:
       bdist(i,j) = dist(i,j) + P1*b(ij) + P2*(b(i)+b(j))
  These biased distances are passed to Prim's algorithm which returns
  a pred vector. This pred vector is used to evaluate the average
  network delay for this tree.

 ***************************************************************************/

#define   N_MAX    20
#define   K_BITS   8

double    bga20();

/* Here's the special string for the Genesis control parameters */
/* GAeval bga20       8uib210       */
/*
 * where:
 *      bga20   is the name of the evaluation function
 *      8                 is the size in bits of each parameter (must match K_BITS)
 *      u                 indicates parameters are unsigned
 *      i                 indicates parameters are integers
 *      b            encode parameters as binary (not grey-code)
 *      210       N(N-1)/2 + N parameters
 ***************************************************************************/

void    GenerateNet(int);
void    Prim( int , int * );

int     preds[N_MAX];
int     req[N_MAX][N_MAX];
```

```
int     lreq[N_MAX][N_MAX];
int     costs[N_MAX][N_MAX];
double  dists[N_MAX][N_MAX], bdists[N_MAX][N_MAX];
int     nodes[N_MAX][2];
int     link_indices[N_MAX][N_MAX];

/*------ evaluation function -------*/

double bga20 ( unsigned int *chromosome )
{
   register int    i, j;
   int    k, p, maxcost;
   static   double sum=0.0;
   static   int    InTree[N_MAX];
   static   int    nTreeNodes, TreeNodes[N_MAX];
   static   unsigned long  numcalls=0;
   static   int    n, numlinks;
   unsigned int *pbij, *pbi;
   static   double   Bij_factor, Bi_factor, Bj_factor, p1, p2;
   static   double   gamma;
   int      ni, pni, nflow, eflow1, eflow2;
   double   rho;

   sum  = 0.0;


   /* If this is the first entry, we have to generate   */
   /* (x,y) pairs for the nodes, and calc the distances */
   /* between them.                                     */

   if (numcalls++ == 0L)
```

```
{
    n = N_MAX;
    GenerateNet(N_MAX);
    gamma = 0.0;
    for ( i = 0 ; i < N_MAX ; i++ )
    {
        for ( j = 0 ; j < N_MAX ; j++ )
        {
            gamma += 0.1 * req[i][j];
        }
    }

    /* find the max link costs to precalculate the */
    /* bias constants and extract P1 and P2 */

    maxcost = 0;
    for (i=0; i<n; ++i)
        for (j=0; j<n; ++j)
        {
            if (costs[i][j] > maxcost)
                maxcost = costs[i][j];
        }

    p1 = atof(GArgv[2]);
    p2 = atof(GArgv[3]);

    numlinks = (n*(n-1))/2;

    /* the Bij factor is p1 times max_link_cost / 2**k */
    Bij_factor = p1 * maxcost / (1 << K_BITS);
```

```
    /* the Bi & Bj factors are p2 times max_link_cost / 2**k */
    Bi_factor = Bj_factor = p2 * maxcost / (1 << K_BITS);

    /* precalculate the mapping from ij into the pbij part  */
    /* of the chromosome (can't think of a formula to do it)*/
    k=0;
    for (i=0; i<n; ++i)
        for (j=i+1; j<n; ++j)
            link_indices[i][j] = k++;
}

/* produce the biased distance (cost) matrix */
/* using the link and node biases from the   */
/* chromosome and parameters P1 and P2.      */
pbij = chromosome;
pbi  = chromosome + numlinks;
for (i=0; i<n; ++i)
    for (j=i; j<n; ++j)
    {
        if (i==j)
            bdists[i][j] = bdists[j][i] = 0;
        else
            bdists[i][j] =
                bdists[j][i] =
                    costs[i][j] + pbij[link_indices[i][j]]*Bij_factor
                                + pbi[i]*Bi_factor
                                + pbi[j]*Bj_factor;
    }

/* produce the preds */
Prim(n, preds);
```

```
/* Finally, we calculate the sum of the shortest */
/* paths and return that sum as the fitness.     */


/* calculate the distance (cost) of     */
/* the path between all pairs of nodes. */
for ( i = 0 ; i < N_MAX ; i++ )
{
    if ( i == preds[i] )
    {
        InTree[i] = 1;
        TreeNodes[0] = i;
        dists[i][i] = 0L;
    }
    else
        InTree[i] = 0;
}

nTreeNodes = 1;
while( nTreeNodes < N_MAX )
{
    for ( i = 0 ; i < N_MAX ; i++ )
    {
        p = preds[i];
        if ( InTree[i]==0 && InTree[p]==1 )
        {
            InTree[i] = 1;
            TreeNodes[nTreeNodes++] = i;
        }
    }
```

