Optimisation of a Tree Structured Centralized Data Network using an Evolutionary Algorithm

by

Jeff Katzen

Submitted to the Department of Electrical Engineering
in partial fulfillment of the requirements for the degree of

M.Sc.(Elec Eng)

at the

UNIVERSITY OF CAPE TOWN

October 1997

Signature of Author .....
Department of Electrical Engineering
7 October 1997

Certified by .....
Mr Neco Ventura
Director of ATM Research Group
Thesis Supervisor

Accepted by .....
Prof. G. De Jager
Acting Head of Department

Acknowledgements

I would like to thank my supervisor, Mr. Neco Ventura, for all his guidance throughout my thesis.

I would also like to thank the following people :

- Professor Les Berry of the Royal Melbourne Institute of Technology (RMIT) in Australia for all his help and for providing me with his latest research material.
- Associate Professor John Greene for providing me with information relating to Evolutionary Algorithms.

Optimisation of a Tree Structured Centralized Data Network using an Evolutionary Algorithm

by

Jeff Katzen

Submitted to the Department of Electrical Engineering
on 7 October 1997, in partial fulfillment of the
requirements for the degree of
M.Sc.(Elec Eng)

Abstract

This thesis attempts to solve the problem of optimising the design of tree structured centralized data network using an Evolutionary Algorithm. A centralized data network is also known as a client-server network. In this type of network, the client, which is usually a terminal connected to the network, would send a request for information to the server. The server would then download the reply back to the client. An example of such a network would be a bank's ATM network. Each ATM machine would be a client and the central server would store information relating to all the bank's customers.

The idea was that once this was done the fitness function used in the above problem would be modified to suite the design of a network used to interconnect LANs that would also form a tree structure. Each of the nodes in this network would be a LAN connected to the network via a bridge or router.

Unfortunately the results obtained in attempting to optimise the topology of the centralized data network were very poor. A heuristic normally used to solve this problem outperformed the Evolutionary Algorithm on all the three counts that the comparison was performed. Therefore another method using an Evolutionary Algorithm that can optimise the network interconnecting LANs was introduced.

The first chapter in this thesis is an introduction to the thesis and all the terms and concepts that are used in it. The second chapter explains the heuristic used. The third chapter discusses what particular properties are needed by a coding scheme used in an Evolutionary Algorithm to solve this problem. It introduces a few alternatives that have been used in the past but do not meet all the requirements. Then it introduces the coding scheme that was used in this thesis and the fitness function used to evaluate each candidate solution. The next chapter tabulates the results and draws conclusions from these results. The final chapter discusses areas of future research possibilities.

There are also several appendices. The first introduces the Genetic Algorithm (GA) and discusses some hypotheses that attempt to explain why it is so successful at problem solving. The next appendix introduces Population Based Incremental Learning (PBIL). This is the Evolutionary Algorithm that is used in attempting to solve this problem. Appendix C explains a method of converting between real and binary numbers, this method is not used in this

thesis but is important to know when dealing with Evolutionary Algorithms that are only capable of manipulating binary values. The next two appendices discuss Prim's algorithm and Competitive Learning. Prim's algorithm is an MST algorithm that is used in the coding scheme. Competitive Learning is a classification technique that PBIL is partly based on. An explanation of each function used to implement the heuristic and PBIL is given in Appendix F. This is followed by a listing of the Matlab code of each function.

Thesis Supervisor: Mr Neco Ventura
Title: Director of ATM Research Group

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The advent of the information age has been brought about by recent advances in the digital computer which has had a major effect on every technological discipline. Perhaps the area which has been affected the most is communications, specifically digital communications.

In the early days of computing all resources were attached to a single mainframe. Any peripherals, such as printers, card readers or terminals, that were connected were all restricted to be in close proximity to the mainframe and acted as secondary slave devices. As technology advanced, facilities became more spread out. Peripherals were no longer confined to be near the mainframe, but were moved further and further away. This was the beginning of the computer network.

Since then, computer networks have come a long way. The Internet is one of the fastest growing tools used in information transfer and global communications. It will change the way we communicate, the way we entertain ourselves and the way we do business. With the increasing use of computers and the realisation of the importance of communication networks for the efficient operation of business, there has been a lot of research dedicated to the design of these networks.

The networks could encompass many fields of communication, including analogue telephone networks, GSM digital cellular networks, ISDN multimedia networks and wide-area and local-area computer networks. Network designers in all of these applications have a similar goal :

How to optimise the topology of the network so that the cost of transferring data between the nodes is a minimum. This cost refers to the economic cost of the network and can be calculated using the traffic sent between each node and the cost of transmitting data between each node. Further constraints on the topology could also be specified. The constraint could be that the network has to achieve a certain level of reliability or that the maximum delay between any two nodes in the network has to be below a certain limit.

The goal of this thesis is to find an Evolutionary Algorithm capable of optimising the topology of a tree structured centralized data network. The constraint in this case is that the amount of traffic travelling in any subtree within the network has to be below a certain level. This solution is then compared to a solution generated by a heuristic method that is commonly used. The two methods will be compared according to how good a solution they provide, how long it takes to produce the solution and how flexible the algorithm is. The more flexible an algorithm the easier it is to modify the algorithm to suite a particular type of network or design criteria. This thesis is meant to be the basis of further research to be undertaken in this department, therefore it is important that all aspects of the problem and the tools used to solve the problem are clearly explained.

The tools used in this thesis also have relevance in the design of a network interconnecting LANs. The LAN would be connected in a simple topology such as a ring or star, the different LAN segments would be connected together by either a bridge or a router thereby forming a tree topology. The problem statement mentioned in section 1.5 could be modified to suite this problem. More about this is mentioned in chapter 5.

When considering this problem it should be noted that even for a small number of nodes there are many thousands of ways of forming this tree network. It is almost impossible to perform an exhaustive search of all the possible topologies and then find the most economical one. This type of problem is termed NP-Complete. All problems falling into this class have no polynomial-time algorithms which can guarantee an exact solution.

Minoux ([1]) provides a general overview of network design principles. He specifies some of the applications, models and solution techniques that can be applied to network topology optimisation. There are also many papers ([2], [3], [4] ,[5]) and books ([6], [7]) that are either dedicated to network design or mention some aspect of the problem. Three papers ([8],

[9], [10]) have applied Evolutionary Algorithms to a similar network topology optimisation problem.

The following few sections in this chapter are provided so that the reader can clearly understand exactly what is being done in this thesis and what all the terms mentioned above and in the next few chapters mean.

1.1 Solution Methods

This section gives a brief overview of possible methods that have been used to solve network topology optimisation problems.

In the past, and even still today, designers have tackled this problem by the manual method. This is the combination of years of experience and rules of thumb in order to come up with a design. The major advantage of using this method is its flexibility. No data regarding traffic and cost information needs to be collected. The designer can change or modify the goals of the network without worrying about lost time because there is no set-up time. However, the disadvantages of using this method outweigh its advantages. It is rarely quantitative, this means that decisions are made subjectively without much basis in theory. It is therefore difficult to repeat any good design or learn from one's mistakes. It is also too time consuming to consider many alternatives. Therefore, the designs tend to follow the designer's intuition. This can result in some serious design errors being made.

It is important to devise a method that will generate a small number of solutions that are not necessarily optimal but are good solutions none the less. These methods need to retain the advantages of manual design and most importantly still keep the designer an integral part of the design process. Heuristic methods are often used in order to achieve this, they are relatively fast and can provide good, if not optimal, solutions. Heuristics are principles that are used in designs and are incorporated into algorithms, this makes the design process automated. Similar to rules of thumb they are good techniques that include design experience. The difference between a heuristic and a rule of thumb is that the heuristic can be laid out in mathematical form and is repeatable. Therefore many different heuristics can be tried on the same design problem and the best solution retained.

One of the most widely used heuristics is called the greedy algorithm. The greedy algorithm finds a solution by always taking the best option when faced with a choice, this however does not always lead to the best set of decisions being made.

Another method for solving this problem is the Genetic Algorithm (GA). The GA is a powerful optimisation technique. It was first introduced by Holland in his 1975 PhD thesis ([11]). Since then it has been the focus of widespread research. The research has focused on applications of the GA as well as on finding an all purpose powerful optimisation tool.

The GA is a useful tool for solving NP-Complete problems. This and the fact that it has a well known ability to avoid termination at local optima were the main motivations for considering the use of a GA to solve this particular problem. It should however be noted that due to the nature of its operation, a GA is not guaranteed to find the global optimum on every run.

The GA is based on Darwin's theory of natural selection. Hawking ([12]) gives a good overview of this theory :

> The idea is that in any population of self-reproducing organisms, there will be variations in the genetic material and upbringing that different individuals have. These differences mean that some individuals are better able than others to draw the right conclusions about the world around them and to act accordingly. These individuals will be more likely to survive and reproduce so their pattern of behaviour and thought will come to dominate.

In GA theory the population of organisms would be candidate solutions to the function that is being optimised. Each of these solutions contain schemata[1] (genetic material) which define how good a solution they are. In the theory of natural selection the individuals that are better able to adapt to their environment, or in GA theory the individuals that offer better solutions, would be more likely to survive and reproduce. The idea is that this reproduction may produce individuals that are even better solutions than their parents. In this way, after many generations, the global optimum can be found.

[1]Schema theory attempts to explain how and why a Genetic Algorithm works. This is explained in greater detail in Appendix A

The first example of a GA was called the Simple Genetic Algorithm (SGA). Palmer and Kershenbaum ([8]) and Berry et al. ([9]) have used the SGA to successfully solve a similar problem in network design. The SGA is a relatively old technique and since then other Evolutionary Algorithms, based on the GA have been developed. One such technique is called Population Based Incremental Learning (PBIL). PBIL outperforms the SGA on a wide range of problems. It is for this reason that it was the Evolutionary Algorithm chosen to optimise this problem.

1.2 Trees

This section introduces the concept of trees from two perspectives. Firstly it discusses how a tree topology is used in the context of a local network. Secondly it defines a tree in terms of graph theory, explaining certain terms that are necessary in understanding some aspects in the following chapters.

1.2.1 Tree Network Topology

The tree topology is a generalisation of the linear bus topology. A diagram of it is shown in figure 1-1. In this case the transmission medium is a branching cable. Packetized transmission is generally used. Therefore a message originating at a node will be broken up into smaller packets before being transmitted. A transmission from a node will propagate along the length of the cable and can be received by any other node. In order that a node knows which packet is intended for it, control information containing the address of the destination node is appended to each packet.

1.2.2 Graph Theory[2]

This section introduces terminology that can be used to describe networks, graphs and their properties. Graphs have many real-world applications. These applications range from Mathematics to Chemistry to Computers and to Communications. In communications graphs are

[2]Gibbons [27], Hartsfield and Ringel [28]

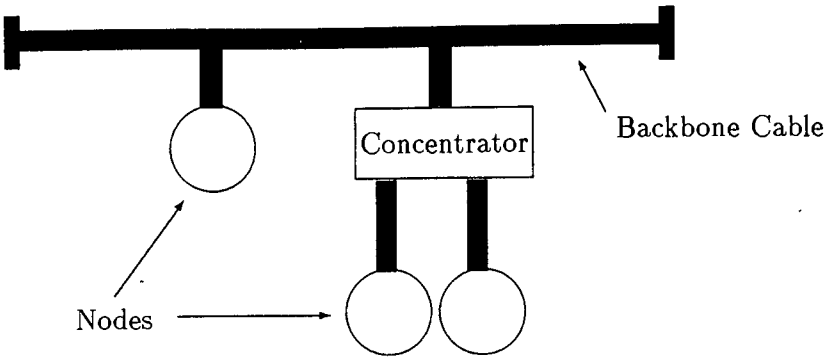


Figure 1-1: The Tree Topology

used to show the interconnection of nodes, or the topology, in a communication network.

A graph, G, is defined by two characteristics, the set of all vertices, V, and the set of all edges, E. In the application of graph theory to networking, the vertices are usually called nodes and represent communication devices such as a terminal or switch. The edges are usually referred to as links and these are transmission facilities between the nodes. This can be expressed in mathematical form as,

$$G = (V, E)$$

Where $V = \{v_1, v_2, \ldots, v_N\}$,with N the number of nodes in the graph, and $E = \{e_1, e_2, \ldots, e_M\}$, with M the number of links in the graph.

A link is in fact a connection between a pair of nodes, this is often represented as,

$$e_j = (i, k)$$

Where i and k correspond to v_i and v_k the $i-th$ and $k-th$ nodes respectively. In order to keep the notation used throughout this thesis consistent, a node is just referred to using its node number.

A link joining two nodes is said to be incident on each of the nodes. An adjacent pair of nodes, i and k, have a link (i, k) between them. These two nodes are referred to as neighbours. The

degree of a node is the number of links that are incident on it, or alternatively the number of neighbours it has.

A network is a graph where there are properties associated with its links and nodes. These properties could be length, capacity, cost etc. It is networks that are used in communications and the properties related to the links and nodes depend on the particular problem that is being dealt with.

A path in a network is a set of links from node S to node T. If node S and T are the same point then the path is called a cycle.

A graph is said to be connected if there is a path between any two nodes i and k. The set of nodes that have paths to one another is called a component. For instance in figure 1-2 there are two components, one containing the nodes 1, 2 and 3 and the other containing the nodes 4 and 5.

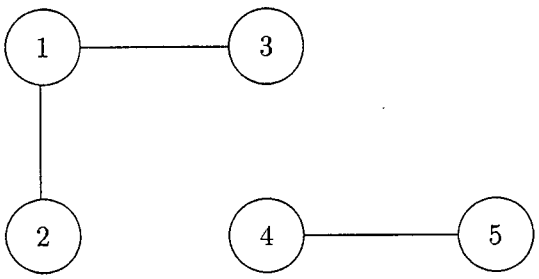


Figure 1-2: Components of a Graph

Given a graph, $G = (V, E)$, H is considered to be a subgraph of G if $H = (V', E')$ where V' and E' are subsets of V and E respectively.

A tree is a graph that does not contain any cycles. A spanning tree is a connected graph, a graph with paths between each of its nodes, that has no cycles. A graph that is not connected is called a forest. For example the component in figure 1-3 is called a forest.

A subtree is a term used in this thesis to distinguish between different sections of a network. These sections can be rooted or unrooted. In figure 1-2 there are three subtrees. The first two are rooted. Subtree one contains the link $(1, 2)$, subtree two contains the link $(1, 3)$ and subtree three contains the link $(4, 5)$.

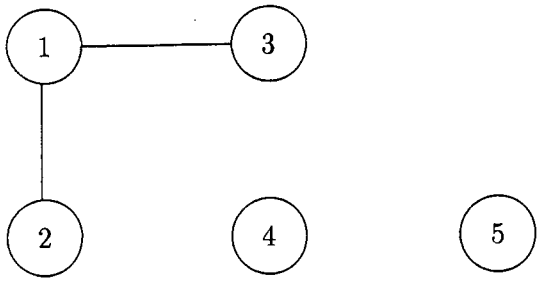


Figure 1-3: A Forest in a Graph

This terminology will come in useful when Kruskal's and Prim's algorithms are explained. These are both minimum spanning tree algorithms. Kruskal's algorithm is used in the development of the heuristic and Prim's algorithm is used in the coding scheme related to the Evolutionary Algorithm.

1.3 Centralized Data Network[3]

This section introduces the centralized data network and where it is in use today.

Centralized data networks are also known as client-server networks. These are the simplest forms of data communication networks in which a terminal (client) accesses a central data site (server). A diagram of this process is shown in figure 1-4.

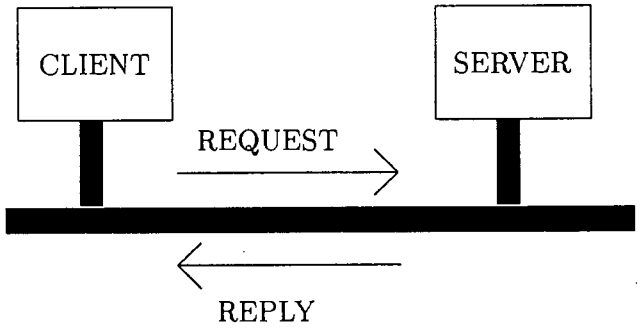


Figure 1-4: The Client-Server Process

[3]Kershenbaum [6]

The client would generally send a request to the server for work to be done. The server would then process the work and send it back to the client as a reply. One of the goals of this type of network is usually to save money. The price/performance ratio of a small personal computer is much higher than that of a mainframe. Therefore there are generally many clients connected to a few servers.

There are many applications where this type of network is currently in use today. An insurance company might use a client-server network. The server would contain a centralized database of all information relating to its clients, such as the type of policies they have and what claims have been made against those policies. A terminal connected to this centralized database would be able to access and update this information from any office throughout the country.

Another example of a client-server network would be a bank's automatic teller machine (ATM) network. In this instance the clients would be the ATM machines and the server would maintain a database of all the necessary information related to the bank's customers. This type of network may also form the local access section of a more complex network. For instance, the client-server networks from different banks may be linked together so that people can access their banking information or money from any ATM. The Saswitch network is a network in this country that performs this task.

It is important at this point to note the difference between remote access and the client-server network architecture. In a client-server network each node is permanently connected to the network. A node connected via remote access would be connected to a network via a dial up connection. This would be more useful when a node only needs to transmit a small amount of information and when the transmissions are infrequent.

Figure 1-5 is an example of a centralized data network.

As can be seen the main components of the network are terminals, multiplexors and concentrators. These are grouped under the term nodes. Throughout this thesis the term node is often used and these are the devices it will represent. A brief discussion of what they are appears below.

Terminal A terminal is a simple device that serves a single user. For this reason they are

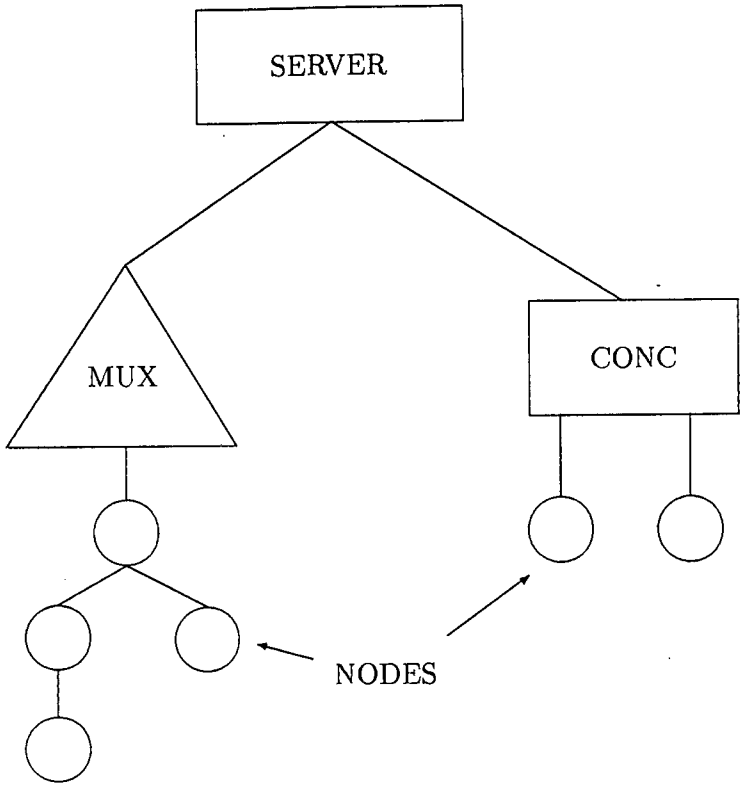


Figure 1-5: A Centralized Data Network

only the source of a small amount of traffic. A terminal would usually consist of a keyboard and monitor but can also include peripherals such as disc drives and printers. Some examples of a terminal would be a PC, workstation or even a telephone.

Multiplexor The function of a multiplexor is to join many low speed lines onto one high speed output line. The speed of the output line could be greater than or equal to the sum of the speeds of the input lines. There are also more sophisticated multiplexors known as statistical multiplexors. These multiplexors compress the output data stream so that it uses less capacity than the sum of the input channels.

Concentrator A concentrator is similar to a statistical multiplexor but it buffers the input lines thereby enabling the output channel to have a considerably lower speed than the sum of the input channels.

The topology of a centralized data network, as shown in figure 1-5 is often restricted to a

tree. This is because the terminals cannot perform any routing. The functionality of these terminals is often kept to a minimum. Therefore most of the network management would be performed by the server. The advantage of this technique is that a lot of cost and complexity within the network is saved. The disadvantage is that a lot of extra data would be transmitted throughout the network. This is because any data inputted at a client is transferred to the server to be stored and at some later stage is sent back to the original client because this is where the information is generally needed. In some applications this would not be a major disadvantage, especially in applications where a centralized copy of the data is required.

There are two ways a terminal can be connected to the server. Either through a point-to-point line or a multi-point line. Multi-point lines are trees. In a multi-point line all the terminals are connected to the same transmission medium. Therefore any transmission from the server to a terminal can be heard by all the terminals connected to that line. In order that only the destination terminal accepts the message, the address of the destination terminal is attached to any transmission from the server. In order to avoid collisions caused by more than one terminal on a multi-point line transmitting messages simultaneously, the server controls who sends a message by polling and selecting each terminal in turn.