```
}

sum = 0L;
for ( i = 0 ; i < N_MAX ; i++ )
{
    for ( j = 0 ; j < N_MAX ; j++ )
    {
        lreq[i][j] = req[i][j];
    }
}

for ( i = 0 ; i < N_MAX ; i++ )
{
    ni = TreeNodes[nTreeNodes−1−i];
    pni = preds[ni];
    nflow = 0;
    eflow1 = 0;
    eflow2 = 0;
    for ( j = 0 ; j < N_MAX ; j++ )
    {
        if ( ni == j )
        {
            nflow += lreq[ni][j];
            lreq[ni][j] = 0;
        }
        else
        {
            nflow += lreq[ni][j] + lreq[j][ni];
            if ( ni ≠ pni )
            {
                eflow1 += lreq[ni][j];
```

```
                lreq[pni][j] += lreq[ni][j];
                eflow2 += lreq[j][ni];
                lreq[j][pni] += lreq[j][ni];
            }
            lreq[ni][j] = 0;
            lreq[j][ni] = 0;
        }
    }
    rho = nflow / 8138.0;
    if ((1-rho) < 1e-20)
    {
        sum = 10e7;
        break;
    }
    sum += rho / ( 1 - rho );
    if (rho < 0.0 || sum ≤ 0.0)
    {
        sum = 10e7;
        break;
    }
    rho = eflow1 / 7500.0;
    if ((1-rho) < 1e-20)
    {
        sum = 10e7;
        break;
    }
    sum += rho / ( 1 - rho );
    if (rho < 0.0 || sum < 0.0)
    {
        sum = 10e7;
        break;
```

```
        }
        rho = eflow2 / 7500.0;
        if ((1-rho) < 1e-20)
        {
            sum = 10e7;
            break;
        }
        sum += rho / ( 1 - rho );
        if (rho < 0.0 || sum < 0.0)
        {
            sum = 10e7;
            break;
        }
        if (sum > 10e7)
        {
            sum = 10e7;
            break;
        }
    }
    sum = (sum /gamma) * 1000.0 ;

    if (sum < 0.0 || sum > 10e7)
    {
        sum = 10e7;
    }
    return( sum );
}
```

Appendix D

Input Data for Delay Analysis Experiments

The input data used for the minimum delay spanning tree problems were reproduced in the required format based upon Dr. Ersoy's work [8] and were used with his permission. The input files contain a list of LANs, the inter-LAN traffic requirements, and a constant cost for each bridge.

Due to their bulk, the input files for the 15, 20, and 30 node minimum delay spanning tree problems are not included here. They may be obtained by sending a request via email to *cpalmer@watson.ibm.com*. The input file for the 6, 7, and 10 node problems are shown on the following pages.

D.1 6 Node Minimum Delay Spanning Tree Problem Input Data

D.1.1 Cost matrix

	N1	N2	N3	N4	N5	N6
N1	0	100	100	100	100	100
N2	100	0	100	100	100	100
N3	100	100	0	100	100	100
N4	100	100	100	0	100	100
N5	100	100	100	100	0	100
N6	100	100	100	100	100	0

D.1.2 Requirements matrix

	N1	N2	N3	N4	N5	N6
N1	0	208	160	153	93	29
N2	152	0	104	333	118	163
N3	235	220	0	183	54	182
N4	184	68	131	0	132	9
N5	232	263	197	50	0	41
N6	214	119	46	192	86	0

D.2 7 Node Minimum Delay Spanning Tree Problem Input Data

D.2.1 Cost matrix

	N1	N2	N3	N4	N5	N6	N7
N1	0	100	100	100	100	100	100
N2	100	0	100	100	100	100	100
N3	100	100	0	100	100	100	100
N4	100	100	100	0	100	100	100
N5	100	100	100	100	0	100	100
N6	100	100	100	100	100	0	100
N7	100	100	100	100	100	100	0

D.2.2 Requirements matrix

	N1	N2	N3	N4	N5	N6	N7
N1	0	401	55	127	93	117	190
N2	96	0	14	147	123	186	186
N3	39	156	0	175	117	277	146
N4	33	187	160	0	191	173	113
N5	125	120	129	144	0	14	40
N6	22	99	182	147	55	0	285
N7	334	270	413	236	354	386	0

D.3 10 Node Minimum Delay Spanning Tree Problem Input Data

D.3.1 Cost matrix

	N1	N2	N3	N4	N5	N6	N7	N8	N9	Na
N1	0	100	100	100	100	100	100	100	100	100
N2	100	0	100	100	100	100	100	100	100	100
N3	100	100	0	100	100	100	100	100	100	100
N4	100	100	100	0	100	100	100	100	100	100
N5	100	100	100	100	0	100	100	100	100	100
N6	100	100	100	100	100	0	100	100	100	100
N7	100	100	100	100	100	100	0	100	100	100
N8	100	100	100	100	100	100	100	0	100	100
N9	100	100	100	100	100	100	100	100	0	100
Na	100	100	100	100	100	100	100	100	100	0

D.3.2 Requirements matrix

	N1	N2	N3	N4	N5	N6	N7	N8	N9	Na
N1	0	101	67	64	39	12	63	43	5	49
N2	68	0	98	192	76	23	43	77	28	54
N3	55	4	0	97	76	182	21	17	89	49
N4	19	72	36	0	10	40	73	110	32	87
N5	48	76	53	15	0	79	95	2	26	127
N6	25	31	53	99	83	0	189	153	214	166
N7	99	198	204	166	191	176	0	52	52	46
N8	3	97	40	120	45	52	90	0	35	63
N9	19	12	70	69	55	184	87	14	0	48
Na	81	83	95	87	41	99	93	174	76	0

Bibliography

[1] E. Aarts and J. Korst. *Simulated annealing and boltzmann machines.* Wiley and Sons, 1989.

[2] R. Cahn, P. Kermani, A. Kershenbaum, and D. Milch. A library of network benchmarks. Unpublished report, Network Design Tools Group, IBM T. J. Watson Research Center, 1990.

[3] L. A. Cox, L. Davis, and Y. Qiu. Dynamic anticipatory routing in circuit-switched telecommunications networks. In L. Davis, editor, *Handbook of Genetic Algorithms*, pages 124–143. Van Nostrand Reinhold, New York, NY, 1991.

[4] L. Davis. *Genetic Algorithms and Simulated Annealing.* Research Notes in Artificial Intelligence. Morgan Kaufmann, Los Altos, CA, 1987.

[5] K. A. DeJong. *Analysis of the Behavior of a class of genetic adaptive systems.* PhD thesis, Department of Computer and Communications Sciences, Univ. of Michigan, 1975.

[6] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[7] R. Dionne and M. Florian. Exact and approximate algorithms for optimal network design. *Networks*, 9:39–59, 1979.

[8] Cem Ersoy. *Topological Design of Interconnected Local and Metropolitan Area Networks.* PhD thesis, Polytechnic University, Electrical Engineering Department, Brooklyn, New York, 1992.

[9] S. Forrest. Documentation for prisoners dilemma and norms programs that use the genetic algorithm. Unpublished manuscript, University of Michagan, Ann Arbor, 1985.

[10] S. Forrest, editor. *Fifth International Conference on Genetic Algorithms*, University of Illinois at Urbana-Champaign, 1993. Morgan Kaufmann.

[11] M. R. Garey and D. S. Johnson. *Computers and Intractibility.* W. H. Freeman and Company, New York, NY, 1979.

[12] B. Gavish. Formulations and algorithms for the capacitated minimal directed tree problem. *Journal of the ACM*, 30(1):118–132, 1983.

[13] A. Gibbons. *Algorithmic Graph Theory.* Cambridge University Press, New York, NY, 1985.