There may be many levels within such a network. A terminal can be connected directly to the server by a point-to-point line. Alternatively, many terminals can be connected through a concentrator or multiplexor. The multiplexor may in turn be connected directly to the server or go through another multiplexor or concentrator. Concentrators are usually connected directly to the central site but they too can be cascaded. Despite all these possibilities the network is usually kept as simple as possible so that it is easy to manage. Therefore while there are many possibilities mentioned in the above discussion, usually only a few are used in any given network.

A client-server network is in fact a virtual network. In this instance the functionality of the virtual network matches the operation of the physical network. However this need not always be the case, a client-server network could also run on a meshed type physical network.

1.4 Input Data for Network Design Problem[4]

The most time consuming part of the design process is the collection of the input data. This input data takes the form of traffic and link cost information for all of the links between the nodes. In order for the input data to be useful it is important that this information is current. Designing a network based on traffic flows that are not up to date will not satisfy its design requirements. The reason being that when a network is modified it is usually because the current network cannot handle the load. Long term approximations are also generally inaccurate. It is therefore better to work with current data even if it is incomplete.

One of the main reasons an organisation will build and manage its own network, rather than relying on a service provider's facilities, is the ability to quickly change the network to suite its requirements. A lengthy procedure to collect the input data for the network design problem will negate this advantage. Therefore, it is important from an efficiency point of view that the design process is able to be completed with incomplete or inaccurate data. A distinction is drawn here between incomplete information and "wild guesses". Therefore, it is necessary to find as much accurate data as possible within a given amount of time and effort exerted.

Below is a description of the traffic and cost data needed as inputs to the network topology optimisation problem. There are many models that will enable a designer to come up with meaningful input matrices. Each of these models is applicable in certain situations and it is important that a designer clearly understands the current circumstances in order to choose an appropriate model.

1.4.1 The Traffic Matrix

This is usually the largest part of the data collected. There are a large number of dimensions to this data. These could include the date, time of day, the amount of information sent and the application used to send this information. It is important to process this data so that it is in a more manageable form and can effectively be used in the design procedure.

The traffic matrix can have many dimensions. This is an inefficient way of storing and

[4]Kershenbaum [6], Tanenbaum [7]

processing this information so a compression of the matrix along several of the axes is often performed. A two dimensional matrix is often desirable with the elements of the matrix representing values from the various models that can be used. This means that each element in the matrix could represent the total number of messages sent between an origin and destination pair multiplied by the average message length. Alternatively it could represent the total number of characters sent between the origin and its destination. It is also possible, from the data accumulated, to develop a distribution of the traffic by the hour for each day. The network could then be designed to handle the traffic of the busiest hour.

The capacity of a channel is the amount of traffic the channel can carry. The elements of the traffic matrix could be this capacity. In most systems, overhead uses a small amount of this capacity. The capacity left over to be used by the applications running on the network is called the usable capacity. The usable capacity depends on the technology of the network. It is therefore important to understand how this technology works in order to make a good model of the capacity.

Communication requirements can be either directed or undirected. Most data requirements would be directed. An example of this would be a computer network. In this instance there is a relationship between the data sent in each direction along a channel but the amount of information is not the same. An example of an undirected communication requirement would be analogue voice which ties up the channel in both directions. This would make the traffic matrix symmetrical. Clearly in both of these instances the diagonal elements of the traffic matrix would be meaningless since we are considering each node, whatever it may be, as an entity and therefore intranode traffic is irrelevant.

All of the above approaches are useful depending on the situation a designer would find himself in. It is important to note that a traffic matrix built up from raw data using one of the above mentioned models is just an estimate. Therefore there may be several iterations between the development of the traffic matrix and the network design, each time refining the model used.

1.4.2 The Cost Matrix

The elements of the cost matrix are the costs of a link between any two nodes in the network. Costs may be either fixed or variable.

Fixed costs consist of monthly charges and one off fees such as installation costs. This installation cost can be converted to a monthly cost by amortising it over a fixed period of time. An example of this would be a microwave link that has been set up as part of a private network. The major costs involved here are the purchase and installation fees of the equipment. This value can be converted to a monthly cost by amortising it over the expected lifetime of the equipment, usually a few years. Communications costs can also be usage sensitive. In this case the elements of the matrix represent a cost per minute or cost per bit value.

There are many different tariff structures offered by the common carrier. This would depend on distance, the type of technology used and the bit rate required. Due to the variability in the costs an approximate value of the link costs is often used. Therefore a compromise is made between accuracy and the ability to quickly gather information. This will ensure that the network designer is able to respond to the design requirements in a reasonable amount of time.

1.5 The Problem Statement[5]

This section describes the problem in greater detail. It mentions what assumptions are made and how the input data is generated.

Section 1.3 discussed the different communication devices a client-server or centralized data network can have. It also explained the different ways all the devices can be connected together in order to achieve the required topology.

In the optimisation of the topology of this type of network there are three separate problems that need to be considered.

[5]Kershenbaum [6]

1. The concentrator location problem - where to place the concentrators and whether to use them at all

2. The terminal assignment problem - which terminals need to contain concentration capabilities

3. The topology selection problem - whether to connect a terminal to the server using point-to-point or multi-point lines and how to layout the topology of the multi-point lines

This thesis focuses on the topology selection problem, specifically how to layout the topology of the multi-point lines. There are certain assumptions that need to be made when approaching this problem.

There are N nodes in the network. These nodes are numbered from 1 to N with node 1 being the root node or server. The traffic from the root node to node j is denoted t_{1j} and the traffic in the other direction is denoted t_{j1}. Each node can represent a single terminal or a group of terminals. The values for t_{1j} and t_{j1} can be arrived at by monitoring, or estimating if the network is currently not in operation, the traffic between the nodes and the server over a period of time. This could be a day or a week. t_{1j} and t_{j1} are then assigned the maximum of these recorded values.

In this instance the assumption is made that there is only one number used to model the traffic flow in each direction. However, depending on the situation, the traffic matrix may contain more than two dimensions. The other dimensions may contain traffic for each application running on the network or traffic at certain times during the day. This data can then be used to design a network to meet precise performance requirements. In this problem, however, the network model that is being used is a fairly simple one. In fact, the traffic data is simplified even further. Since the traffic from a terminal to a server generally only takes the form of requests, the traffic in the other direction, the replies from the server, will dominate. Therefore the only traffic value that needs to be used is t_{1j} the traffic from the root node to node j. This value is called the weight of the node. It would be fairly easy to modify the problem to handle the cases where these simplifying assumptions cannot be made.

The cost matrix is also an input to the problem. This matrix is symmetric, implying that the

cost of a link from node i to node j is the same as the cost of the link in the other direction. In fact, this is generally the same link. The design of this client-server network is done using single speed lines. This is done because a tree , or a multi-point line, has terminals which are all connected to the same transmission medium and it is not possible for different segments of the medium to operate at different speeds. The elements of the cost matrix are the costs of the links between all of the nodes. Since constant speed lines are used in the design of this network, the line rental will be a constant amount per month. It will not depend on the amount of data that is transmitted over that line but on the distance over which is was transmitted. If it is not possible for a link to be constructed between any of the nodes, this position in the matrix is given the value infinity. The overall cost of the network is just the cost of all the links in the tree. This is a fairly simple cost model but other costs such as installation costs and equipment costs can be included if required.

The problem is then to find the optimum topology of all the terminals connected to the server using multi-point lines. The terminals with higher weights are connected directly to the server using point-to-point lines. Therefore, a formal definition of the problem is to determine the tree topology which has a minimum total cost subject to the constraint that each subtree in the network has a maximum amount of traffic flowing through it that does not exceed the threshold, W_{max}. This can be stated mathematically as,

$$\text{Find :} \qquad z = min \sum_{i,j} c_{ij} x_{ij}$$

Subject to :

$$\sum_{i,j \epsilon T_k} t_{1i} x_{ij} \quad \leq \quad W_{max}$$

$$\sum_{i,j} x_{ij} \quad = \quad N - 1$$

$$x_{ij} \ \epsilon \ \{0,1\} \qquad 0 \leq i < j \leq N$$

cycles are avoided

Note :

- t_{1i} is the amount of traffic flowing from the root node to node i.
- $x_{ij} = 1$ if link (i, j) is in the network
- z is the cost of all the links in the network
- k can take on values from 1 to N
- T_k is the $k - th$ multi-point line or subtree

There are four constraints in the above problem statement. The first ensures that the traffic in any subtree does not exceed the threshold, W_{max}. The second ensures that the tree is connected and the next two are self explanatory.

Chapter 2

The Heuristic

This section describes the heuristic used to solve the problem of network topology optimisation. It is based on a minimum spanning tree (MST) algorithm. An MST is a connected tree of minimum total length. In this case the length of a link refers to its cost.

The name of the heuristic is the Constrained Minimum Spanning Tree (CMST). It is based on Kruskal's MST algorithm. Kruskal's algorithm is a greedy algorithm. This means that at each stage in the algorithm, when a link needs to be added, it takes the shortest link. This set of decisions does not always lead to the best solution.

The constraint in this MST is that the traffic flowing through all the subtrees rooted at the centre must be below a certain threshold. A CMST algorithm is used to solve this problem because MST algorithms are simple and provide effective solutions in a relatively short amount of time.

This chapter begins by describing Kruskal's algorithm, providing an example for clarity. It then describes the CMST algorithm, showing how it differs from the MST algorithm by using the same example as above.

2.1 Kruskal's MST Algorithm

As mentioned previously the goal of an MST algorithm is to find a spanning tree that has the smallest possible sum of the weights of its edges. An informal description of the algorithm appears below.

The first step is to list the edges in ascending order of length. Following that there are a number of steps that need to be completed. These are shown below.

1. Select the edge that has the smallest weight in the graph. If there is more than one edge with this weight, the decision is arbitrary. This edge is the first link in the tree.
2. Again select an edge with the smallest weight that is not yet in the existing forest and add it to the tree provided it does not make a cycle.
3. Check to see if you have a spanning tree, if so stop, if not proceed to step 2.

A proof that this algorithm produces a minimum spanning tree is beyond the scope of this thesis. It can however be found in most books on operations research or discrete mathematics.

The efficiency of the algorithm depends on the procedure used to check if adding an edge would produce a cycle. There are many methods that can be employed to achieve this. The method used in this thesis is described below.

If the endpoints of the edge can both be found in the same subtree then adding this edge will create a cycle. If the endpoints belong to different subtrees and both these subtrees are rooted then again, adding this edge will create a cycle, however if only one of these subtrees are rooted then the edge can be safely added to the tree. The edge can also be added to the tree if only one of its endpoints can be found in the tree. If neither of the endpoints can be found in the tree then this link will form a new subtree.

Below is an example of how the algorithm works. The problem is to find the MST of the graph in figure 2-1.

Sorting these components in order will lead to table 2.1 being generated.

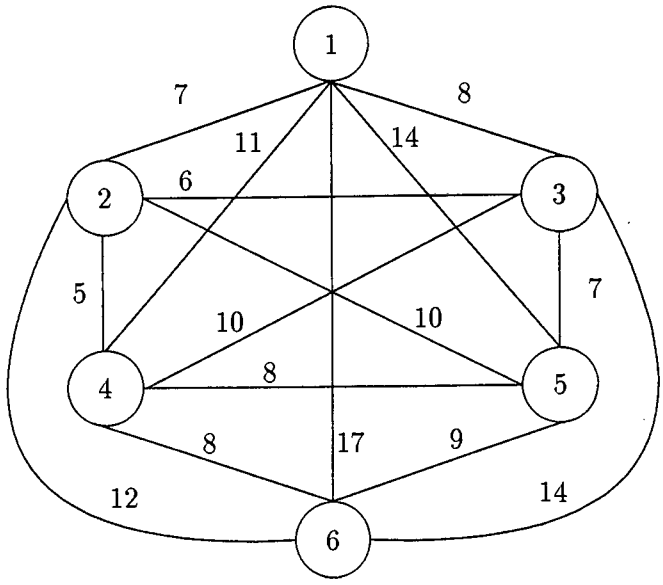


Figure 2-1: Find an MST of this Graph

The first link selected by Kruskal's algorithm would be $(2,4)$. Adding the next three links $(2,3)$, $(1,2)$ and $(3,5)$ does not create any cycles. The forest generated thus far is shown in figure 2-2.

As can be seen in figure 2-3 adding the next link $(1,3)$ does create a cycle. The link creating the cycle is shown marked with a '*'. The reason a cycle is created here is because both the subtree and the link are rooted and the other endpoint of the link can be found in the subtree.

Proceeding through the table in this way will finally generate the MST shown in figure 2-4.

2.2 The Constrained Minimum Spanning Tree Problem

This is the algorithm that is going to be compared with the Evolutionary Algorithm. The problem is to find an MST of the cost matrix subject to the constraint that the maximum amount of traffic flowing in any rooted subtree does not exceed a certain threshold, W_{max}. As mentioned earlier the CMST is based on Kruskal's algorithm which was explained in the previous section. A more detailed explanation of how the CMST algorithm was implemented can be found in Appendix F.

Value	Node 1	Node 2
5	2	4
6	2	3
7	1	2
7	3	5
8	1	3
8	4	5
8	4	6
9	5	6
10	2	5
10	3	4
11	1	4
12	2	6
14	1	5
14	3	6
17	1	6

Table 2.1: Ordered Link Weights

The CMST algorithm begins the same way as does Kruskal's algorithm. It sets up a list of all the link weights in ascending order. Included in this list are the nodes that each link is incident on. The smallest link in this list is the first link of the tree. The procedure to generate the CMST is as follows. The next smallest link from the list is extracted. If adding this link does not violate any constraints, i.e. it does not create a cycle and it does not cause the traffic constraint to be violated, then the link is added to the tree. If any constraints are violated the link is discarded. The above procedure is repeated until the tree is connected.

The above algorithm produces a constrained minimum spanning tree with subtrees meeting the capacity constraint. If the capacity is large enough then the tree that is generated is just a minimum spanning tree, i.e. the algorithm reverts back to Kruskal's algorithm.

This algorithm is tested on the same problem as Kruskal's algorithm. There are however additional inputs. These are the traffic threshold, W_{max}, which is set equal to 3 and the traffic matrix which appears below.

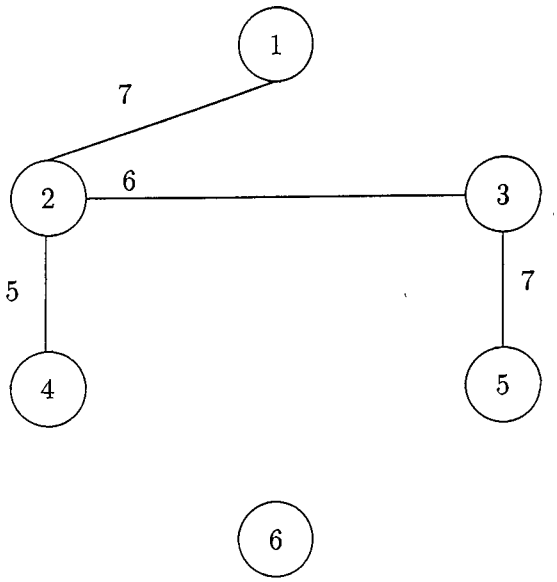


Figure 2-2: The First Four Links of the MST

$$traffic = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

What this matrix means is that the only traffic being taken into account in this problem is the traffic from the root node to each of the terminals, t_{1j}. This traffic value is arbitrarily set equal to 1.

The first few links added to the tree would follow the same procedure as Kruskal's algorithm. Therefore after adding the first three links the tree would be as in figure 2-5.

In Kruskal's algorithm the next link to be added would be $(3, 5)$ as seen in figure 2-2. However, adding this link would violate the traffic constraint as a maximum traffic flow of four units would flow in this subtree.

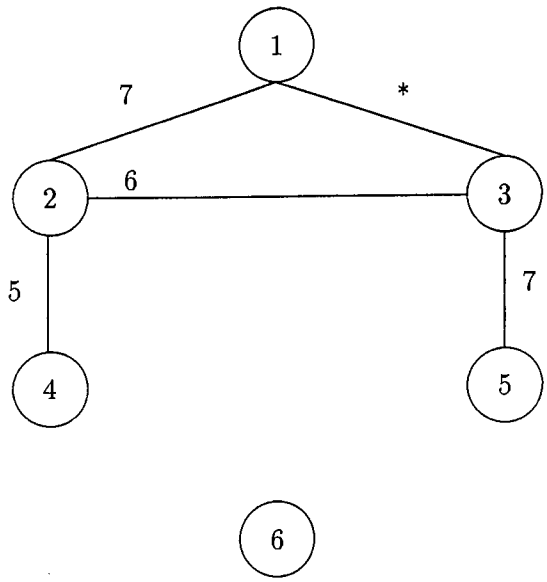


Figure 2-3: A Cycle is Generated

The solution is completed by first adding link $(5, 6)$, this is the next link in the list that does not violate any constraints. Adding this link divides the tree into two subtrees. Adding the next suitable link, $(1, 5)$, completes the tree. The constrained minimum spanning tree appears in figure 2-6.

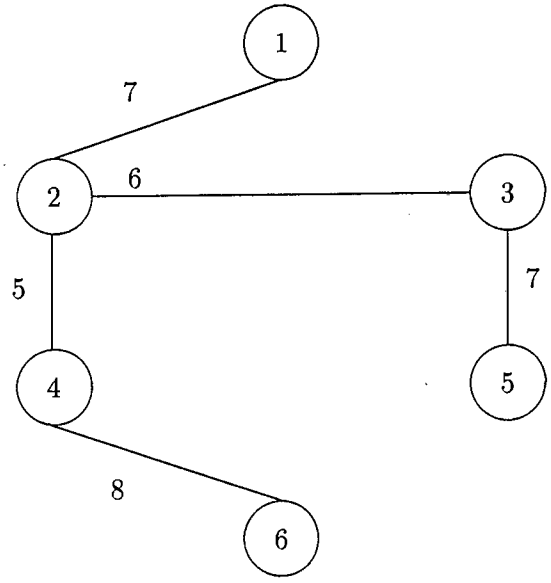


Figure 2-4: The MST

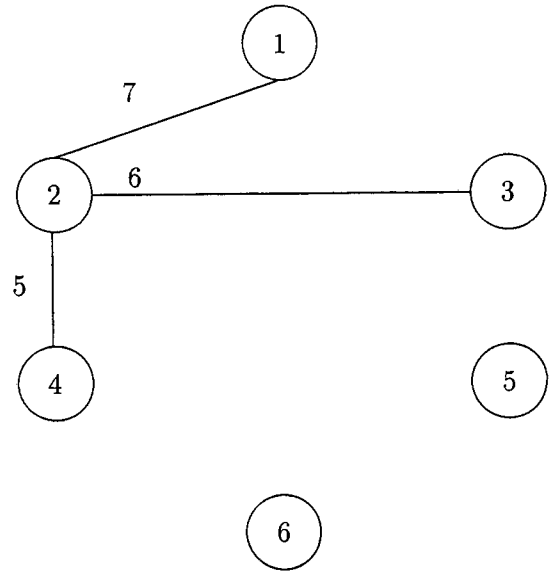


Figure 2-5: The First Three Links in the Tree

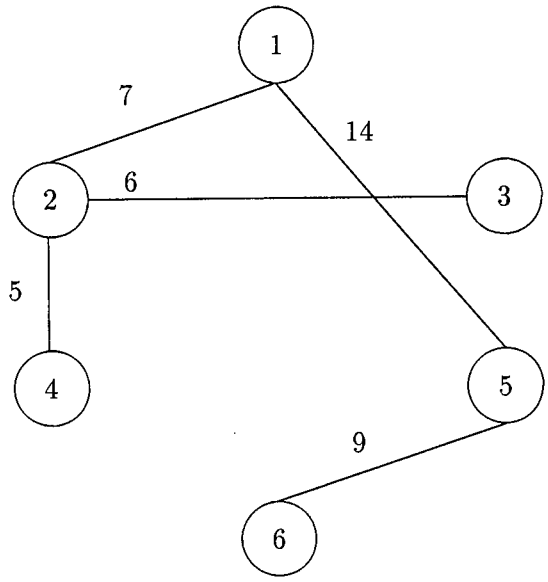


Figure 2-6: The Constrained Minimum Spanning Tree

Chapter 3

Implementing Networks as Trees

This chapter deals with the mapping of the problem into a domain that PBIL is able to understand and evaluate (For more details on PBIL and the Genetic Algorithm see Appendices A and B). In order to gain a good understanding of the encodings it is important to first explain how the fitness function and coding scheme are used in Evolutionary Algorithms.

The first section of this chapter discusses how the coding scheme is used in Evolutionary Algorithms. The second section discusses the requirements for an effective method of encoding the problem. The next section introduces a few alternatives and then proposes a method that meets all the requirements (A more detailed discussion on how this is implemented in Matlab can be found in Appendix F). The fourth section introduces the fitness function and how it is used in Evolutionary Algorithms. The final section discusses how the fitness function is calculated, in other words, how the performance of each candidate solution to the problem is evaluated.

3.1 The Coding Scheme and the Evolutionary Algorithm

The coding used in a problem maps the problem into a domain that the evolutionary algorithm can understand and evaluate. Each candidate solution is represented as a set of parameters, this is often referred to as a chromosome. If the problem is to optimise a computer network topology then the parameters would in some way represent that topology (see section 3.3).

Initially all research was directed at using binary values in these chromosomes and all the theory explaining why GAs work was developed with binary values in mind. Recently however, a lot of research has been performed using high-cardinality alphabets. Each symbol in these alphabets could, for example, represent a real number or an integer. Some papers have made a direct comparison between a real valued GA and its binary valued counterpart and found the results for the real case to be better ([21], [22], [23], [25]). All discussions explaining how and why the GA and PBIL work will be related to the binary valued chromosome as this is how the traditional GA operates (This explanation can be found in Appendix A).

Consider a three dimensional function that needs to be optimised called $F(x, y, z)$. Each of these real valued variables can be represented by 10 bits[1]. This would mean that each chromosome would be thirty bits long.

The terms used to define various elements within the GA are in keeping with its analogy to genetics, all taken from this area. The parameters represented by a chromosome are referred to as a genotype. In the above example the variables x, y and z would be the genotype. A phenotype is a particular solution. So, any candidate solution with values for x, y and z would be called a phenotype. Each bit within the phenotype is called a gene. The fitness of an individual depends on this phenotype.

3.2 Criteria for Coding Scheme[2]

In order for an Evolutionary Algorithm to operate effectively and efficiently in solving this problem it is important that the coding scheme used has certain properties. These are listed below.

1. The coding scheme should be capable of representing only trees. If this is not the case then every time a new population is created, which would be at the beginning of each new generation, there is a possibility that many of the candidate solutions may not be trees. This would mean that these individuals would need to be regenerated or repaired

[1]The number of bits used will depend on the range of the variables and their required accuracy. See Appendix C for a more detailed explanation.

[2]Palmer and Kershenbaum [8]

and would increase the complexity of the algorithm as well as the time the algorithm needs to reach a solution.

2. It should be easy to move between the encoded version of the tree and the representation of the tree that would be used to calculate its fitness.

3. Small changes to the encoded version of the tree should result in small changes to the tree itself. This property is called locality. In practice this would mean that searching in the area of the search space surrounding a high quality solution would produce similar solutions that may have a higher fitness. If this were not the case searching in the area surrounding an individual may produce a completely different solution. In this case an Evolutionary Algorithm would not perform well and the problem may as well be attempted by a purely randomised search algorithm.

4. All trees should be able to be represented by the same number of parameters. This would give the Evolutionary Algorithm a fair chance of reaching all areas of the search space.

5. The coding scheme should be capable of representing all trees. Clearly this is an important quality for any coding scheme to possess. It ensures that the algorithm is capable of searching in all areas of the search space.

3.3 Different Types of Coding Schemes[3]

As mentioned section 1.4.1 communication networks can be either directed or undirected. Throughout this thesis all discussions and results will involve the undirected case since this is the easiest and the quickest to evaluate.

Before beginning to discuss the various coding schemes there are certain parameters that need to be defined. These are the inputs to the network topology optimisation problem as well as certain notations which will be used throughout this discussion. We are given N nodes labelled 1, 2, ..., N. The cost and traffic matrices are also two of the input variables however since they relate to the edges between the nodes and not the nodes themselves they will not

[3]Palmer and Kershenbaum [8]

be needed for this discussion. The edge between two nodes i and j will be represented as (i, j).

It was shown by Gibbons ([27]) that the total number of trees that can be represented by a complete graph of N nodes is N^{N-2}. Each of these N nodes can be designated a root node therefore there are N^{N-1} possible trees that can be represented. This information can be used to check the efficiency of the coding scheme. This will be done by comparing the total number of trees that can be represented using the coding scheme to the total number of trees that can possibly be generated.

3.3.1 Characteristic Vector Representation

In this coding scheme an index p is associated with each link (i, j) in the tree. A tree, T, would be represented by a vector, $E = [e_0, e_1, ..., e_K]$. In this vector K is the total number of edges in the tree and e_p is one if an edge is included in the tree and is zero if the edge is excluded.

The total number of edges in any tree that has N nodes is $K = N(N-1)/2$. Therefore every vector E can have $2^{N(N-1)/2}$ different values and as will be shown, most of these are not trees. Any tree that has N nodes will have exactly $N-1$ edges. Therefore if there are more than $N-1$ ones in the vector E, that vector will not represent a tree. It can also be shown that as N increases the probability of E representing a tree decreases. This means that in PBIL every time a new generation of individuals is created from the probability vector most of these individuals will not be trees.

It can be shown that it will take $O(N^2)$ effort to go back and forth between this representation of a tree and the representation used to calculate its fitness. This is not very good because as will be seen, in other coding schemes only $O(N)$ effort is required.

This coding scheme also has some good points. All trees can be represented by these vectors and all will be represented equally. The representation possesses locality. This means that changing a bit in the vector would add or delete a link on the tree.

All in all this coding scheme is not a good one to use since a lot of the vectors will not

represent trees and this would present PBIL with a lot of problems.

3.3.2 Predecessor Vector Representation

In this representation each tree can be represented as a vector of length N. The root node is designated to be node 1. The vector is generated by listing in position i the predecessor node to node i relative to the root. With 0 being placed in the position of the root node. For example, the vector $[0, \mathbf{1}, 4, \mathbf{2}, \mathbf{2}]$ represents a tree with links $(\mathbf{1}, 2)$, $(\mathbf{2}, 5)$, $(\mathbf{2}, 4)$ and $(4, 3)$. The link $(\mathbf{1}, 2)$ is part of the tree because **1** is in the second column of the vector. Since 0 is in the first column, node 1 is the root node.

As can be seen each rooted tree is represented by a unique N digit number. For each tree there are N^{N-1} possible combinations of those numbers. As mentioned previously for every N nodes there are N^{N-2} trees that can be generated, keeping the same node to be the root each time. Therefore a vector in this representation will be a tree with probability $\frac{1}{N}$. This is better than the characteristic vector representation for two reasons. The probability is lower and will decrease, not increase, as N increases. It does however allow for many non-trees to be generated.

It can be shown that it will take $O(N)$ effort to convert between this representation and a more conventional representation that would be used to calculate its fitness. This is an improvement over the previous coding scheme.

In summary, this coding scheme is better than the characteristic vector representation. There are however still many vectors that can be generated that do not represent trees.

3.3.3 Prüfer Number Representation

The Prüfer number of a tree with N nodes is a vector of length $N - 2$. The elements of the vector are numbers between 1 and N and the algorithm used to generate this vector is defined below.

1. Let i be the lowest numbered leaf[2] node. Let j be the predecessor node to i. Then j becomes the first element of the Prüfer number, $P(T)$. The vector $P(T)$ is built up by adding digits to the right.

2. Remove node i and the edge (i, j) from the tree. It is therefore possible that if i was the only successor to j that j now becomes a leaf node.

3. When only two nodes are left $P(T)$ has been formed. If there are more than two nodes left proceed to step 1.

Figure 3-1 is a diagram of a tree. This tree will be used to show how $P(T)$ is generated.

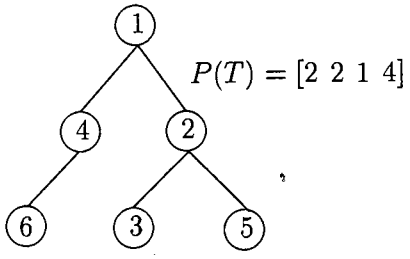


Figure 3-1: A Tree and its Prüfer Number Representation

As can be seen from this figure the lowest numbered leaf node is 3. The predecessor to this node is node 2. Therefore $P(T) = [2]$ and node 3 and link $(2, 3)$ are removed from the tree. Since there are more than two nodes left in the tree the process is repeated again. This time the lowest numbered leaf node is node 5, its predecessor is also node 2 so $P(T) = [2\ 2]$. Node 5 and link $(2, 5)$ are removed from the tree therefore making node 2 the lowest numbered leaf node. Its predecessor is the root node, node 1, and therefore $P(T) = [2\ 2\ 1]$. This time node 2 and link $(1, 2)$ are removed from the tree. The lowest numbered leaf node is now node 6, its predecessor is node 4 making $P(T) = [2\ 2\ 1\ 4]$. Removing node 6 and link $(4, 6)$ from the tree results in only two nodes being left. Therefore the Prüfer number representation of the tree is $P(T) = [2\ 2\ 1\ 4]$.

It is not immediately obvious from $P(T)$ what the tree it represents will look like. Therefore there is another algorithm to convert from the Prüfer number back to a tree. This algorithm

[2] A leaf node is a node that is at the end of the tree, i.e. it only has one connection to another node

is shown below.

1. Let $P(T)$ be the Prüfer number and let all nodes that don't appear in $P(T)$ form the vector Y.
2. If $P(T)$ is empty there should be exactly two nodes, i and j, in Y. Add the link (i,j) to the tree. The tree has now been completely generated. If $P(T)$ is not empty proceed to step 3.
3. Let i be the lowest element in the vector Y. Let j be the leftmost element of $P(T)$. Add the link (i,j) to the tree. Remove i from vector Y and j from $P(T)$. If j does not occur anywhere else in $P(T)$ insert it into Y.
4. Goto step 2

In order to explain this algorithm the same example will be used as was used previously. This would make $P(T) = [2\ 2\ 1\ 4]$ and $Y = [3\ 5\ 6]$. The lowest number in Y is 3 and the leftmost number of $P(T)$ is 2. Therefore the first link in the tree is $(2,3)$. Removing 2 from $P(T)$ and 3 from Y gives $P(T) = [2\ 1\ 4]$ and $Y = [5\ 6]$. The next link to be added to the tree is $(2,5)$. This would result in $P(T) = [1\ 4]$ and $Y = [2\ 6]$. Node 2 is added to vector Y because it is no longer part of $P(T)$. The lowest number in Y is 2 and the leftmost element of $P(T)$ is 1 therefore the next link to be added to the tree is $(1,2)$. Removing 1 from $P(T)$ and 2 from Y would give $P(T) = [4]$ and $Y = [1\ 6]$. The next link to be added to the tree is $(1,4)$ and removing the relevant nodes would leave $P(T)$ empty and $Y = [4\ 6]$. Therefore the last link to be added to the tree is $(4,6)$. This would mean that the links in the tree are $(2,3)$, $(2,5)$, $(1,2)$, $(1,4)$ and $(4,6)$. This is the same tree as shown is figure 3-1.

A network with N nodes would be able to generate N^{N-2} Prüfer numbers. This is the same as the number of possible trees that can be generated from N nodes. Since there is a one to one relationship between a tree and its Prüfer number, each Prüfer number representing a different tree, this representation can only represent trees.

It can be shown that it would take $O(NlogN)$ effort to convert between a Prüfer number and its tree representation. Although both of the algorithms used to convert a tree between

the two representations are more complicated than in the other representations this is not considered to be a major disadvantage.

The major problem with this representation is that it possesses little locality, changing one number in $P(T)$ will change the tree dramatically.

3.3.4 LNB Representation

The previous three coding schemes that were discussed met most of the criteria mentioned in section 3.2 but all of them fell short of being ideal. This section introduces a coding scheme that does meet all the criteria and is therefore used in this thesis.

In this coding scheme the chromosome consists of a bias value for each node in the network. This bias value represents the tendency of a particular node either to be an interior node or to be a leaf node. The higher the bias value for a particular node the more likely that node is to be a leaf node. It is for this reason that the coding scheme is called the leaf node bias (LNB) representation.

In a four node network the chromosome would be $[b_1\ b_2\ b_3\ b_4]$. In this thesis biases are limited to the range $[0, 255]$. Therefore each bias can be represented by an eight bit binary number and the total length of any chromosome is the number of nodes in the network multiplied by eight.

This set of biases together with a multiplication factor p and the largest element in the cost matrix, cst_{max}, bias the cost matrix in the following way.

$$newcst_{ij} = cst_{ij} + p * cst_{max} * (b_i + b_j)$$

Where $newcst_{ij}$ are the elements of the modified cost matrix, cst_{ij} are the elements of the original cost matrix and p is multiplication factor which is set equal to 1.

Prim's algorithm is then applied to this modified cost matrix. Prim's algorithm finds the minimum spanning tree of a graph (see Appendix D for more details). The output generated by Prim's algorithm would put the tree in a form that can be easily used to calculate the

cost of the network.

Section 3.2 mentions certain criteria that a coding scheme needs to meet in order to be effectively and efficiently used in this problem. The discussion below shows that this coding scheme does in fact meet all of these criteria.

The first criteria was that the coding scheme should only be capable of representing trees. It should be easy to see that this is true. This is because the candidate solution is converted from its bias representation to a representation that can be used to calculate the cost of the network by a minimum spanning tree algorithm which, by definition will always generate a tree.

The second criteria specified that it should be easy to move between the encoded version of the tree and the version that would be used to calculate its fitness. It is fairly easy to move between these two representations. There are just two steps, the modification of the cost matrix and the generation of the MST from this matrix. The speed of this conversion depends on how efficient the implementation of Prim's algorithm is.

The third criteria stated that the tree should possess locality. This coding scheme does possess locality. Changing a bias by a small amount will have a minor affect on the MST being produced.

The fourth criteria was that each tree should be represented by the same number of parameters. Since each node in the network is represented by one bias value, all trees for a particular network will be generated by chromosomes of the same length.

The final criteria stated that the coding scheme should be able to represent all trees in the solution space. This is where the coding scheme possibly falls short. Palmer and Kershenbaum ([8]) show an example where this occurs. Consider the network shown in figure 3-2. The weights associated with each of these rows are the elements of the cost matrix.

Now consider that the tree to be encoded contains the links $(1,2)$, $(2,3)$ and $(3,4)$. Due to the nature of the MST algorithm it is impossible for an MST to contain the largest edge in any cycle. The three links $(1,2)$, $(2,3)$ and $(1,3)$ form a cycle. Therefore in order for the link $(1,2)$ to be selected as part of the MST the following inequalities need to hold true.

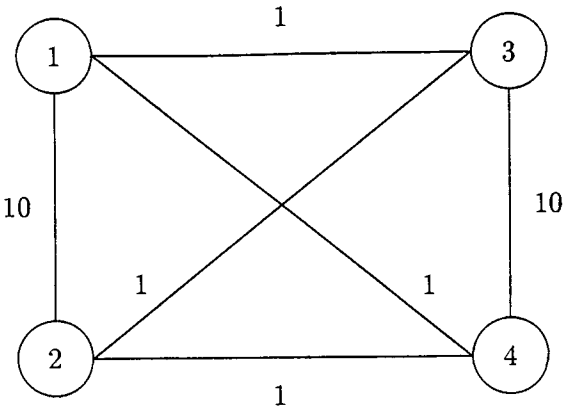


Figure 3-2: Example of a Tree that cannot be Represented using Node Biases

$$newcst_{12} \leq newcst_{13}$$

$$newcst_{12} \leq newcst_{23}$$

Now consider the cycle $(3,4)$, $(2,4)$ and $(2,3)$. If link $(3,4)$ is to be in the MST the following inequalities must hold true.

$$newcst_{34} \leq newcst_{23}$$

$$newcst_{34} \leq newcst_{24}$$

If the inequality $newcst_{12} \leq newcst_{13}$ is expanded using the equation to modify the cost matrix described above, the inequality simplifies to $9 + 10b_2 \leq 10b_3$. If the inequality $newcst_{34} \leq newcst_{23}$ is expanded and simplified in the same way it leads to the inequality $9 + 10b_3 \leq 10b_2$. These two equations are contradictory, therefore there can be no set of biases $[b_1\ b_2\ b_3\ b_4]$ that will generate the MST containing the links $(1,2)$, $(2,3)$ and $(3,4)$.

It would seem that this coding scheme has failed the final criteria. However on closer inspection we can see that this is not the case. The reason for this is that although the set of links qualify as a tree under its formal definition, these links actually represent a linear bus

topology and therefore need not be considered.

3.4 The Fitness Function and the Evolutionary Algorithm

Along with the coding scheme that needs to be devised for any problem, is the fitness function. The purpose of the fitness function is to return some fitness or figure of merit which is in some way a measure of how good the individual is at solving the problem. In many cases it is easy to determine what the fitness function is. Numerical optimisation problems would have the fitness value be the value of the function being optimised. However this is not always the case and as in the network topology optimisation problem a more complicated fitness function is often needed.

3.5 Fitness Function

The final step in explaining the mechanisms used to solve the problem of network topology optimisation for a centralized data network is a description of the fitness function.

As mentioned previously, the LNB coding scheme uses two steps to convert from the bias notation to a representation that can be used to calculate an individual's fitness. It firstly modifies the cost matrix using the biases along with some other variables. Then it finds an MST of this matrix using Prim's algorithm.

The output of Prim's algorithm is an array of links that define the tree. The endpoints of each of these links occupy adjacent odd and even numbered positions in the array. Therefore the MST of a four node network could be represented by the following array of links.

$$MST = [1\ 2\ 2\ 3\ 2\ 4]$$

The links in this tree are $(1, 2)$, $(2, 3)$ and $(2, 4)$.

The first step to be performed on a tree is to see whether it violates the traffic constraint.

If it does it is assigned a fitness of infinity. If it doesn't the cost of the tree is calculated by adding together the costs of all the links defining the tree. So in the above example the cost of the tree would be,

$$cost = cst(1,2) + cst(2,3) + cst(2,4)$$

Where *cst* is the original cost matrix. A more detailed description of how this is implemented in Matlab can be found in Appendix F.

Chapter 4

Results and Conclusions

This chapter introduces the results from this thesis and the conclusions that can be drawn from these results. An explanation of all the algorithms used in achieving these results can be found in Appendix F. A listing of the Matlab code associated with these algorithms can be found in Appendix G.

Table 4.1 shows the four different networks that PBIL and the CMST algorithm were tested on. The first column is the number of nodes in the network. The second column is the value of the maximum amount of traffic that is allowed to flow in any subtree of the network, the constraint.

In all of these problems the only traffic information that was taken into account was the traffic flowing from the root node to every other node in the network. These values were all arbitrarily set equal to 1. The cost matrix was randomly generated each time. It was not necessary to use more meaningful input information for these algorithms since they are both subjected to the same input information and the only thing that we are concerned with is

Nodes	Constraint
6	3
12	6
18	9
24	12

Table 4.1: The Number of Nodes and the Constraint used in each Network

their relative performance. Below is a listing of the cost matrices used as inputs to the four different problems.