[14] D. Glover. Solving a complex keyboard configuration problem through generalized adaptive search. In L. Davis, editor, *Genetic Algorithms and Simulated Annealing*, pages 12–31. Morgan Kaufmann, Los Altos, CA, 1987.

[15] D. E. Goldberg. *Genetic algorithms in search, optimization, and machine learning.* Addison-Wesley, Reading, MA, 1989.

[16] R. E. Gomory and T. C. Hu. Multi-terminal network flows. *SIAM Journal on Applied Math*, 9:551–570, 1961.

[17] J. J. Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, 16(1):122–128, 1986.

[18] J. J. Grefenstette. Incorporating problem specific knowledge into genetic algorithms. In L. Davis, editor, *Genetic Algorithms and Simulated Annealing*, pages 42–60. Morgan Kaufmann, Los Altos, CA, 1987.

[19] J. J. Grefenstette. A user's guide to genesis 4.5. Technical report, Navy Center for Applied Research in Artificial Intelligence, Naval Research Laboratory, Washington, D.C. 20375-5000, 1987. The Genesis package is available from the author who may be reached via email at gref@aix.nrl.navy.mil.

[20] J. J. Grefenstette and J. M. Fitzpatrick. Genetic search with approximate function evaluations. In J. J. Grefenstette, editor, *Proceedings of the First International Conference on Genetic Algorithms*, pages 112–120, Carnegie-Mellon University, Pittsburgh, PA, July 24-26 1985. Lawrence Erlbaum Associates.

[21] J. H. Holland. Hierarchical descriptions of universal spaces and adaptive systems. Technical Report ORA Projects 01252 and 08226, University of Michigan, Department of Computer and Communication Sciences, Ann Arbor, MI, 1968.

[22] J. H. Holland. *Adaptation in Natural and Artificial Systems*. Univ. of Michigan Press, Ann Arbor, MI, 1975; reprinted by MIT Press, Cambridge, MA, 1992, 1992.

[23] T. C. Hu. Optimum communication spanning trees. *SIAM Journal on Computing*, 3(3):188–195, 1974.

[24] F. K. Hwang. On steiner minimal trees with rectilinear distance. *SIAM Journal of Applied Mathematics*, 30(1):104–114, 1976.

[25] F. K. Hwang. An o(n log n) algorithm for suboptimal rectilinear steiner trees. *IEEE Transactions on Circuits and Systems*, 26(1):75–77, 1979.

[26] D. S. Johnson, J. K. Lenstra, and A. H. G. Rinnooy Kan. The complexity of the network design problem. *Networks*, 8:279–285, 1978.

[27] K. Juliff. A multi-chromosome genetic algorithm for pallet loading. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 467–473, University of Illinois at Urbana-Champaign, July 17-21 1993. Morgan Kaufmann.

[28] B. A. Julstrom. A genetic algorithm for the rectilinear steiner problem. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 474–479, University of Illinois at Urbana-Champaign, July 17-21 1993. Morgan Kaufmann.

[29] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecci. Optimization by simulated annealing. *Science*, 220:671–680, May 1983.

[30] S. Lin. Effective use of heuristic algorithms in network design. *Proceedings of Symposia in Applied Mathematics*, 26:63–84, 1982.

[31] J. W. Moon. Various proofs of Cayley's formula for counting trees. In F. Harary, editor, *A Seminar on Graph Theory*, pages 70–78. Holt, Rinehart, and Winston, New York, NY, 1967.

[32] D. Orvosh and L. Davis. Shall we repair? Genetic algorithms, combinatorial optimization, and feasibility constraints. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, page 650, University of Illinois at Urbana-Champaign, July 17-21 1993. Morgan Kaufmann.

[33] C. C. Palmer and A. Kershenbaum. Representing trees in genetic algorithms. In *Proceedings of the First IEEE Conference on Evolutionary Computation*, Orlando, FL, June 21 - July 2 1994.

[34] C. C. Palmer and A. Kershenbaum. Two algorithms for finding optimal communications spanning trees. Technical Report RC19394, IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY, 1994.

[35] R. C. Prim. Shortest connection networks and some generalizations. *Bell Systems Technical Journal*, 36:1389–1401, 1957.

[36] G. J. E. Rawlins. *Foundations of Genetic Algorithms.* Morgan Kaufmann, San Mateo, CA, 1991.

[37] N. N. Schraudolph and J. J. Grefenstette. A user's guide to gaucsd 1.4. Technical Report CS92-249, UC San Diego, CSE Department, UC San Diego, La Jolla, CA 92093-0114, 1990.

[38] D. J. Stockton and Liam Quinn. Identifying economic order quantities using genetic algorithms. *International journal of operations and production management*, 13(11):92–103, 1993.

[39] D. Whitley, T. Starkweather, and D. Shaner. The traveling salesman and sequence scheduling: Quality solutions using genetic edge recombination. In L. Davis, editor, *Handbook of Genetic Algorithms*, pages 350–372. Van Nostrand Reinhold, New York, NY, 1991.