$$cst = \begin{pmatrix} \infty & 7 & 8 & 11 & 14 & 17 \\ 7 & \infty & 6 & 5 & 10 & 12 \\ 8 & 6 & \infty & 10 & 7 & 14 \\ 11 & 5 & 10 & \infty & 8 & 8 \\ 14 & 10 & 7 & 8 & \infty & 9 \\ 17 & 12 & 14 & 8 & 9 & \infty \end{pmatrix}$$

$$cst = \begin{pmatrix} \infty & 1 & 7 & 9 & 3 & 5 & 1 & 8 & 5 & 8 & 7 & 2 \\ 1 & \infty & 8 & 7 & 4 & 3 & 7 & 3 & 3 & 6 & 9 & 6 \\ 7 & 8 & \infty & 7 & 2 & 2 & 7 & 3 & 2 & 8 & 3 & 8 \\ 9 & 7 & 7 & \infty & 5 & 9 & 8 & 4 & 2 & 2 & 2 & 6 \\ 3 & 4 & 2 & 5 & \infty & 2 & 2 & 5 & 6 & 3 & 4 & 9 \\ 5 & 3 & 2 & 9 & 2 & \infty & 1 & 6 & 7 & 7 & 8 & 5 \\ 1 & 7 & 7 & 8 & 2 & 1 & \infty & 8 & 1 & 2 & 6 & 2 \\ 8 & 3 & 3 & 4 & 5 & 6 & 8 & \infty & 5 & 2 & 2 & 9 \\ 5 & 3 & 2 & 2 & 6 & 7 & 1 & 5 & \infty & 3 & 6 & 4 \\ 8 & 6 & 8 & 2 & 3 & 7 & 2 & 2 & 3 & \infty & 4 & 2 \\ 7 & 9 & 3 & 2 & 4 & 8 & 6 & 2 & 6 & 4 & \infty & 6 \\ 2 & 6 & 8 & 6 & 9 & 5 & 2 & 9 & 4 & 2 & 6 & \infty \end{pmatrix}$$

$$cst = \begin{pmatrix}
\infty & 5 & 4 & 9 & 8 & 7 & 6 & 3 & 1 & 8 & 2 & 7 & 6 & 4 & 4 & 3 & 6 & 6 \\
5 & \infty & 3 & 4 & 3 & 2 & 6 & 1 & 4 & 9 & 6 & 6 & 2 & 5 & 7 & 2 & 4 & 1 \\
4 & 3 & \infty & 5 & 9 & 1 & 7 & 5 & 5 & 3 & 4 & 6 & 9 & 7 & 6 & 3 & 2 & 1 \\
9 & 4 & 5 & \infty & 3 & 5 & 5 & 5 & 6 & 3 & 2 & 7 & 5 & 3 & 8 & 7 & 9 & 2 \\
8 & 3 & 9 & 3 & \infty & 2 & 3 & 5 & 7 & 4 & 6 & 1 & 6 & 2 & 7 & 5 & 2 & 7 \\
7 & 2 & 1 & 5 & 2 & \infty & 8 & 5 & 6 & 8 & 4 & 3 & 8 & 4 & 8 & 6 & 7 & 1 \\
6 & 6 & 7 & 5 & 3 & 8 & \infty & 2 & 5 & 8 & 8 & 3 & 1 & 9 & 7 & 8 & 9 & 5 \\
3 & 1 & 5 & 5 & 5 & 5 & 2 & \infty & 1 & 3 & 2 & 6 & 6 & 9 & 4 & 3 & 2 & 8 \\
1 & 4 & 5 & 6 & 7 & 6 & 5 & 1 & \infty & 7 & 9 & 6 & 7 & 3 & 8 & 7 & 6 & 3 \\
8 & 9 & 3 & 3 & 4 & 8 & 8 & 3 & 7 & \infty & 1 & 6 & 5 & 2 & 6 & 3 & 2 & 3 \\
2 & 6 & 4 & 2 & 6 & 4 & 8 & 2 & 9 & 1 & \infty & 4 & 1 & 2 & 4 & 4 & 4 & 2 \\
7 & 6 & 6 & 7 & 1 & 3 & 3 & 6 & 6 & 6 & 4 & \infty & 5 & 5 & 5 & 2 & 3 & 7 \\
6 & 2 & 9 & 5 & 6 & 8 & 1 & 6 & 7 & 5 & 1 & 5 & \infty & 2 & 5 & 5 & 5 & 7 \\
4 & 5 & 7 & 3 & 2 & 4 & 9 & 9 & 3 & 2 & 2 & 5 & 2 & \infty & 4 & 3 & 8 & 5 \\
4 & 7 & 6 & 8 & 7 & 8 & 7 & 4 & 8 & 6 & 4 & 5 & 5 & 4 & \infty & 7 & 8 & 3 \\
3 & 2 & 3 & 7 & 5 & 6 & 8 & 3 & 7 & 3 & 4 & 2 & 5 & 3 & 7 & \infty & 7 & 7 \\
6 & 4 & 2 & 9 & 2 & 7 & 9 & 2 & 6 & 2 & 4 & 3 & 5 & 8 & 8 & 7 & \infty & 4 \\
6 & 1 & 1 & 2 & 7 & 1 & 5 & 8 & 3 & 3 & 2 & 7 & 7 & 5 & 3 & 7 & 4 & \infty
\end{pmatrix}$$

$$cst = \begin{pmatrix}
\infty & 6 & 2 & 8 & 7 & 2 & 7 & 7 & 3 & 4 & 2 & 6 & 8 & 3 & 7 & 2 & 7 & 1 & 2 & 6 & 2 & 3 & 8 & 2 \\
6 & \infty & 4 & 6 & 5 & 8 & 6 & 2 & 2 & 2 & 3 & 6 & 7 & 8 & 3 & 4 & 5 & 6 & 9 & 8 & 3 & 1 & 8 & 8 \\
2 & 4 & \infty & 2 & 6 & 5 & 6 & 2 & 7 & 6 & 2 & 1 & 6 & 8 & 4 & 4 & 2 & 4 & 5 & 2 & 6 & 1 & 3 & 4 \\
8 & 6 & 2 & \infty & 8 & 6 & 4 & 4 & 7 & 7 & 3 & 9 & 4 & 8 & 7 & 8 & 2 & 1 & 7 & 8 & 6 & 8 & 7 & 1 \\
7 & 5 & 6 & 8 & \infty & 5 & 2 & 7 & 1 & 8 & 7 & 7 & 6 & 9 & 1 & 1 & 5 & 2 & 2 & 9 & 5 & 8 & 8 & 2 \\
2 & 8 & 5 & 6 & 5 & \infty & 3 & 3 & 7 & 5 & 2 & 4 & 6 & 8 & 3 & 5 & 3 & 6 & 5 & 5 & 4 & 8 & 4 & 6 \\
7 & 6 & 6 & 4 & 2 & 3 & \infty & 4 & 2 & 8 & 5 & 1 & 6 & 7 & 3 & 7 & 6 & 4 & 1 & 3 & 5 & 5 & 7 & 9 \\
7 & 2 & 2 & 4 & 7 & 3 & 4 & \infty & 8 & 6 & 9 & 2 & 3 & 8 & 8 & 3 & 4 & 6 & 5 & 5 & 4 & 2 & 4 & 2 \\
3 & 2 & 7 & 7 & 1 & 7 & 2 & 8 & \infty & 4 & 3 & 9 & 1 & 5 & 3 & 9 & 6 & 7 & 8 & 4 & 2 & 5 & 5 & 4 \\
4 & 2 & 6 & 7 & 8 & 5 & 8 & 6 & 4 & \infty & 3 & 8 & 9 & 6 & 7 & 9 & 2 & 4 & 5 & 8 & 8 & 8 & 5 & 3 \\
2 & 3 & 2 & 3 & 7 & 2 & 5 & 9 & 3 & 3 & \infty & 6 & 3 & 7 & 4 & 9 & 8 & 4 & 8 & 3 & 5 & 8 & 2 & 4 \\
6 & 6 & 1 & 9 & 7 & 4 & 1 & 2 & 9 & 8 & 6 & \infty & 6 & 8 & 7 & 7 & 8 & 4 & 9 & 4 & 2 & 6 & 4 & 2 \\
8 & 7 & 6 & 4 & 6 & 6 & 6 & 3 & 1 & 9 & 3 & 6 & \infty & 5 & 7 & 4 & 4 & 2 & 5 & 7 & 6 & 5 & 7 & 4 \\
3 & 8 & 8 & 8 & 9 & 8 & 7 & 8 & 5 & 6 & 7 & 8 & 5 & \infty & 6 & 7 & 2 & 6 & 2 & 8 & 9 & 5 & 3 & 4 \\
7 & 3 & 4 & 7 & 1 & 3 & 3 & 8 & 3 & 7 & 4 & 7 & 7 & 6 & \infty & 3 & 7 & 4 & 8 & 6 & 8 & 7 & 3 & 6 \\
2 & 4 & 4 & 8 & 1 & 5 & 7 & 3 & 9 & 9 & 9 & 7 & 4 & 7 & 3 & \infty & 6 & 2 & 3 & 7 & 6 & 4 & 2 & 6 \\
7 & 5 & 2 & 2 & 5 & 3 & 6 & 4 & 6 & 2 & 8 & 8 & 4 & 2 & 7 & 6 & \infty & 4 & 6 & 6 & 3 & 2 & 7 & 1 \\
1 & 6 & 4 & 1 & 2 & 6 & 4 & 6 & 7 & 4 & 4 & 4 & 2 & 6 & 4 & 2 & 4 & \infty & 6 & 4 & 2 & 5 & 6 & 4 \\
2 & 9 & 5 & 7 & 2 & 5 & 1 & 5 & 8 & 5 & 8 & 9 & 5 & 2 & 8 & 3 & 6 & 6 & \infty & 2 & 1 & 4 & 6 & 2 \\
6 & 8 & 2 & 8 & 9 & 5 & 3 & 5 & 4 & 8 & 3 & 4 & 7 & 8 & 6 & 7 & 6 & 4 & 2 & \infty & 2 & 8 & 4 & 5 \\
2 & 3 & 6 & 6 & 5 & 4 & 5 & 4 & 2 & 8 & 5 & 2 & 6 & 9 & 8 & 6 & 3 & 2 & 1 & 2 & \infty & 9 & 2 & 5 \\
3 & 1 & 1 & 8 & 8 & 8 & 5 & 2 & 5 & 8 & 8 & 6 & 5 & 5 & 7 & 4 & 2 & 5 & 4 & 8 & 9 & \infty & 2 & 3 \\
8 & 8 & 3 & 7 & 8 & 4 & 7 & 4 & 5 & 5 & 2 & 4 & 7 & 3 & 3 & 2 & 7 & 6 & 6 & 4 & 2 & 2 & \infty & 3 \\
2 & 8 & 4 & 1 & 2 & 6 & 9 & 2 & 4 & 3 & 4 & 2 & 4 & 4 & 6 & 6 & 1 & 4 & 2 & 5 & 5 & 3 & 3 & \infty
\end{pmatrix}$$

The CMST algorithm is such that if you run it more than once you will get the same answer every time. Therefore this algorithm was only run once. PBIL, however, is not guaranteed to achieve the same answer every time. Therefore, in order to make sure that PBIL does perform as well as it possibly can, for each network this algorithm was run ten times and the best solution recorded.

As mentioned in the introduction to this thesis the two algorithms are to be compared on three points. They are their flexibility, the time taken to achieve the solution and the actual solution they produce.

The flexibility of an algorithm is its ability to adapt to any changes made in the problem

Nodes	CMST	PBIL
6	41	49
12	18	21
18	25	35
24	32	37

Table 4.2: The Results Obtained using the two Algorithms

statement. For instance if a further constraint was imposed on the network such that there should not be more than a certain delay between any two nodes, would the algorithms be able to adapt to handle this change. It was found that both PBIL and the CMST algorithm would be able to adapt well. This is due to the nature of their respective operations. PBIL evaluates an entire network at a time. Any network that violates any of the constraints could be assigned a fitness of infinity. The CMST algorithm builds a network one link at a time. Therefore any link added to the subgraph that violates any of the constraints would be discarded. Then the next smallest link in the graph could be considered for inclusion in the tree.

The second aspect on which the two algorithms are compared is the time they take to produce a solution. In this comparison the CMST algorithm outperformed PBIL drastically. This is mainly because the nature of PBIL is such that it is a generational algorithm. This means that it needs to evaluate a lot of different networks that have evolved over a relatively long period of time in order to reach a good solution. The CMST algorithm on the other hand, only evaluates whether adding a link to the current subtree will violate any of the constraints. Depending on how efficient the algorithm is this can be reduced to a very simple addition, as was the case in the version of the CMST algorithm produced for this thesis. As mentioned earlier PBIL is not guaranteed to find its best solution on every run therefore multiple runs would need to be made in order to ensure a good solution. This would further add to the time difference between the two algorithms.

The final aspect of the comparison is the solution that each algorithm produced. Table 4.2 gives the minimum cost obtained by the two algorithms for each of the four different networks.

In some of these problems PBIL's performance was almost as good as the CMST algorithm's performance. As can be seen from table 4.2 this is not related to the size of the network. The only reason this is so is because of the uncertainty in the results generated by PBIL.

Figures 4-1 to 4-8 are the networks produced by the algorithms for each of the four input problems.

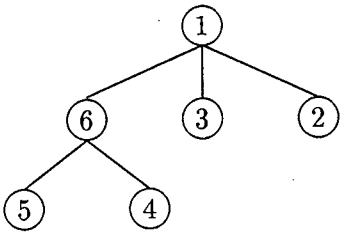


Figure 4-1: The PBIL Generated Six Node Network

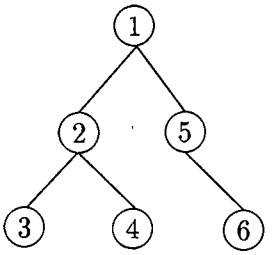


Figure 4-2: The CMST Generated Six Node Network

As can be seen from all these three aspects, especially the final one, the performance of PBIL in optimising this problem is very poor. In order to ensure that it was not just PBIL that could not handle this type of problem, the problem was optimised using another Evolutionary Algorithm called Real Coded CHC[1] ([23]). The performance of the CMST algorithm was again better than this Evolutionary Algorithm.

In conclusion it would seem that the Evolutionary Algorithm is not a useful tool in solving this problem. However, each Evolutionary Algorithm is able to solve problems belonging to certain classes. For instance a problem that PBIL was able to solve well may not be able to be solved by Real Coded CHC (see chapter 5). There is no universal optimiser that has been found. Therefore there may be an algorithm capable of improving on the performance of the heuristics in this problem but it has not been found in this thesis.

An important fact that should be noted is that just because an evolutionary algorithm is good

[1]Real Coded CHC is also meant to be a high performance optimiser that has outperformed the SGA in many problems

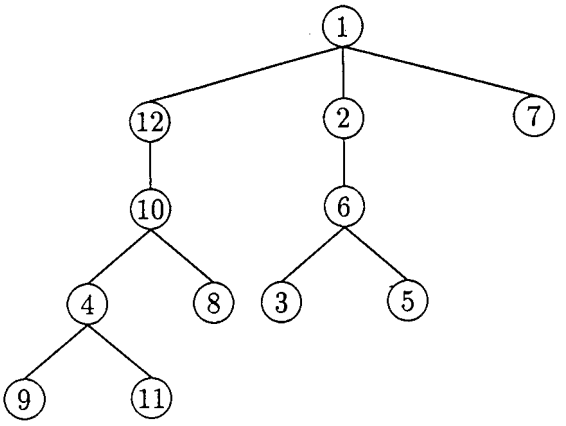


Figure 4-3: The PBIL Generated Twelve Node Network

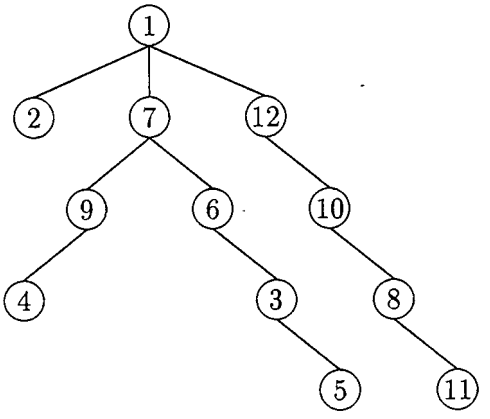


Figure 4-4: The CMST Generated Twelve Node Network

at solving NP-complete problems it does not mean it will be able to solve every problem of this type. In other words if an evolutionary algorithm is able to successfully solve the problem of minimising the cost of a LAN (see chapter 5), there is no guarantee that if the problem statement is changed to reflect a slightly different type of network that the Evolutionary Algorithm will be able to solve this new problem as well. This is an important fact that should be recognised by anyone wishing to perform further research based on this thesis.

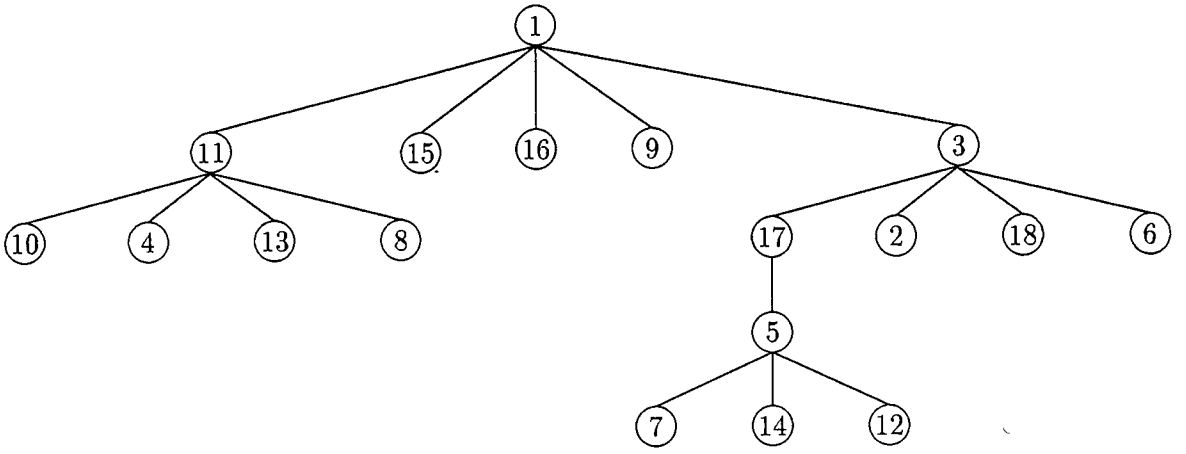


Figure 4-5: The PBIL Generated Eighteen Node Network

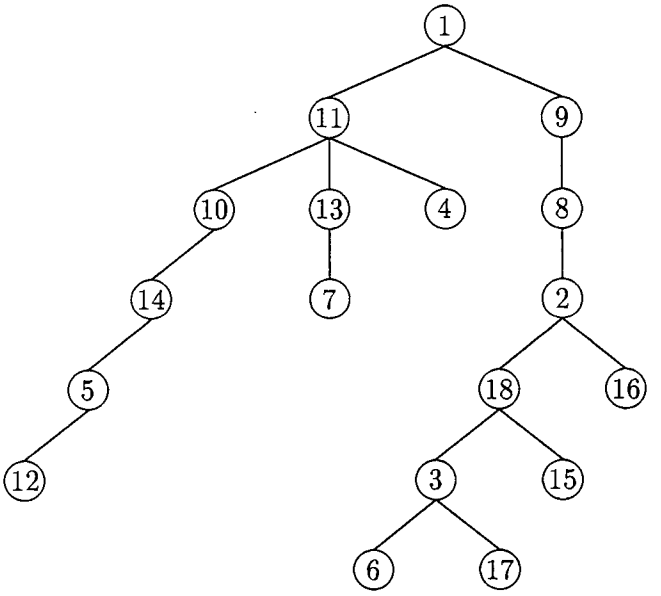


Figure 4-6: The CMST Generated Eighteen Node Network

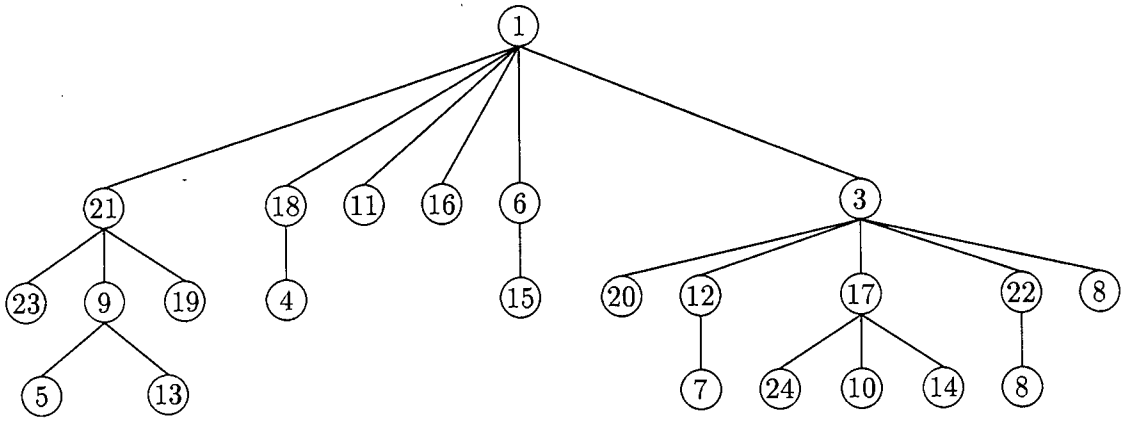


Figure 4-7: The PBIL Generated Twenty-Four Node Network

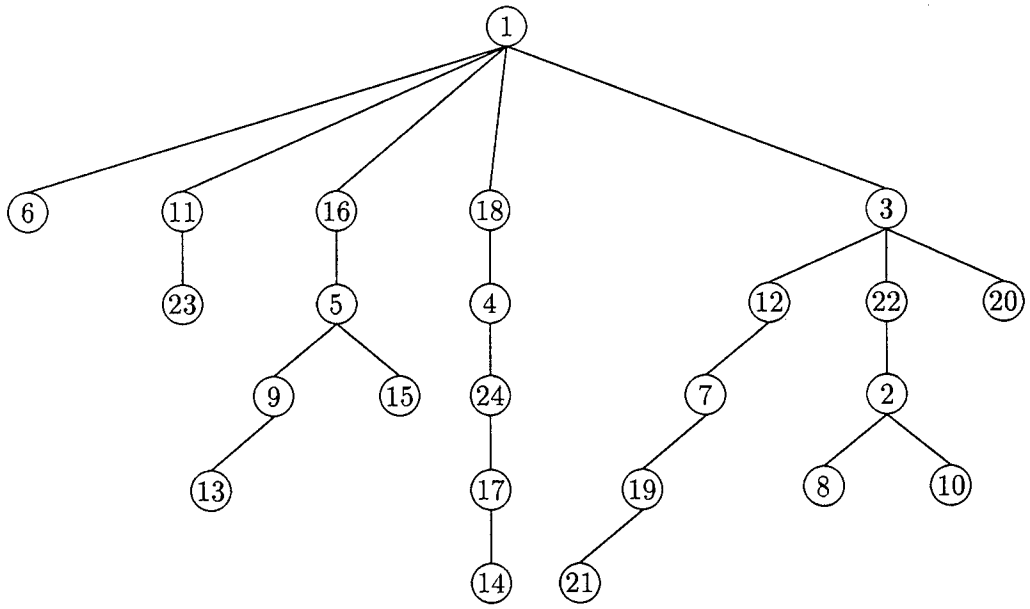


Figure 4-8: The CMST Generated Twenty-Four Node Network

Chapter 5

Future Research Possibilities

The goal of this thesis was to find a new, better method of optimising a tree structured centralized data network. Once this had been achieved the problem statement would then have been modified so that it could be used in the design of a network interconnecting LANs. There are already Evolutionary Algorithms that have been used to achieve this, however the idea here was instead of copying someone else's work, to come up with a new method that perhaps offers an improvement over previous methods used. Then any future research possibilities could be devoted to this area.

Unfortunately the first step, the design of a new, better method of optimising a tree structured centralized data network was not achieved. As can be seen from the previous chapter the CMST algorithm outperformed PBIL in this regard. There is therefore no point in spending more time devoted to researching this problem.

However, so that there is still some mention made of the interconnecting of LANs, the future research possibilities will be based on another method already developed. This is the method developed by Palmer and Kershenbaum ([8]).

The first section of this chapter introduces the problem statement and shows some results that have been obtained using PBIL and the LNB coding scheme that were introduced in this thesis. The next section discusses areas of future research that could be related to this problem.

5.1 The Problem Statement

The problem is to design a tree structured network interconnecting LANs. The LANs will generally be in a simple topology, such as a ring or star, and will be connected to each other either by bridges or routers. If routers are used then the network will have switching capability.

The inputs to this problem are also the cost and traffic matrices, the information in these matrices can be based on the many different models mentioned in section 1.4. The meanings of the elements of these matrices are different to that used in the problem defined in this thesis. In this case the elements of the cost matrix are not fixed values but values dependant on the amount of traffic sent between each node. The elements of the traffic matrix represent the maximum amount of traffic that is sent between each pair of nodes during the course of a day or week. However, only traffic in one direction is taken into account. This is not a very accurate method of determining cost since in a network such as this the amount of traffic flowing in one direction between two nodes may be related to but is not the same as the traffic flowing in the other direction. More about this is mentioned in the next section on areas of future research.

The actual problem statement is to find a tree that will minimise the cost, z, such that

$$z = \sum_{i,j} cst(i,j) traf_{ij}$$

Where $cst(i,j)$ is the cost per unit of bandwidth of all the links interconnecting nodes i and j, and $traf_{ij}$ is the traffic flowing between node i and node j. This value is extracted directly from the traffic matrix. This problem is also subjected to the constraints that no cycles are to be included and that the tree should be a spanning tree.

Figure 5-1 shows a four node tree structured network. This network will be used to explain the cost function in more detail.

Consider nodes 3 and 4. The cost of transmitting data between these two nodes is shown below.

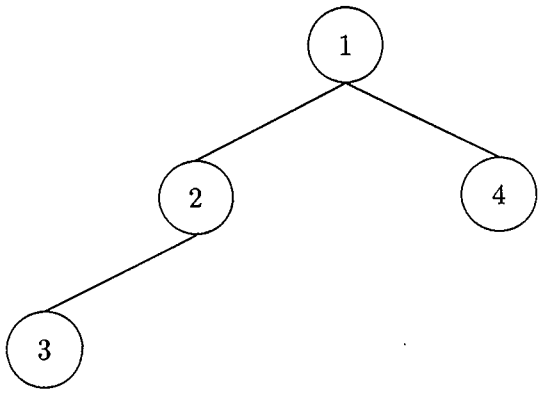


Figure 5-1: A Four Node Network

$$cost = traf_{34}(cst_{32} + cst_{21} + cst_{14})$$

Where $traf_{34}$ is the traffic flowing between nodes 3 and 4 and cst_{ab} is the cost per unit of bandwidth of transmitting data down link (a, b). This calculation is repeated once, i.e. in only one direction, for every pair of nodes in the network and all these costs are then summed together to get the overall cost of the network.

The problem statement was optimised using a combination of PBIL and the LNB representation discussed in this thesis. The input matrices were those used in Berry et al. ([9]). The problem that was attempted was the optimisation of the six node network. This problem was successfully optimised and the minimum cost of the network was found to be 534 units (It should be noted here that Real-Coded CHC was unable to optimise this problem).

5.2 Areas of Future Research

The first problem that needs to be tackled is to compare the above mentioned method of optimising a network to heuristic methods already in use. However, the fact that the six node network was successfully optimised together with Palmer and Kershenbaum's results

([8]) showing an improvement using an Evolutionary Algorithm over a heuristic, means that this problem can be attempted with a lot of confidence.

The next area of improvement could be to modify the cost function so that it more realistically reflects the cost of the network.

The first way of doing this would be to change the cost function so that traffic in both directions is introduced. However, this is not sufficient, the traffic flowing in both directions between any other two nodes whose path include that particular link should also be included. This makes the problem far more complicated and would introduce many more variables.

Another way of improving the cost function would be to investigate how redundancy could be added to the tree. In other words a constraint could be introduced that would specify that between all or certain of the nodes there would have to be more than one path that the data could travel along. This would of course introduce cycles into the graph and would mean that the network would no longer be a tree. However, when introducing these redundancies the tree network could be generated as usual, i.e. using the LNB representation, then another method could be used to determine the best way the redundant links could be added and it could then assign some figure of merit to this. Then the full network could be evaluated in the usual way.

Another constraint that could be added is to restrict the degree of a node to be below a certain number. The degree of a node is the number of edges that are incident on it.

Appendix A

The Genetic Algorithm

This chapter is an explanation of the Genetic Algorithm and the hypotheses which attempt to explain why it is so successful at problem solving. Even though the GA is not the Evolutionary Algorithm chosen to perform the network optimisation discussed in this thesis it is important to understand the basics of the GA before PBIL is explained in the next chapter. A lot of the aspects of PBIL are explained in relation to the GA.

All explanations of the GA will be done using the traditional Genetic Algorithm. This is the GA in its most basic form. Although there are many other types of GAs that perform better, all of them are based on the traditional GA.

The first section is an introduction to the Genetic Algorithm. The second section describes how it works. It includes a description and explanation of the algorithm itself. The final section in this chapter describes certain theories explaining why they work.

A.1 Introduction to Genetic Algorithms

The use of GAs to solve search and optimisation problems began in the 1970's with John Holland's pioneering work. Since then there has been a great deal of research in this area, the majority of this work being initiated by world-wide academic institutions. It is only recently that industry has seen the advantages of the GA and some companies have set up

labs specifically for the purpose of developing GAs to solve their problems. Problems that were once considered "impossible" or very difficult to solve can now be solved by the GA.

The basic operating principles of the GA has been described in many texts (including [18], [19], [20]). The GA is inspired by the principles of natural selection and "survival of the fittest" first introduced by Charles Darwin in his book *The Origin of the Species.* By working in a method similar to these principles the GA is able to solve real world problems.

In nature, individuals in a population all need to adapt to their surroundings and learn to survive. For instance, an individual that does not learn that a certain type of species is a predator, will get eaten and die. The individuals that do learn to adapt will have relatively large numbers of offspring. The poorer performing individuals will have a smaller amount of offspring or may even have none at all. The result of this would be that the genes from the individuals that are better able to adapt would be more likely to dominate the next generation. What may also happen is that genes from two ancestors may combine to form a "superfit" individual who is better able to adapt to its surroundings than its parents.

The way a GA works is based on the above mentioned principles. It works with a population of individuals. Each of these individuals is a candidate solution to a given problem that is being optimised. These individuals are each assigned a fitness according to how good a solution they are. This fitness is calculated by using the individual as an input to the fitness function. The fitter individuals are given opportunities to mate with each other, thus producing new offspring. Each of these new offspring will have qualities from each of their parents. The least fit individuals in the population are less likely to get an opportunity to mate and will die out.

Therefore, a new population of candidate solutions is generated by selecting certain of the fitter individuals and mating them, their offspring forming the new generation. Due to the nature of the mating procedure, this new generation would generally contain a higher proportion of the characteristics possessed by the fitter members of the parent population. This is continued throughout many generations, and over time good characteristics are spread throughout the population and mixed with good characteristics of other individuals. By ensuring that the fitter individuals have a higher probability of mating, the better areas of the search space are explored. In this way a good solution can be found.

What makes the GA a powerful technique is that it is robust and can successfully deal with a wide range of problems without knowing anything about the search space it is trying to find the optimum of. Due to the nature of the GA it is not guaranteed to find the global optimum of the search space. Instead it is more likely to find a reasonably good solution in a reasonable amount of time.

A.2 How GAs Work

There are certain aspects of a GA which are directly related to the algorithm itself and are not necessarily problem specific (although these parameters may be changed to improve the performance of a GA in solving a particular problem). They are the selecting of certain individuals for reproduction, the recombining of those individuals to generate an intermediate population and the selecting of certain individuals to survive into the next generation. However before being able to use a GA to solve a particular problem there are certain things that need to be done. The first is to develop a coding scheme. This is a way of mapping the problem into a domain that the GA can understand. The next is the development of a fitness function. This is a method of evaluating how good a solution each coded individual is. These elements are explained in the next two sections. The following section explains the operation of the GA itself by means of an algorithm.

A.2.1 Coding Scheme

The coding used in a problem maps the problem into a domain that the GA can understand and evaluate. Each candidate solution is represented as a set of parameters, this is often referred to as a chromosome. If the problem is to optimise a computer network topology then the parameters would in some way represent that topology (see section 3.3).

Initially all research was directed at using binary values in these chromosomes and all the theory explaining why GAs work was developed with binary values in mind. Recently however, a lot of research has been performed using high-cardinality alphabets. The symbols in these alphabets could, for example, be real numbers or integers. Some papers have made a direct comparison between a real valued GA and its binary valued counterpart and found

the results for the real case to be better ([21], [22], [23], [25]). All discussions explaining how and why the GA works will be related to the binary valued chromosome as this is how the traditional GA operates.

Consider a three dimensional function that needs to be optimised called $F(x, y, z)$. Each of these real valued variables can be represented by 10 bits[1]. This would mean that each chromosome would be thirty bits long.

The terms used to define various elements within the GA are in keeping with its analogy to genetics, all taken from this area. The parameters represented by a chromosome are referred to as a genotype. In the above example the variables x, y and z would be the genotype. A phenotype is a particular solution. So, any candidate solution with values for x, y and z would be called a phenotype. Each bit within the phenotype is called a gene. The fitness of an individual depends on this phenotype.

A.2.2 Fitness Function

Along with the coding scheme that needs to be devised for any problem, is the fitness function. The purpose of the fitness function is to return some fitness or figure of merit which is in some way a measure of how good the individual is at solving the problem. In many cases it is easy to determine what the fitness function is. Numerical optimisation problems would have the fitness value be the value of the function being optimised. However this is not always the case and as in the network topology optimisation problem a more complicated fitness function is often needed.

A.2.3 An Algorithmic Description of the traditional GA

The algorithm below defines what is meant by the traditional GA.

begin

t = 0;

[1]The number of bits used will depend on the range of the variables and their required accuracy. See Appendix C for a more detailed explanation.

initialise P(t);
evaluate individuals in P(t);
while termination condition not satisfied do
t = t + 1;
selectr I(t) from P(t - 1);
recombine individuals in I(t) forming C(t);
evaluate individuals in C(t);
P(t) = C(t);
end
end

This section explains the operation of the traditional GA using this algorithm. A practical example is included so that each aspect of the algorithm can be clearly explained. The example used will be to minimise the one dimensional parabola. Clearly the GA is not needed to solve this problem. It is used just as an illustration. The function and its parameters appear below.

$$f(x) = x^2 \qquad x \in [-5.12, 5.11]$$

As can be easily seen this function has a minimum at $f(x) = 0$ when $x = 0$. This is a mathematical optimisation problem that involves real numbers. In order for the GA to be able to manipulate the chromosomes they need to consist of binary values. This can be done by assigning ten bits to represent x. This will lead to an accuracy of 0.01 (see Appendix C for details), which is acceptable since the range of possible values that x can represent is only specified to two decimal places.

The first step in the algorithm is to *initialise* P(t). At this point t, the generation number, is zero. This makes P(0) the initial population. P(0) is usually generated randomly with a binary 0 or 1 having an equal probability of being placed in any bit position. The size of the population may vary but a common value to use is 100 individuals. Clearly it is not possible to show all one hundred individuals when explaining a particular concept by relating it to the

mathematical optimisation problem. Therefore where this problem is used to clarify certain aspects of the algorithm it will only be done with an appropriate number of individuals. It is implied that the operations shown apply to the entire population.

The next step is to *evaluate* each of the individuals in P(0). This is done by converting each 10 bit binary number back to its real representation and then using this representation as input to the function that is being optimised. Two typical individuals in the population and their fitness values appear below.

$$\overbrace{0\ 1\ 0\ 1\ 0\ 0\ 0\ 1\ 0\ 0}^{x=-1.88} \qquad \text{fitness} = (-1.88)^2 = 3.53$$

$$\overbrace{0\ 0\ 0\ 0\ 1\ 1\ 0\ 1\ 0\ 1}^{x=-4.59} \qquad \text{fitness} = (-4.59)^2 = 21.07$$

The next step in the algorithm is to begin the loop. This loop repeats until a termination condition is met. This termination condition is not easy to specify. It may be that the loop terminates after a certain amount of function evaluations or after a certain amount of time has passed. Another termination condition is convergence. The way the GA should operate is that as the generations pass and the best solution ever found approaches the global optimum of the search space, the average fitness of the population should also approach this value. The formal definition of convergence is that a population is deemed to have converged when all the genes have converged. A gene has converged when 95% of all the population share the same value. In real world problems this will not always occur and therefore the best termination conditions to use are the first two.

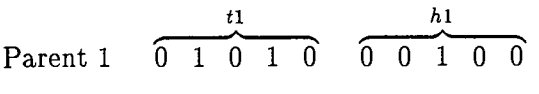
Once in the loop the next step is to increase the generation number, t. The *selectr* procedure selects individuals from the population of the previous generation, P(t-1), to form an intermediate population, I(t). I(t) is then the population of individuals that will be mated. The idea is that the highly fit individuals will get more reproductive opportunities than the unfit

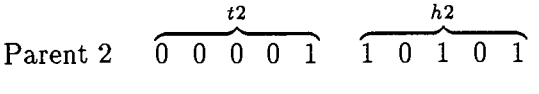
individuals who are likely to receive none at all. The size of I(t) is equal to the size of P(t-1).

There are many methods of selecting individuals to form the intermediate population. It is, however, beyond the scope of this thesis to go into all of them. The method that will be explained is called tournament selection. There are many variants of tournament selection, the simplest is binary tournament selection. In this algorithm two individuals are randomly picked from the population and the fittest one passed into the intermediate population. The two individuals are then replaced into the parent population and may be selected again. There is also the possibility of implementing larger tournaments where n individuals are chosen at random and the fittest one selected. This selection process is continued until the intermediate population is full.

The next step in the algorithm is to *recombine* the individuals in I(t) to form C(t), the child population. The members of I(t) are chosen randomly, without replacement, for recombination. There are two different operators that make up this recombination procedure, crossover and mutation. Each of these operators have many different variants, the simplest of each is explained below.

Crossover is the main operator of the GA. Two parents are randomly chosen from the intermediate population. Crossover is performed on these two parents with a predefined crossover probability (typically 0.6). If crossover is not performed the two parents are placed directly into the child population, C(t). If crossover is performed the two parents are each cut into two segments at a randomly chosen point in the chromosome. This will leave two tail segments, $t1$ and $t2$, and two head segments, $h1$ and $h2$. The two tail segments are swapped and in this way two children are generated. The process is shown below.





The two tail segments of the parents are swapped generating two children.

P1	0101000100	3.53
P2	0000110101	21.07
C1	0000100100	22.65
C2	0101010101	2.93
C1'	1000100100	0.13

Table A.1: Fitness' Before and After Crossover and Mutation

Child 1 $\overbrace{0\ 0\ 0\ 0\ 1}^{t2}\ \overbrace{0\ 0\ 1\ 0\ 0}^{h1}$

Child 2 $\overbrace{0\ 1\ 0\ 1\ 0}^{t1}\ \overbrace{1\ 0\ 1\ 0\ 1}^{h2}$

This operator is called single point crossover. The above process is repeated until the child population contains as many individuals as the parent population.

Mutation is performed on each individual separately, after crossover. This is also performed with a predefined mutation probability, this time much lower than that of crossover (typically 0.001). Mutation inverts a bit, chosen randomly, in an individual. This can be seen below where the bit in position 1 of the first child has been inverted.

$$\mathbf{1}\ 0\ 0\ 0\ 1\ 0\ 0\ 1\ 0\ 0$$

Next the entire population of children is *evaluated* and this then becomes the parent population, P(t), that will be used in the next generation.

The same two individuals have been used throughout this section. Table A.1 is a list of the two parents, the two children before and after mutation and their respective fitness'.

As can be seen from this table the two children, C2 and C1' have better fitness' than their parents. As the generations progress, these chromosomes and others like them will tend to dominate the population and eventually the global optimum should be reached.

A.3 Why GAs Work[2]

The fundamental operating principles of the algorithm have now been explained. It is important for a clear understanding of the GA and to be able to use it effectively to understand why it works. There is no definitive theory but there are hypotheses which can partially explain the GA's success. These are schema theory, the building block hypothesis and exploration and exploitation.

A.3.1 Schema Theory

The schema theory first introduced by Holland ([11]) was the first hypothesis attempting to explain how GAs work. A schema is a pattern of values from the alphabet {#, 0, 1} which can be found in any binary number. In this case the '#' is the don't care symbol and can represent either 0 or 1. Therefore the individual '10110' will contain amongst others the following schemata : '1##10', '1011#'. This can be clearly seen since the fixed bit positions in each of these schemata are also to be found in the original individual. These schemata are all subsets of the search space and in this instance would define a hyperplane in five dimensions running through the search space. The order of the schema would be the number of fixed bit positions it has. In the above two examples the order would be 3 and 4. The defining length of a schema is the distance between the outermost fixed positions. In the above two examples it would be 5 and 4.

The schema theorem explains the power of the GA in terms of these schemata. The number of times an individual will get the opportunity to reproduce in any generation is directly proportional to its fitness. Therefore the fitter individuals will have more of an influence on the next generation. This will be because they will contribute more of their genes to the next generation. An individual will have a high fitness due to the number and quality of its schemata. By passing more of these schemata onto the next generation there is more chance of finding even better solutions in the next generation.

Holland showed that the best way to explore the search space was to make sure that the

[2]Beasley, Bull and Martin [17]

number of reproductive trials allocated to each individual was directly proportional to its fitness relative to the rest of the population. In this way good schemata will receive an exponentially increasing number of opportunities to reproduce in subsequent generations. This is the schema theorem. Holland also showed that since each individual has a large number of schemata, the effective number of schemata being processed in each generation is of order N^3 (where N is the size of the population). This is called implicit parallelism and is a phenomenon apparently unique to the GA. Implicit parallelism allows the GA to search large areas of the search space simultaneously while only manipulating relatively few strings.

A.3.2 The Building Block Hypothesis

Goldberg ([19]) believes that the power of the GA lies in its ability to find good building blocks. A building block is a schema that has a short defining length and when incorporated into a candidate solution leads to improved performance. These building blocks are most likely to survive crossover without being broken up and will therefore be propagated through future generations at a rate proportional to the average fitness of the individuals containing them. He says that when designing a coding scheme it is important that the scheme encourages the formation of these building blocks. There are two important properties that the coding scheme needs in order to achieve this.

The first is that any related genes need to lie close to one another in the chromosome. The second is that there be a low level of epistasis or interaction between the genes. This means that the contribution of one particular gene to the overall fitness of the individual should not depend on the value of any other genes contained in the individual.

Unfortunately these conditions are not easy to meet. Often when designing a coding scheme it may not be possible to ensure that related genes are in close proximity. Furthermore, the GA is generally a useful optimisation tool when not much is known about the search space. Therefore even if there is a small relationship between any of the genes it may be impossible for the designer to determine this and hence meet the first condition.

A.3.3 Exploration and Exploitation

Any good optimisation algorithm needs to make use of two techniques in order to find a global optimum. These are exploration and exploitation. In exploration the algorithm explores new areas of the search space in order to find an area that will produce good results. In exploitation the algorithm exploits the knowledge that there are good solutions to be found in a certain area of the search space in an attempt to find better solutions. It should be quite clear that these two techniques are contradictory. In order for an algorithm to be effective it needs to perform a trade off between them.

On the one end of the scale is a purely random search which only engages in exploration. On the other end of the scale is the hillclimbing algorithm which exploits good solutions already found but performs no exploration of the search space. A combination of these two techniques would provide quite an effective search algorithm. It is, however, difficult to know how much to exploit a good solution before giving up and exploring further.

Holland ([11]) showed that a GA is able to perform both exploration and exploitation in an optimum way. However, he made some assumptions which would ensure that this were true in theory, but in practice many problems arise. The list of assumptions appears below.

1. The GA has an infinite population size
2. The fitness function is an accurate reflection of how good a particular solution is.
3. There is a low level of epistasis between the genes of a chromosome.

Having an infinite population in practice is clearly not possible. This will hinder the performance of the GA by introducing stochastic errors. The second and third assumptions can be true for specially designed laboratory functions but a designer has no control over real world functions and more often than not these assumptions will not be true.

Appendix B

Population Based Incremental Learning

This chapter introduces Population Based Incremental Learning (PBIL). PBIL is an optimisation technique that combines properties of the generational GA and Competitive Learning (see Appendix E). This combination produces a search technique that is far simpler than the GA and outperforms it in many optimisation problems. The performance improvement is in both accuracy and speed ([26]). It is for these reasons that PBIL was the evolutionary algorithm chosen to optimise the network topology problem.

This chapter begins with an introduction to the concepts of implicit and explicit parallelism. It then goes onto explaining the use of the probability vector in PBIL and how this relates to both implicit and explicit parallelism. Finally the PBIL algorithm is shown. This algorithm is self explanatory and no explanations are provided.

B.1 Explicit and Implicit Parallelism[1]

Genetic Algorithms have a property called implicit parallelism. This is the ability of the algorithms to search many areas of the search space in parallel. This is possible due to the

[1] Baluja [26]

fact that the GA maintains a population of about one hundred individuals each containing a number of schemata. These schemata are all evaluated simultaneously. It is however not easy for the GA to maintain useful parallelism throughout the search. The crossover operator does not introduce any new information into the population. This is because the two children are generated from segments of two parents. This may lead to premature convergence. Premature convergence occurs when several generations of runs of the GA do not provide any new, high quality solutions. Over the several generations the fitter individuals will come to dominate the population. This may result in the entire population being very similar, or having prematurely converged.

In the GA, the mutation operator introduces diversity into the population. It does this by randomly flipping a bit in a small percentage of individuals. This is one method of introducing explicit parallelism. Other methods that have been used are to implement parallel GAs which operate many subpopulations independently. There is some interaction between these subpopulations which takes the form of swapping chromosomes but this is limited and only done after a predefined number of generations has passed. Explicit parallelism is used to counter the effects of premature convergence and maintain diversity in the population.

Thus, the GA will not be able to maintain implicit parallelism as the generations pass. An algorithm that can maintain parallelism in the search by effectively introducing explicit parallelism should be able to solve problems in a shorter amount of time and provide better solutions.

B.1.1 The Probability Vector

PBIL uses a probability vector (PV) from which samples are drawn to create each new population. The coding scheme and fitness function used would be the same as that used for the GA. All the variables needed to solve the problem are converted to binary representation.

If the effects of mutation are ignored, the current population, t, can be used to generate an expected distribution of values for each bit position in the next generation, $t+1$. This distribution can be calculated by finding the probability of a particular value, j, being found in position i in the chromosome. The formula for this probability is shown below.

$$P(x_i = j) = \frac{\sum_{v \epsilon P(t) \& v_i = j} \text{fitness(v)}}{\sum_{v \epsilon P(t)} \text{fitness(v)}}$$

In the above formula, v is each candidate solution in the current generation, $P(t)$ is the $t-th$ generation and fitness(v) is the fitness of the individual v. Expressed in another way the equation states that the probability that j will appear in position i can be found by dividing the sum of the fitness' of all the vectors in generation t that have j in position i by the sum of all fitness' of all the vectors in generation t. This equation can be used in each bit position to obtain a probability vector for the next generation.

There are two observations that can be made about generating any population using this vector. The first is that many populations will have the same probability vector. The second observation is that the quality of the solutions produced in generation $t+1$ would be unlikely to be better than those found in generation t. This is because an assumption is made in using this vector. The assumption is that the value of each bit position is independent of the value of any other bit positions in a candidate solution vector. This is different to the crossover operator which maintains more information between the bit positions. This is because the children are produced from only two parents whereas in the probability vector case the children are generated by taking the entire parent population into account.

PBIL uses the probability vector to generate the population in each generation. The sampling of the probability vector to produce the next generation is termed a pool-wise mating operator. The values that the probability vector hold are the probabilities that the bit position will contain a 1.

The probability vector represents a prototype vector that has a high fitness in the search space. This is similar to Competitive Learning where when the training is complete, each set of weights related to each output unit will represent a vector from the classification group that the output unit represents. In order for this to occur the probability vector needs to be updated in a similar way to the weights in Competitive Learning. The probability update rule is shown below.

$$PV_i = PV_i(1 - LR) + LR * vector_i$$

Where PV_i is the probability of generating a 1 in position i, LR is the learning rate and $vector_i$ is the $i - th$ position in the vector towards which the probability vector was moved.

The next step is to decide which vector to use to update the probability vector. Each generation a number of candidate solutions are produced and evaluated. The idea of the algorithm is to find the global optimum of any search space. Therefore the probability vector is updated towards the individual that has the highest fitness in each generation.

As mentioned previously, in the early portions of the search the GA is able to exhibit implicit parallelism. This means that it is able to explore large areas of the search space simultaneously. In PBIL, all values of the probability vector are initially set equal to 0.5. This together with the fact that a whole population of individuals is generated ensures that implicit parallelism is also a property of PBIL in the early stages of the search. However, as the search progresses the values of the probability vector move away from 0.5 towards either 0.0 or 1.0. As this happens the similarity between the vectors produced will increase. Thus as generations pass PBIL is likely to suffer the same problem as the GA, premature convergence. One advantage of PBIL is that the learning rate can be changed, in this way the speed of the convergence of the population can be controlled.

In order to counter the effect of premature convergence the GA introduces the mutation operator. This helps to maintain the diversity in the population. The mutation operator has the same function in PBIL. Mutation can be performed by physically mutating the vectors created each generation or by mutating a value in the probability vector. In the version of PBIL developed for use in this thesis the second method was chosen.

There are also further extensions to the PBIL algorithm that can be done to improve its performance. In Competitive Learning there is a supervised classification procedure known as Kohonen's Learning Vector Quantization ([29]). The weight update procedure in this method works as follows. If the network correctly classifies an individual, the weights of the winning output are strengthened. If the classification is incorrect the weights are weakened.

This rule can be applied to updating the probability vector in PBIL. Instead of just updating it towards the best individual in any population, it can also be updated away from the worst individual. One way of doing this is to move the vector towards the complement of the worst vector. This is not very effective because as the search progresses the vectors generated will become more and more similar to each other. Therefore moving the probability vector away from the worst vector may also move it away from the best vector in a lot of bit positions. A solution is to move the probability vector away from the bits in the worst vector that are different to those in the best one.

B.2 The PBIL Algorithm

Below is a representation of the PBIL algorithm. It is done using a pseudo code similar to Matlab. It is straight forward and if read in conjunction with the above sections should not need any further explanations.

```
*** Initialise Probability Vector
PV(1,1:length) = 0.5;

While (termination condition not met)

        *** Generate and evaluate samples
        sample_vectors = samples_of_probability_vector(PV);
        fitness = evaluate(sample_vectors);

        best_vector = fittest_vector_in_population(sample_vectors);
        worst_vector = least_fit_in_population(sample_vectors);

        *** Update Probability Vector Towards the best_vector
        PV = PV*(1-LR) + best_vector*LR;
```

```
*** Update the Probability Vector away from the worst_vector
for loop=1:length
        if (best_vector(1,i) ≠ worst_vector(1,i))
        PV(i) = PV(i)*(1-NLR) + best_vector(i)*NLR;


*** Mutate the Probability Vector
for loop=1:length
        if (random(0,1] < MP)
        PV(i) = PV(i)*(1-MS) + random(0 or 1)*MS;
```

In the above algorithm there are a number of user defined variables. Below is list of them, their meanings and what values they were assigned to in the implementation of PBIL used in this thesis. The values were used by Baluja ([26]). The results he obtained using them were superior to any other set of values he tried.

length This is the length of the encoded solution and is problem specific.

LR This is the learning rate. The amount the probability vector is shifted towards the best solution. It was set to 0.1.

NLR This is the negative learning rate. The amount the probability vector is shifted away from the worst solution. It was set to 0.075.

MP This is the probability of mutation occurring in each bit position. It was set to 0.02.

MS This is the mutation shift. It is the amount by which mutation affects the probability vector. It was set to 0.05.

The PBIL algorithm used in this thesis was set to terminate after a certain number of generations had passed. The number of generations was set to 150.

Appendix C

Converting between Real and Binary Numbers[1]

This chapter explains a method of converting from a real number to a binary number and back again. This is important when using an Evolutionary Algorithm to solve a problem that uses real numbers as its inputs but where the Evolutionary Algorithm is only able to manipulate binary numbers.

Any algorithm that is used should be quick in converting between the two representations. This will limit the time the Evolutionary Algorithm takes to reach an acceptable solution. It should also allow the programmer to choose the level of accuracy he requires for his application. In this way the number of bits that is used to represent each variable can be controlled and hence the size of the search space can be limited. The algorithm described below meets both of these requirements.

Consider a real number x_i that can take on values in the range $a_i \leq x_i \leq b_i$. Consider also that the programmer has decided to represent each variable in binary form by n bits. Any number falling between $a_i + k(\frac{b_i - a_i}{2^n})$ and $a_i + (k+1)(\frac{b_i - a_i}{2^n})$ would be represented by the standard binary code for the integer k.

In this way the range of values that x_i can take on is divided into 2^n subdivisions each of

[1] Wright [21]

length $\frac{b_i-a_i}{2^n}$. Therefore the accuracy using this encoding would be $\frac{b_i-a_i}{2^n}$. The actual method of converting between a real number and its binary representation can best be described through an example. This appears below.

Consider a variable x that can take on values in the range $-5.12 \leq x \leq 5.11$. In other words, $a_i = -5.12$ and $b_i = 5.11$. This variable, x, is set equal to 1.02. If ten bits where chosen to represent each variable, the accuracy would be,

$$\text{accuracy} = \frac{b_i - a_i}{2^n} = \frac{5.11 + 5.12}{2^{10}} = 0.00999$$

For simplicity this value can be rounded to 0.01. This would also be the length of each subdivision. Since the range is only specified to two decimal places this accuracy would be sufficient.

The integer value k is generated by subtracting the lower end of the range, a_i, from the real number, x, and then dividing the result by the accuracy. This is shown below.

$$k = \frac{x - a_i}{\frac{b_i-a_i}{2^n}} = \frac{1.02 + 5.12}{0.01} = 614$$

The binary number is then just the standard binary form of k, this is shown below.

$$1\ 0\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ 0$$

In order to convert from this binary number back to the real number each step is reversed. In other words, first the binary number is converted back to its integer representation. This is shown below.

$$1\ 0\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ 0 = 614$$

This number is then multiplied by the accuracy of the conversion and a_i is then added to this

value. This results in the formula below.

$$x = k\frac{b_i - a_i}{2^n} + a_i = 614 * 0.01 - 5.12 = 1.02$$

Appendix D

Prim's Algorithm

Prim's algorithm finds the minimum spanning tree of a graph. It is used in the LNB coding scheme to convert from the bias representation of a network to a representation that can be more easily used to calculate the fitness of the network.

Below is an algorithm describing how it works, this algorithm was taken from Gibbons ([27]). Following the algorithm is a description explaining each step in more detail. (An explanation of how this was implemented in Matlab can be found in Appendix F).

1. $T \leftarrow \{\}$
2. $V' \leftarrow \{1\}$
3. for all $v \in (V - V')$ do $L(v) \leftarrow w((1, v))$
4. while $V' \neq V$
5. find an ω for which $L(\omega) = min\{L(v) | v \in (V - V')\}$ and denote the associated edge from V' to ω by e
6. $T \leftarrow T \bigcup \{e\}$
7. $V' \leftarrow V' \bigcup \{\omega\}$
8. for all $v \in (V - V')$ do
   if $w((v, \omega)) < L(v)$ then $L(v) \leftarrow w((v, \omega))$

The first two steps are the initialisation of the vectors T and V'. T will eventually contain all the links in the tree and V' keeps a record of what vertices have already been connected in the subgraph, it is initialised to 1 because this is the root node. V is a list of all the vertices in the subgraph and therefore $(V - V')$ is a list of all the vertices that are not part of the subgraph.

The third step creates a label, $L(v)$, for each vertex, v, not connected to the subgraph. At this point the graph is empty so $L(v)$ will contain the weights of the edges from each node to the root node, node 1.

Step four is the beginning of the loop and also contains the termination condition. The loop will end when the set of all nodes connected to the tree, V', equals the set of all possible nodes, V. In other words the loop will end when the tree becomes a spanning tree (it would actually be a minimum spanning tree).

Step five selects the next edge that is to be connected to the tree. It does this by selecting the minimum value contained in $L(v)$. This would be the smallest edge linking a node not in the subgraph to a node that is in the subgraph. This edge is denoted e.

The sixth step adds the selected edge, e, to the subgraph. The seventh step adds ω, the new node that is connected to the subgraph, to V', the set of all nodes connected to the tree.

The eighth step updates $L(v)$. $L(v)$ maintains a list of the minimum link costs from each node not in the subgraph to any node within the subgraph. It does this by selecting an unconnected node, v. It finds the weights of the links from v to each member of the subgraph and stores the smallest one. This is repeated for all nodes not in the subgraph. The loop then returns to step four. If the termination condition has been met, the loop ends and T is a minimum spanning tree.

Appendix E

Competitive Learning

Competitive Learning (CL) is used to group a number of points into clusters. The points will be centred on an ideal or prototype pattern for each of these clusters. This is done with no a priori knowledge of what each clusters' distinguishing characteristics will be. This type of clustering takes place in what is known as an unsupervised learning environment. The idea is that CL will be able to divide the vectors up into separate classes by determining the most important features of each class.

Figure E-1 is a diagram of a Competitive Learning network.

There are four elements that belong to this network. These are the input units, the output units, the inhibitory connections and the excitatory connections. The input units each correspond to a variable or a feature of the set of vectors that is being classified. The output units define which class a particular input vector belongs to. In this network only one output unit is allowed to be turned on at a time. This rule is enforced by the inhibitory connections, the connections between each of the output units. The excitatory connections contribute to the output of each output unit. This network is fully connected. This means that each input unit is connected, via the excitatory connections, to each of the output units.

The algorithm used to train the network is described below.

Each output unit has a weight associated with each of the inputs connected to it. Initially these weights are chosen randomly and normalised. More details about the normalisation

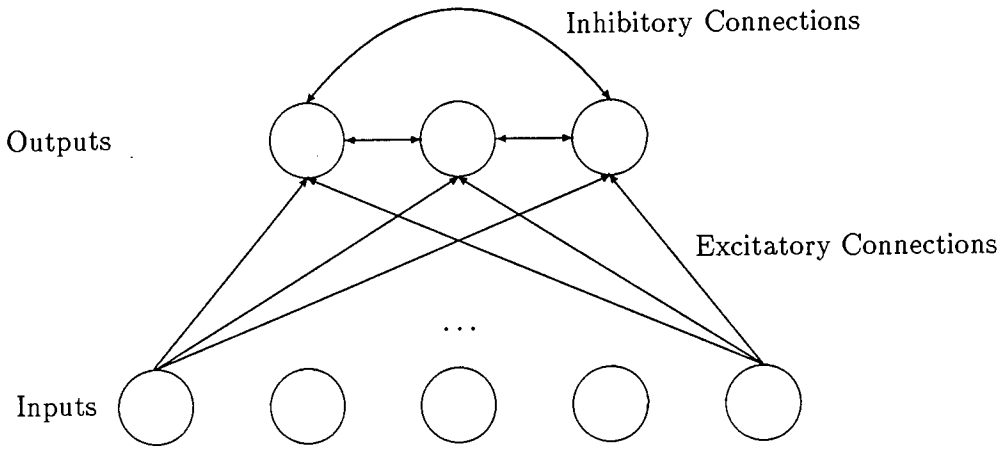


Figure E-1: A Competitive Learning Network

procedure can be found in Hertz, Krogh and Palmer ([29]).

The output of each output unit is calculated by multiplying each input unit with the weight associated with it and then summing all these values together. This is shown below in a formula.

$$output_i = \sum_j w_{ij} * input_j$$

Where w_{ij} is the weight associated with each $input_j$ and $output_i$. As mentioned previously in a CL network only one output can be active at a time. This is the output which has the highest value of $output_j$. The input vector is therefore classified to belong to $cluster_j$. During the training procedure the weights of the winning unit are updated in the direction of the input vector. The formula describing this is shown below.

$$\Delta w_{ij} = LR * (input_j - w_{ij})$$

Where LR is the user defined learning rate. The training procedure consists of constantly applying input vectors to the network until the weights have stabilised. There is, however,

no guarantee that the network will stabilise if the weights are updated after each input unit. Methods have been devised that ensure that stabilisation does occur. More information about these methods can be found in Hertz, Krogh and Palmer ([29]).

After the training is complete, the weights of each output unit will represent a prototype vector from each cluster. The highest values in a particular weight vector will represent the features that have been used in classifying each input into that particular cluster.

PBIL also works on the principle of defining a prototype vector. The updating of the prototype vector in PBIL is based on the weight update procedure in CL. Further extensions to CL can be used to improve the performance of PBIL. This is discussed in greater detail in Appendix B.

Appendix F

Explanation of Matlab Programs

This chapter explains how the algorithms described in the earlier chapters of this thesis are implemented in Matlab. The chapter is divided into two sections, the first explains how the CMST algorithm was implemented. The second explains how the PBIL algorithm as well as the coding scheme and fitness function were implemented. Each section begins with a tree diagram showing how each function relates to the other. For every function a brief explanation of what it does and what its inputs and outputs are is given. This is followed by an explanation of the function and should be looked at in conjunction with the Matlab code which can be found in Appendix G.

F.1 The CMST Algorithm

Figure F-1 is a diagram showing all the functions involved in the generation of the constrained MST. Each of these functions are explained in the order they appear in the tree, from top to bottom and from left to right.

F.1.1 The Information Structures

Before the functions can be explained, it is important that two information structures, or matrices, used in the functions are introduced. The first is *subtree* and the second *substat.*

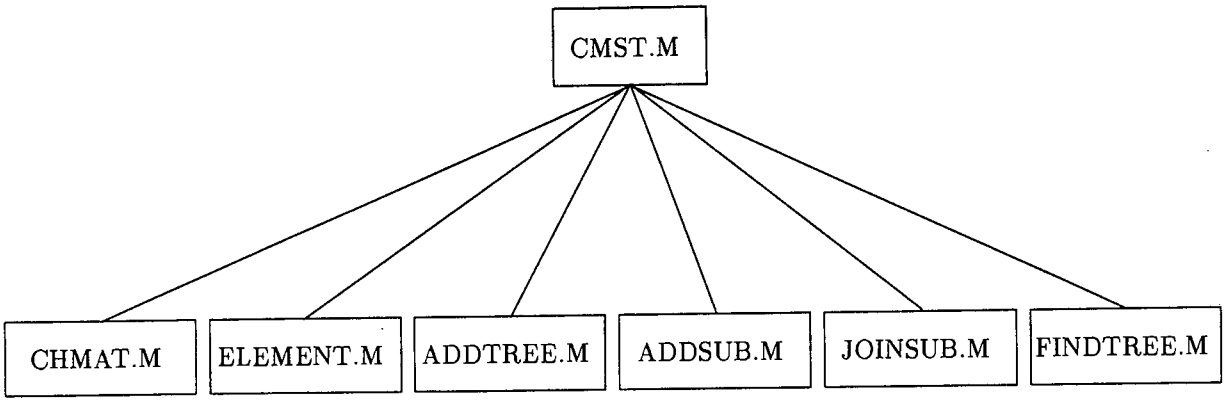


Figure F-1: The Functions used in the Implementation of the CMST Algorithm

These matrices hold information relating to the tree and enable the program to easily check whether any of the constraints are violated.

subtree is a matrix that contains the links in each subtree on a different line. *substat* is a matrix that contains certain statistics on all the subtrees of the tree stored in *subtree*. Each row in *substat* corresponds to a row in *subtree*. *substat* has three columns. The first is used to add links to the tree and contains a pointer to the next open space in the row. The second column is a flag which indicates whether that section of the trce contains the root node, node 1. A value of one in this column means that the subtree does contain the root node and a value of zero means that it does not. The third column contains the total amount of traffic flowing in that section of the tree.

The use of these matrices can be better explained by an example. Figure F-2 is a diagram of a subgraph of a tree.

As can be seen from figure F-2 there are three separate subtrees in this graph, two are rooted. The links $(1, 2)$ and $(2, 8)$ form the one rooted subtree, the link $(1, 6)$ forms the other rooted subtree. The third subtree is not rooted and contains the links $(3, 4)$ and $(4, 5)$. Table F.1 is a table showing what the matrix *subtree* would look like. Table F.2 is a table showing what the matrix *substat* would look like.

The three rows in table F.1 show the links contained in each of the three subgraphs. The first

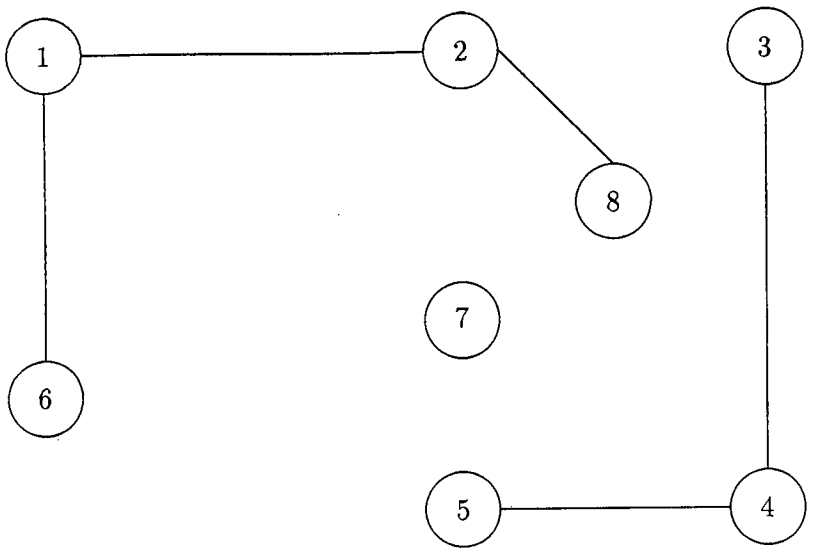


Figure F-2: A Subgraph of a Tree

1	2	2	8
1	6	0	0
3	4	4	5

Table F.1: The *subtree* Information Structure

column in table F.2 shows the next empty space in each row of *subtree*. The next column in *substat* is a flag denoting which row in *subtree* contains the root node. As can be seen it is the first two rows that are rooted. The final column in *substat* gives the amount of traffic flowing in each subgraph. The only traffic that is taken into account is the traffic from each node to the root node, this was set arbitrarily to 1. Therefore the traffic in the first subgraph is 2 units, in the second 1 unit and in the third 3 units.

F.1.2 CMST.M

Now that the two information structures have been explained the function CMST.M can be explained. This is the main function used in the generation of the CMST. For any explanation

5	1	2
3	1	1
5	0	3

Table F.2: The *substat* Information Structure

of why a particular step is taken or for any clarification of the algorithm see section 2.2.

The purpose of this function is to generate a constrained minimum spanning tree from the cost matrix while using the elements of the traffic matrix in calculating the constraint. The inputs to this function are *cst*, the cost matrix, *traf*, the traffic matrix, n, the number of nodes in the network and *thresh*, the maximum traffic allowed to flow in any rooted subtree. The output is *tree*, the constrained minimum spanning tree.

The first thing to do is to convert the cost matrix, *cst*, to another format. This is done using the function *chmat*. This function lists the elements of cst in ascending order with the row and column each element appears in listed in columns two and three. It puts all this information into a matrix *pop*.

The next step is to initialise all the variables. The variable, *pointer*, is a pointer to the next link in the matrix *pop* that is going to be considered. *links* is a variable that keeps a count of the number of links contained in the tree. When the number of links equals $n - 1$ then the loop terminates.

Next the matrices *subtree* and *substat* are initialised. *subtree* contains the row and column numbers of the smallest link in the graph. This information is contained in the matrix *pop*. The first column in *substat* is set equal to 3 since this is the next free position in *subtree*. The second column is set equal to one if either of the nodes extracted from *pop* is the root node, node 1. The third column in *substat* is the sum of the traffic flowing from node 1 to both of the two nodes just selected.

Next the loop begins. As mentioned earlier the termination condition is when the number of links added to the tree is one less than the number of nodes in the tree. This will occur when the tree is fully connected.

The next two nodes are then extracted from *pop*. These are called *node1* and *node2*. These two nodes are then used as an input to the procedure *element*. This procedure returns the row in which *node1* and *node2* appears in *subtree* and stores these values in *sub1* and *sub2* respectively. *element* will return the value 0 if the node is not contained in *subtree* or if the node is the root node, node 1. The variables *sub1* and *sub2* are used to determine where and whether the current two nodes, *node1* and *node2*, should be added to the tree.

There are four possible combinations of *sub1* and *sub2* that would result in four different actions occurring.

The first is if both *sub1* and *sub2* are zero. If this is the case, none of the nodes are already connected to the tree, or one of the nodes is the root node and the other cannot be found in the tree. First the traffic flowing from the root node to each of these nodes is added together. If this traffic value exceeds the traffic threshold, the two nodes are discarded. If it doesn't the two nodes are sent to a function *addtree*. *addtree* inserts the two nodes on a new row in the matrix *subtree* and updates *substat* accordingly. Then the variable *links* is incremented and the loop begins again.

The second combination is when *sub1* is zero, i.e. *node1* cannot be found in the tree or *node1* is the root node, and *sub2* is nonzero, i.e. *node2* is already to be found in the tree. The first step is to ensure that the traffic threshold is not exceeded. This is done by adding the traffic from the root node to *node1* to that already calculated for subtree *sub2*. If the traffic threshold is not exceeded then there is another condition that needs to be met. This condition ensures that *node1* and the subtree that the link is being added to do not both contain the root node. If this were true then a cycle would be added to the graph and the graph would no longer represent a tree. If it is not true, then *node1* and *node2* are added to row *sub2* of the matrix *subtree* and all the necessary updates are made to *substat*. This is all done in the function *addsub*. Finally the variable *links* is updated and the loop begins again.

The third combination is when *sub2* is zero, i.e. *node2* cannot be found in the tree or *node2* is the root node, and *sub1* is nonzero, i.e. *node1* is already to be found in the tree. This is just the opposite to the situation described above and all checks and calculations are the same as described above except that now *node2* is the node that is not in the tree and *node1* is in the tree.

The final situation is if both *sub1* and *sub2* are nonzero and they are not equal. If they were equal then adding the link consisting of *node1* and *node2* to the tree would form a cycle. The first thing to do is to check that the traffic threshold is not exceeded. This is done by adding the traffic value stored for *sub1* to the traffic value stored for *sub2*. If this value is less than or equal to the threshold then the two subgraphs, *sub1* and *sub2* are joined together and all the necessary updates are made to both of *subtree* and *substat*. All of this is done in the

procedure *joinsub*. Finally the variable *links* is incremented and the loop begins again.

Once the loop has ended the matrices *substat* and *subtree* are passed to a function *findtree* that extracts all the links from all the different rows in *subtree* and combines them onto one array, *tree*, which is the required output of this function.

F.1.3 CHMAT.M

This function converts the cost matrix to a matrix which contains a list of all the elements of the cost matrix in ascending order alongside their row and column positions. These row and column positions are the nodes that the link is incident on. The inputs to this function are *cst*, the cost matrix, and n, the number of nodes in the network. The output is the ordered matrix *pop*.

The first loop puts all the elements in the cost matrix into another matrix *pop*. The first column in *pop* contains the element of the matrix and the second and third columns contain the row and column number of *cst* that this element was found in.

Next the matrix *pop* is placed in ascending order. With the link weights being the column that is sorted.

F.1.4 ELEMENT.M

This function returns the number representing the subtree in which a particular node can be found. The inputs to this function are the matrix *subtree* and *node*. *node* is the variable to look for in *subtree*. The output is *sub* a number representing the row in *subtree* in which the variable *node* can be found.

The first check that is made is to see whether *node* is the root node, node 1. If it is then *sub* is set equal to zero. If not then *subtree* is searched until *node* is found. If it is found then *sub* is set equal to the number of the row in which it is found.

If it is not found then *sub* is set equal to zero.

F.1.5 ADDTREE.M

This function adds a link to a new row in the matrix *subtree*. It also updates the matrix *substat*. The inputs to this function are *subtree*, *substat*, *node1* and *node2* and the sum of the traffic from the root node to these two nodes, the variable *weight*. The outputs of this function are the modified matrices *subtree* and *substat*.

First the size of the matrix *subtree* is calculated. m is the number of rows and n the number of columns. Then the link consisting of *node1* and *node2* is added to the next empty row in the matrix. This is row $m+1$.

The matrix *substat* is also updated. If either of *node1* or *node2* are the root node then the second column of row $m+1$ in *substat* is set equal to 1. The first column of *substat* is set equal to 3, the next free position in the matrix, and the third column in *substat* is set equal to the variable *weight*.

F.1.6 ADDSUB.M

This function adds a link to a row in the matrix *subtree* and updates *substat* accordingly. The inputs to this function are *subtree*, *substat*, *sub*, *node1*, *node2* and *weight*. *sub* is the number of the row in *subtree* that the link comprising *node1* and *node2* must be added to. *weight* is the total traffic flowing in this new subtree. The outputs of this function are the modified matrices *substat* and *subtree*.

The variable *pointer* contains the next empty column number in row *sub* of *subtree*. The link comprising *node1* and *node2* is added to *subtree* in row *sub* at positions *pointer* and *pointer+1* respectively. *substat* is updated so that the value in column one is the next free position in row *sub* of *subtree*. If either *node1* or *node2* are the root node then column two of *substat* is updated accordingly. Otherwise it is left as is.

Next the third column of *substat* is updated so that it contains the variable *weight*, the combined value of the traffic flow in the new subtree.

F.1.7 JOINSUB.M

This function combines two separate subtrees into one. The inputs to this function are *subtree*, *substat*, *sub1*, *sub2*, *node1* ,*node2* and *weight*. The variables *sub1* and *sub2* are indicators of the subtrees that *node1* and *node2* belong to. The *weight* is the amount of traffic flowing in the new combined subtree.

First this procedure initialises two variables, n and *add*. n is the pointer to the next free open space in row *sub1* of *subtree*. *add* is the pointer to the next free open space in row *sub2* of *subtree*. This is used in calculating how many non-zero elements there are in row *sub2*.

The next line assigns the row *sub2*, without any of its padding zeroes, as well as *node1* and *node2* to the end of row *sub1* of *subtree*. The next few lines update *substat*. It first updates the next free position in row *sub1*. Then if the subtree contained in row *sub2* contained the root node this is reflected in the root pointer of *sub1*. Otherwise it is left as is. The amount of traffic flowing in this new subtree, *weight*, is then assigned to the correct position in row *sub1* of *substat*.

Finally all information relating to *sub2* in both *subtree* and *substat* is set equal to zero.

F.1.8 FINDTREE.M

This function combines all the rows of the matrix *subtree* into one array, extracting all the zeroes from it. The inputs to this function are *subtree* and *substat* and the output is the constrained minimum spanning tree, *tree*.

The first step is to find the size of the matrix *subtree*. m is the number of rows it has and n is the number of columns. The row counter is also initialised to 1.

Then the loop begins. The loop will continue until it meets its termination condition which is when the row counter exceeds the number of rows in *subtree*.

The first element of the current row of *subtree* is checked to make sure that it is not zero. If it is it means there is no information contained in that row and it is discarded. If it is not then the entire row, excluding the zeroes that may appear on the end of it are added to the

variable *tree*. This process continues until all the rows have been searched.

F.2 PBIL

This section explains the operation of the PBIL algorithm as well as how the coding scheme and fitness function were implemented in Matlab. Figure F-3 is a diagram showing all the functions involved in the generation of this algorithm. These functions are explained in the order they appear in the tree, from top to bottom and from left to right. There are however two functions, *contain* and *search*, that will not be mentioned in the following description due to their simplicity.

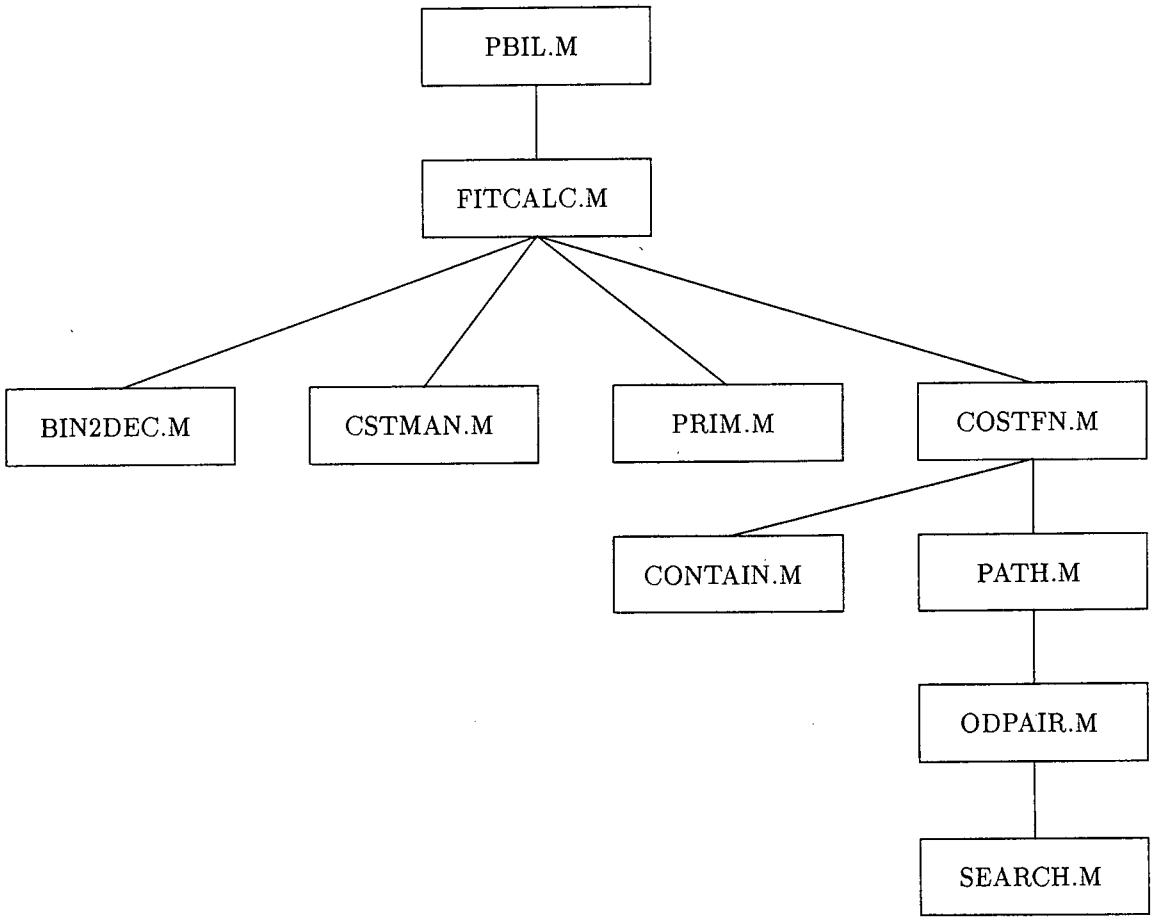


Figure F-3: The Functions used in the Implementation of PBIL

F.2.1 PBIL.M

This is the function which implements the PBIL algorithm. The inputs to the function are *dimen*, *thresh*, *cst* and *traf*. *dimen* is the number of nodes the network has and *thresh* is the traffic threshold, the maximum amount of traffic that is allowed to flow in any subtree of the network. *cst* is the cost matrix and *traf* is the traffic matrix. The outputs of this function are *tree* and *besteverF*. *besteverF* is the best solution obtained after completion of the algorithm, i.e. it is the minimum cost of the network found by PBIL. *tree* is the network topology stored in the representation that is used to calculate its fitness.

The first step in this function is to initialise all the variables. All the variables that are initialised here are not going to be mentioned, but the most important ones are PV, the probability vector, this is initialised to contain values of 0.5, and $MAXGEN$ and $POPSIZE$. These two variables determine the number of generations that will be completed before the algorithm ends and the population size of each of these generations.

Next the main loop begins, as mentioned previously this loop will terminate when the number of generations completed is equal to $MAXGEN$. The generation counter, *gen*, is then incremented.

In the following loop the population of the current generation is generated by sampling the probability vector. All the individuals generated are stored in the matrix B. Once an individual has been generated it is evaluated by passing it onto the function *fitcalc*. The fitness values of each individual are stored in the matrix F. Once this loop is completed the fitness matrix is multiplied by a variable *mark*. If *mark* is set to -1, as it is in this case, it means that PBIL's task is to minimise the current problem.

Next the fittest member of the population is found. The individual is stored in the variable *bestB* and its fitness is stored in the variable *bestF*. The worst member of the population is then found. The individual is stored in *worstB* and its fitness in *worstF*. These values are used in the updating of the probability vector.

Next a check is made to see whether the best individual generated in the current generation is better than the best individual ever generated. If it is then the appropriate variables are

updated.

Next the probability vector is updated towards the best vector. Then it is updated away from the worst vector and then it is mutated.

Before the loop starts again the statistics of the generation are displayed on the screen. These are the generation number, *gen*, the fitness of the best solution ever found, *besteverF*, and the fitness of the best solution found in the current generation, *bestF*.

After the loop has ended the best solution ever found, *besteverB*, is converted to the tree representation.

F.2.2 FITCALC.M

This function calculates the fitness of a vector generated by PBIL. The inputs to this function are *dimen*, *bias*, *cst*, *traf* and *thresh*. *dimen* is the number of nodes in the network currently being optimised. *bias* is a vector containing the binary representation of the bias chromosome. *cst* and *traf* are the cost and traffic matrices and *thresh* is the traffic threshold. The output of this function is the variable *fit* which is a number representing the fitness of *bias*.

First *bias* is converted from a binary to an integer representation. This is done in the function *bin2dec*. The integer value of the chromosome is stored in *realb*. This bias together with certain other variables are used to modify the cost matrix. This is done in the function *cstman* and the modified cost matrix is stored in *newcst*. Then Prim's algorithm is used to find the minimum spanning tree of the modified cost matrix. The function used to perform this is called *prim* and the resulting MST is stored in *tree*. Finally the fitness of the individual is calculated by the function *costfn*. This function also checks to see whether any of the constraints have been violated.

F.2.3 BIN2DEC.M

This function will take a vector of bits, divide them up into groups of eight and then convert each of these groups into their integer representation. The input is the array *bin*, which is the vector of bits to be converted. The output is the array *dec*, which is the integer version

of *bin*.

The first thing to do is to determine the size of the input. Next, this value is divided by eight in order to determine the number of integers that will appear in the output array. *lbeg* and *lend* are pointers to the beginning and end of each group of eight bits.

Then a loop begins. The loop will terminate when the number of elements inserted into the array *dec* is equal to the size the array is supposed to be, *szdec*. The next loop converts the current group of eight bits to its standard binary equivalent.

Then the counter is updated, the integer value is placed in the correct position in *dec* and the pointers to the beginning and end of each group of eight bits are updated.

F.2.4 CSTMAN.M

This function modifies the cost matrix according to the following equation.

$$newcst_{ij} = cst_{ij} + p * cst_{max} * (b_i + b_j)$$

Where $newcst_{ij}$ are the elements of the modified cost matrix, cst_{ij} are the elements of the original cost matrix, p is the multiplication factor, cst_{max} is the maximum value of the cost matrix that is not infinity and b_i and b_j are the biases relating to the nodes i and j respectively.

This is used in the conversion of the bias chromosome to a form which can be easily used to calculate its fitness. Section 3.3.4 gives a detailed explanation of this coding scheme. The inputs to this function are *bias*, *cst*, *p* and *n*. *bias* is the integer version of the bias chromosome. *cst* is the cost matrix. *p* is a multiplication factor that is used in the conversion and is set equal to 1 and *n* is the number of nodes in the network. The output of the function is *newcst*, the modified cost matrix.

First the maximum value in the cost matrix is found. There is a simple command in Matlab that can find the maximum value of a matrix. The loop is used because when there is no possibility of connecting two nodes together the element representing those two nodes in the cost matrix is infinity. So if the Matlab commands were used it would return the value infinity.

This would of course make the conversion by the above mentioned formula meaningless.

Once the maximum value has been found, each element is then modified using the above mentioned formula. This modified value is stored in *newcst*.

F.2.5 PRIM.M

This function is an implementation of Prim's MST algorithm. In order to fully understand how it works, it is recommended that Appendix D is read in conjunction with this section. The inputs to this function are the cost matrix, *cst* and the number of nodes in the network, *n*. The output is the minimum spanning tree, *t*.

First some variables are initialised. *u* is the list of all nodes that are already connected to the tree, this is initialised to 1 since this is the root node and will be the starting point of the algorithm. *t* is a list of all links already connected to the tree, it is initially blank.

The loop begins and will repeat itself *n - 1* times. The first thing to do in the loop is to find the minimum cost in each row of the network. The smallest of all of these values is the cost of the link that will be connected to the subtree.

Once the next link in the tree has been found, *u* and *t* are updated. The next loop ensures that only the nodes that are not already connected to the subtree are eligible for inclusion in the next iteration of the loop. It does this by setting the costs of all the combinations of links between all of the nodes already connected to the network to infinity. In this way when the next iteration of the loop begins and the minimum value of each row is selected, any of the possible links that would create a cycle would never be selected.

F.2.6 COSTFN.M

This function performs two tasks. Firstly it determines whether a network violates the traffic constraint. If it doesn't it calculates the network's fitness, if it does it assigns the network a fitness of infinity. The inputs to this function are *tree*, the tree, *cst*, the cost matrix, *traf*, the traffic matrix, *nodes*, the number of nodes in the network and *threshold*, the traffic threshold.

The first loop in this function finds all the nodes in the tree that are directly connected to the root node. These nodes represent the beginning of each subtree. All these nodes are stored in the array *val*. This is used in determining whether the network violates the traffic constraint.

Next a few variables are initialised. The variable *cval* contains the number of elements in *val*. Then the main loop begins, this loop repeats for all the values in *val*. It will terminate if flag is set equal to 1 or if all the elements in *val* have been checked.

listn is then initialised. *listn* is an array that stores all the nodes in the subtree whose traffic values have already been included in the calculation to determine whether the constraint has been violated. This ensures that a node that is not a leaf node will not have its traffic value included in the calculation more than once. *tcheck* is a cumulative sum of all the traffic flowing in a subtree. It is initialised to be the traffic flowing from the root node to the current node in *val* that is being dealt with. Then another loop begins. This one will terminate if either *flag* is set equal to 1 or if the variable n is greater than the number of nodes in the network. The purpose of this loop is to find the total amount of traffic flowing in the current subtree. It does this by finding the route between the current node in *val* and all other nodes in the subtree. This is done using the function *path*. If the current node n is not in the subtree, the function *path* will return a value of infinity. Once the path has been found it is used to calculate the traffic in the current subtree. This is where *listn* is used, if a node is part of a path to another node and its traffic value has already been added to *tcheck* it will appear in *listn*. The function *contain* checks to see whether a particular node is in fact part of *listn*. If it is the node's traffic contribution is ignored.

If the subtree does not violate the constraint the main loop is repeated this time looking at the next value in *val*, the beginning of the next subtree. If it does violate the traffic constraint then both loops are terminated.

If after the loops have ended the traffic constraint has not been violated then the cost of the network is calculated. If the traffic constraint has been violated the cost of the network is set equal to infinity.

F.2.7 PATH.M

This function determines the path between any two nodes contained in a subtree. The inputs to this function are *tree*, *row* and *col*. *tree* is the tree from which the route needs to be found. *row* and *col* are the start and end nodes of the required route. The output *route* is an array of links comprising the route from node *row* to node *col*.

The tree and the two nodes are first passed to a function *odpair* that attempts to find the route from *row* to *col*. If it can't find this route the flag *inc* is set equal to 1. If this is the case then *odpair* is again used, however this time it is to find the route between *col* and *row*.

If the route between the two nodes still cannot be found it means that they are in different subtrees and if this is the case the route does not need to be found. The variable *route* is then set equal to infinity.

F.2.8 ODPAIR.M

This functions returns the route between two of the input variables o and d. o is the origin and d the destination. The other input variable is *tree*. The outputs are *route* and *inc*. *route* is the route that has been calculated between the two variables, o and d. *inc* is set to 1 if this route cannot be found.

First all the variables are initialised. The variable *no* is set equal to d, the destination. *flag* is set equal to 0.

Next the loop begins, it will continue until flag is set equal to 1. The conditions under which this will occur will be specified later. The first step in the loop is to find the position in the tree of the variable *no*. This is done using the procedure *search*. *search* looks for *no* in the even positions of the tree. This value is stored in *pos*. The two positions in *route* are then defined to be the link containing *no*. *no* is then set equal to the other node in the link.

If *no* is now equal to the root node or to the origin variable, o, then *flag* is set equal to 1 and the loop will terminate.

If the loop ends and the route was found from the destination, d, to the root node the output

variable *inc* is set equal to 1. This will tell the function *path* that the function *odpair* was unable to find the route.

Appendix G

Program Listings

This chapter contains a listing of all the Matlab programs used to implement the algorithms described in Appendix F. As in Appendix F, this chapter is divided into two sections, the first deals with the CMST algorithm and the second with PBIL.

G.1 The CMST Algorithm

This section contains a listing of all the code used in implementing the CMST algorithm. The order in which the program listings occur is the same as the order of the explanations given in Appendix F.

G.1.1 CMST.M

```
function tree = cmst(cst,traf,n,thresh)

% This function computes the constrained Minimum Spanning Tree of the cost
% matrix

% change format of cost matrix
pop = chmat(cst,n);
```

```
% initialise variables
pointer = 2;
links = 1;
subtree(1,1:2) = [pop(1,2) pop(1,3)];
substat(1,1) = 3;
substat(1,3) = traf(1,pop(1,2)) + traf(1,pop(1,3));
if (pop(1,2)==1) | (pop(1,3)==1)
   substat(1,2) = 1;
else
   substat(1,2) = 0;
end

while (links < n - 1)

% extract the nodes comprising the next link from pop
   node1 = pop(pointer,2);
   node2 = pop(pointer,3);

% determine which subtrees the nodes belong to
   sub1 = element(subtree,node1);
   sub2 = element(subtree,node2);

   if (sub1==0)&(sub2==0)

% if neither of the nodes are to be found in the tree
      weight = traf(1,node1) + traf(1,node2);
      if weight <= thresh
         [subtree,substat]=addtree(subtree,substat,node1,node2,weight);
         links = links + 1;
      end

   elseif (sub1==0)&(sub2~=0)

% if node2 is in the tree but node 1 not
      weight = substat(sub2,3) + traf(1,node1);
      if weight <= thresh
```

```
         if (node1~=1)|((node1==1)&(substat(sub2,2)==0))
            [subtree,substat]=addsub(subtree,substat,sub2,node1,node2,weight);
            links = links + 1;
         end
      end

   elseif (sub2==0)&(sub1~=0)

% if node 1 is in the tree but node 2 not
      weight = substat(sub1,3) + traf(1,node2);
      if weight <= thresh
         if (node2~=1)|((node2==1)&(substat(sub1,2)==0))
            [subtree,substat]=addsub(subtree,substat,sub1,node1,node2,weight);
            links = links + 1;
         end
      end

   elseif (sub2~=0)&(sub1~=0)&(sub1~=sub2)

% if both nodes are in the tree and they aren't in the same subtree
      weight = substat(sub1,3) + substat(sub2,3);
      if weight <= thresh
         if ~((substat(sub1,2)==1)&(substat(sub2,2)==1))
            [subtree,substat]=joinsub(subtree,substat,sub1,sub2,node1,node2,weight);
            links = links + 1;
         end
      end

   end

   pointer = pointer + 1;
end

tree = findtree(subtree,substat);
```

G.1.2 CHMAT.M

```
function pop = chmat(cst,n)

% this function changes the cost matrix into an ordered table of values

count = 1;
pop = [];

% convert layout of cost matrix
for row = 1:n - 1
   for col = row + 1:n
      pop(count,1) = cst(row,col);
      pop(count,2) = row;
      pop(count,3) = col;
      count = count + 1;
   end
end

% sort new cost matrix in ascending order
[order index] = sort(pop(:,1));
pop = pop(index,:);
```

G.1.3 ELEMENT.M

```
function sub = element(subtree,node)

% This function finds what row of subtree node can be found in, if node is 1
% or cannot be found in subtree at all then sub equals 0

[m n] = size(subtree);
flag = 0;

if node==1
   sub=0;
```

```
else
   for row = 1:m
      for col = 1:n
         if (subtree(row,col)==node)
            sub = row;
            flag = 1;
         end
      end
   end
end

if flag==0
   sub = 0;
end
```

G.1.4 ADDTREE.M

```
function [subtree,substat] = addtree(subtree,substat,node1,node2,weight)

% This function adds the link comprising of node 1 and node2 to a new row in
% matrix substat, thus forming a new subtree

% add link to new row in subtree
[m n] = size(subtree);
subtree(m+1,1:2) = [node1 node2];

% update all the appropriate values in substat
if (node1==1)|(node2==1)
   substat(m+1,2) = 1;
end
substat(m+1,1)=3;
substat(m+1,3)=weight;
```

G.1.5 ADDSUB.M

```
function [subtree,substat]=addsub(subtree,substat,sub,node1,node2,weight)

% This function adds the link comprising node1 and node2 to row number sub in
% subtree

% add the link to the end of the row
pointer = substat(sub,1);
subtree(sub,pointer:pointer + 1) = [node1 node2];

% update all necessary statistics related to row sub
substat(sub,1) = substat(sub,1) + 2;
if (node1==1) | (node2==1)
   substat(sub,2) = 1;
end
substat(sub,3) = weight;
```

G.1.6 JOINSUB.M

```
function [subtree,substat]=joinsub(subtree,substat,sub1,sub2,node1,node2,weight)

% This function joins two seperate subtrees into one

% add two subtrees together to form new sub1
n = substat(sub1,1);
add = substat(sub2,1);
subtree(sub1,n:n+add) = [subtree(sub2,1:add-1) node1 node2];

% update relevant statistics
substat(sub1,1) = n + add + 1;
if (substat(sub2,2)==1)
   substat(sub1,2) = 1;
end
substat(sub1,3) = weight;
```

```
% delete everthing relating to sub2
subtree(sub2,1:add-1) = ones(1,add-1) - 1;
substat(sub2,1:3) = [0 0 0];
```

G.1.7 FINDTREE.M

```
function tree = findtree(subtree,substat)

% This function finds the tree by joining different rows in the matrix subtree

[m n] = size(subtree);
tree = [];
row = 1;

while (row<=m)
   if (subtree(row,1)~=0)
      tree = [tree subtree(row,1:substat(row,1)-1)];
   end
   row = row + 1;
end
```

G.2 PBIL

This section contains a listing of the Matlab code used in implementing the PBIL algorithm, the coding scheme and the fitness function. As in the previous section the order of the programs is the same as that appearing in Appendix F, however two functions, *contain.m* and *search.m*, that were not included in Appendix F due to their simplicity will be featured here.

G.2.1 PBIL.M

```
function [tree,besteverF] = pbil(dimen,thresh,cst,traf)
```

```
L = .1;          % Learning rate
NLR = 0.075;     % Negative Learning Rate
MPROB = 0.02;    % Probability of Mutation
MSHIFT = 0.05;   % Shift of Mutation in random direction
bits = 8;        % No of bits used to represent each variable
mark = -1;       % mark = 1 to maximize, -1 to minimize
MAXGEN = 150;    % termination condition of loop
POPSIZE = 100;   % population size
n = bits*dimen; % dimension of probability vector

% initialisation of probability vector
PV = .5*ones(1,n);
besteverF = -inf;
besteverB = ones(1,n);
gen = 0;

while (gen < MAXGEN)
   gen = gen + 1;

% Generate and evaluate entire population
   for pop = 1:POPSIZE
      B(pop,1:n) = rand(1,n) < PV;
      F(pop) = fitcalc(dimen,B(pop,:),cst,traf,thresh);
   end

    F = mark*F;

% Find the best member of the population
   [bestF, pos] = max(F);
   bestB = B(pos,:);

% Find the worst member of the population
   [worstF, posit] = min(F);
   worstB = B(posit,:);
```

```
% Update best solution ever found
   if bestF > besteverF
      besteverF = bestF;
      besteverB = bestB;
   end

% update probability vector towards best vector
   PV = (1 - L)*PV + L*bestB;

% update probability vector away from worst vector
   for lp=1:n
      if bestB(1,lp)~=worstB(1,lp)
         PV(1,lp) = PV(1,lp)*(1 - NLR) + bestB(1,lp)*NLR;
      end
   end

% Perform mutation
   for lp=1:n
      pmut = rand(1);
      if pmut < MPROB
         random = rand(1)<0.5;
         PV(1,lp) = PV(1,lp)*(1 - MSHIFT) + random*MSHIFT;
      end
   end

% Display statistics of generation
   gen
   besteverF
   bestF
end

% convert from binary bias chromosome to tree
realb = bin2dec(besteverB);
newcst = cstman(realb,cst,1,dimen);
tree = prim(newcst,dimen);
```

G.2.2 FITCALC.M

```
function fit = fitcalc(dimen,bias,cst,traf,thresh)

% This function calculates the fitness of one vector generated in PBIL

% convert bias chromosome from binary to integer representation
realb = bin2dec(bias);

% modify cost matrix
newcst = cstman(realb,cst,1,nodes);

% Apply Prim's Algorithm to the modified matrix
tree = prim(newcst,nodes);

% Check if the tree violates any constraints, if it does not, calculate it's
% cost
fit = costfn(tree,cst,traf,nodes,thresh);
```

G.2.3 BIN2DEC.M

```
function dec = bin2dec(bin)

% This function divides a binary chromosome into groups of 8 bits and then
% converts the groups to decimal no's

[ignore szbin] = size(bin);
szdec = szbin/8;      % no of decimal values in the output
count = 0;
temp = 0;
lbeg = 1 + count*8;  % beginning of the 1st group of 8 bits
lend = 8 + count*8;  % end of the 1st group of 8 bits

while (count < szdec)
```

```
% convert each group of 8 bits to it's decimal equivalent
   for loop = lbeg:lend
     power = lend - loop;
     temp = temp + 2^power*bin(loop);
   end

% update counters and output variables
   count = count + 1;
   dec(1,count) = temp;
   temp = 0;

% update pointers to the beginning and end of each group
   lbeg = 1 + count*8;
   lend = 8 + count*8;

end
```

G.2.4 CSTMAN.M

```
function newcst = cstman(bias,cst,p,n)

% This function computes a new cost matrix based on the biases in the
% chromosome as well as the multiplication factor p

% compute cmax the maximum cost in the matrix
cmax = 0;

for row = 1:n
   for col = 1:n
      if cst(row,col)~=inf
         if cst(row,col) > cmax
            cmax = cst(row,col);
         end
      end
   end
```

```
end


% compute new biased matrix
for row1 = 1:n
  for col1 = 1:n
     newcst(row1,col1) = cst(row1,col1) + p*cmax*(bias(row1) + bias(col1));
  end
end
```

G.2.5 PRIM.M

```
function t = prim(cst,n)

% Prims Algorithm to find MST from cost matrix

% initialise variables
u = [1];
t = [];
[ignore szu] = size(u);

% beginning of main loop
for main = 1:n-1
   val = inf;

% Finds minimum link cost in each row
   [y,i] = min(cst);
   minmat = [y' i'];

% Finds smallest link to connect to subtree
   for loop = 1:szu
      if minmat(u(1,loop),1) < val
         val = minmat(u(1,loop),1);
         pos = minmat(u(1,loop),2);
```

```
        row = u(1,loop);
      end
   end

   u = [u pos];
   u = sort(u);
   t = [t row pos];
   [ignore szu] = size(u);

% Ensures that there are no cylces in the tree
   for loop = 2:szu
      for loop1 = 1:loop
         cst(u(1,loop),u(1,loop1)) = Inf;
         cst(u(1,loop1),u(1,loop)) = Inf;
      end
   end

end             % end of main loop
```

G.2.6 COSTFN.M

```
function fit = costfn(tree,cst,traf,nodes,threshold)

% This function determines whether tree violates any of the traffic
% constraints, if it doesn't then the cost of the network is calculated

count = 1;
val = [];
[row col] = size(tree);

% find all the nodes that are directly connected to the root node, these are
% the heads of each subtree
for loop = 1:2:col-1
   if tree(1,loop)==1
      val(1,count) = tree(1,loop+1);
```

```
      count = count + 1;
   end
end

flag = 0;
n = 2;
fit = 0;
pos = 1;
[rval cval] = size(val);

while (flag==0)&(pos<=cval)

% repeat for each of the values in val
   n = 2;
% listn maintains a list of all the nodes in the subtree for which the traffic
% has already been accounted
   listn = [];
   tcheck = traf(1,val(1,pos));

   while (flag==0)&(n<=nodes)

% this loop finds the total cost of each of the subtrees
      route = path(tree,val(1,pos),n);
      if route~=[inf]
         [rowr colr] = size(route);

         for loop = 1:2:colr-1

% check whether the traffic of the current node has already been accounted for
            check = contain(route(1,loop),listn);
            if check==0
               tcheck = tcheck + traf(1,route(1,loop));
               listn = [listn route(1,loop)];
            end
         end
```

```
         if tcheck > threshold
            flag = 1;
         end

      end
      n=n+1;
   end

   pos = pos+1;
end

if (flag==0)
   for loop = 1:2:col-1
      fit = fit + cst(tree(1,loop),tree(1,loop+1));
   end
else
   fit = inf;
end
```

G.2.7 CONTAIN.M

```
function check = contain(no,listn)

check = 0;
[row col] = size(listn);

for loop = 1:col
   if listn(1,loop) == no
      check = 1;
   end
end
```

G.2.8 PATH.M

```
function route = path(tree,row,col)
```

```
% This function finds the path between any two nodes in the network.

% initial variable decleration
inc = 0;
route = [];

% Find the route from node row to node col
[route,inc] = odpair(tree,row,col);

% If this route cannot be found then find the route from col to row
if inc==1
   inc = 0;
   route = [];
   [route,inc] = odpair(tree,col,row);
end

% If the route still can't be found the nodes are in different subtrees so set
% route to infinity to reflect this
if inc==1
   route = [inf];
end
```

G.2.9 ODPAIR.M

```
function [route,inc] = odpair(tree,o,d)

% This function returns the route from o to d

% initialise variables
no = d;
inc = 0;
count = 1;
route = [];
pos = 0;
```

```
flag = 0;

while (flag==0)

% find the link in tree that has no in an even position
   pos = search(tree,no);
   route(1,count) = tree(1,pos);
   route(1,count + 1) = tree(1,pos - 1);
   no = tree(1,pos - 1);
   count = count + 2;

% if the next number to be found in o or 1, terminate the loop
   if (no==o) | (no==1)
      flag = 1;
   end
end

% if the route between o and d has not been found set inc = 1
if (flag==1) & (no~=o)
   inc = 1;
end
```

G.2.10 SEARCH.M

```
function pos = search(tree,no)

% This function searches for no in even positions of tree

count = 2;

while tree(1,count)~=no
    count = count + 2;
end

pos = count;
```

Bibliography

[1] M. Minoux. *Network Synthesis and Optimum Network Design Problems: Models, Solution Methods and Applications.* Network, Vol 19(1989), pp 313-360

[2] Robert R. Boorstyn and Howard Frank. *Large-Scale Network Topological Optimisation.* IEEE Transactions on Communications, Vol COM-25, No 1, pp 29-47, January 1977

[3] Rong-Hong Jan, Fung-Jen Hwang and Sheng-Tzong Cheng. *Topological Optimisation of a Communication Network Subject to a Reliability Constraint.* IEEE Transactions on Reliability, Vol 42, No 1,pp 63-70, March 1993

[4] Inder M. Soi and K.K. Aggarwal. *Reliability Indices for Topological Design of Computer Communication Networks.* IEEE Transactions on Reliability, Vol R-30, No 5, pp 438-443, December 1981

[5] B. Gavish. *Augmented Lagrangian Base Algorithms for Centralized Network Design.* IEEE Transactions on Communications, Vol 33, pp 1247-1257, 1985

[6] Aaron Kershenbaum. *Telecommunications Network Design Algorithms.* McGraw Hill, 1993

[7] Andrew S. Tanenbaum. *Computer Networks.* Prentice-Hall 1981

[8] Charles Palmer and Aaron Kershenbaum. *Representing Trees in Genetic Algorithms.* Available at ftp://ftp.uct.ac.za/pub/mirrors/EC/GA/papers/trees94.ps.gz

[9] L. Berry, B. Murtagh, G. McMahon and S. Sudgen. *Applications of a Genetic-Based Algorithm for Optimal Design of Tree-Structured Communications Networks.* Proceedings of the Regional Teletraffic Engineering Conference of the International Teletraffic Congress, pp. 361-370, South Africa, September 1995

[10] L. Berry, B. Murtagh, G. McMahon and S. Sudgen. *Optimisation Models for Communication Network Design.* As yet unpublished

[11] John H. Holland. *Adaption in Natural and Artificial Systems.* MIT Press, 1975

[12] Stephen W. Hawking. *A Brief History of Time - From the Big Bang to Black Holes.* Bantam Press, 1988

[13] Joseph L. Hammond and Peter J.P. O'Reilly. *Performance Analysis of Local Computer Networks.* Addison-Wesley Publishing Company Inc, 1988

[14] William Stallings. *Local Networks : An Introduction.* MacMillan Publishing Company, 1984

[15] J.E. Flood, Editor. *Telecommunications Networks.* Peter Pregrinus Ltd., 1977

[16] K. F. Man, K. S. Tang and S. Kwong. *Genetic Algorithms : Concepts and Applications.* IEEE Transactions on Industrial Electronics, Vol 53 No 5, pp. 519-533

[17] David Beasley, David R. Bull and Ralph R. Martin.
An Overview of Genetic Algorithms : Part 1, Fundamentals. Available at ftp://ftp.uct.ac.za/pub/mirrors/EC/GA/papers/over93.ps.gz

[18] Lawrence Davis, Editor. *Handbook of Genetic Algorithms.* International Thomson Computer Press, 1996

[19] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Publishing Company, 1989

[20] Melanie Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, 1996

[21] Alden H. Wright. *Genetic Algorithms for Real Parameter Optimization.* Foundations of Genetic Algorithms, Edited by J.E. Rawlings, pp205-218, Morgan Kaufmann, 1991

[22] Larry J. Eshelman and J. David Schaffer. *Real-Coded Algorithms and Interval-Schemata.* Foundations of Genetic Algorithms-2, Edited by L. Darrell Whitley, pp 187-202, Morgan Kaufmann, 1993

[23] Larry J. Eshelman, Keith E. Mathias and J. David Schaffer. *Convergence Controlled Variation.* Foundations of Genetic Algorithms-4, Edited by K. Belew and M. Vos, pp 18-33, Morgan Kaufmann, 1996

[24] Larry J. Eshelman. *The CHC Adaptive Search Algorithm : How to Have Safe Search When Engaging in Nontraditional Genetic Recombination.* Foundations of Genetic Algorithms, Edited by J.E. Rawlings, pp265-283, Morgan Kaufmann, 1991

[25] C.Z. Janikow and Z. Michalewicz. *An Experimental Comparison of Binary and Floating Point Representation in Genetic Algorithms.* Proceedings of the Fourth International Conference on Genetic Algorithms, pp 31-36, Morgan-Kaufmann, 1991

[26] Shumeet Baluja. *Population-Based Incremental Learning : A Method for Integrating Genetic Search Based Function Optimization and Competitive Learning.* Available at http://www.dai.ed.ac.uk/groups/evalg/eag_local_copies_of_papers.body.html

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

[28] Nora Hartsfield and Gerhard Ringel. *Pearls in Graph Theory : A Comprehensive Introduction.* Academic Press, 1990

[29] John Hertz, Anders Krogh and Richard G. Palmer. *Introduction to the Theory of Neural Computation.* Addison-Wesley, 1991