ROLAND: A TOOL FOR THE REALISTIC OPTIMISATION OF LOCAL ACCESS NETWORK DESIGN

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE,
FACULTY OF SCIENCE
AT THE UNIVERSITY OF CAPE TOWN
IN FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Simon Buffler
February 1999

Supervised by
Prof P.S. Kritzinger

Abstract

Investment in the local access network represents between 50% and 70% of capital investment of a telecommunications company. This thesis investigates algorithms that can be used to design economical access networks and presents ROLAND: a tool that incorporates several of these algorithms into an interactive environment. The software allows a network designer to explore different approaches to solving the problem, before adopting a particular one.

The family of problems that are tackled by the algorithms included in ROLAND involve determining the most economical way of installing concentrators in an access network and connecting demand nodes such as distribution points to these concentrators. The Centre-of-Mass (COM) Algorithm identifies clusters of demand in the network and suggests good locations for concentrators to be installed.

The problem of determining which concentrators in a set of potential sites to install is known as the *concentrator location problem* (CPL) and is an instance of the classical capacitated plant location problem. Linear programming techniques such as branch-and-bound can be used to find an optimal solution to this problem, but soon becomes infeasible as the network size increases. Some form of heuristic approach is needed, and ROLAND includes two such heuristics, namely the Add and Drop Heuristic.

Determining the layout of multidrop lines, which allow a number of demand nodes to share the same connection to a concentrator, is analogous to finding minimal spanning trees in a graph. Greedy approaches such as Kruskal's algorithm are not ideal however, and heuristics such as Esau-William's algorithm achieve better results. Kruskal's algorithm and Kershenbaum's Unified Algorithm (which encapsulates a number of heuristics) have been implemented and come bundled with ROLAND. ROLAND also includes an optimal terminal assignment algorithm for associating distribution points to concentrators.

A description of ROLAND's architecture and GUI are provided. The graphical elements are kept separate from the algorithm implementations, and an interface class provides common data structures and routines for use by new algorithm implementations. A test data generator, able to create random or localized data, is also included.

A new hybrid concentrator location algorithm, known as the *Cluster-Add* Heuristic is presented. The implementation of this algorithm is included in ROLAND, and demonstrates the ease with which new solution methods can be integrated into the tool's framework. Experimentation with the concentrator location algorithms is conducted to show the Cluster-Add Heuristic's relative performance.

Acknowledgements

For their involvement throughout the duration of this dissertation, I would like to thank:

- My family (in particular Andy for reading the final document) and Lara Strong for their love and support.
- Two generations of the DNA Lab (Andrew Hutchison, Heinz Kabutz, William Knottenbelt, Mark Mestern, Philip Wall, and Justin Templemore-Finlayson) for their inspiration and assistance at various stages of the project.
- Prof Kritzinger for his guidance, assistance, and patience throughout the past three years.
- Telkom and the Foundation for Research and Development for their financial support.
- My Creator for never giving up on me

Plans fail for lack of counsel, but with many advisers, they succeed
Proverbs 15:22

Contents

Abstract ii

Acknowledgements iv

1 Introduction 4

1.1 Motivation and Background 4

1.2 Project Context 4

1.3 The Local Access Network 6

1.3.1 Definition of the Access Network 6

1.3.2 Evolution of the Access Network 8

1.4 Dissertation Outline 10

2 Modelling and Optimizing Access Networks 12

2.1 Modelling a System 12

2.1.1 Network Models 15

2.2 Approaches to Designing Networks 17

2.2.1 The "Hands-on" Approach 17

2.2.2 Heuristics 17

2.2.3 Formal Optimization Techniques 18

2.3 Designing Access Networks 19

2.3.1 Feeder Network Expansion 19
2.3.2 Traditional Approach to Access Network Design 20
2.3.3 Access Network Models 21
2.4 Solution Methods included in ROLAND 22
2.4.1 Line Layout Problem 22
2.4.2 Terminal Assignment Problem 23
2.4.3 Concentrator Location Problem 23

3 ROLAND as an Adaptive Design Tool **25**
3.1 Network Design Tools 25
3.2 ROLAND's Architecture 28
3.2.1 The LAN Editor 29
3.2.2 The Algorithm Module 31
3.2.3 Test Data Generator and Form Module 32
3.3 Extending ROLAND 33
3.3.1 Finding the right path to the best solution: the need for extensibility 34
3.3.2 Adding a new algorithm 34
3.3.3 A case study 35

4 Linear Programming Fundamentals **37**
4.1 Linear Programming 37
4.1.1 Definitions 37
4.1.2 Integer Programming 38
4.2 Examples of Two Models 40
4.2.1 Concentrator Location Problem 40
4.2.2 Medium-Voltage Network Planning Problem 41

4.3 Solution of Integer Programming Problems 42
4.3.1 Branch-and-Bound Algorithm 43
4.4 Relaxation 47
4.4.1 Relaxation Algorithm for the Concentrator Location Problem . . . 50

5 Concentrator Location Algorithms **56**
5.1 The Centre-of-Mass (COM) Algorithm 56
5.1.1 Algorithm Description 56
5.1.2 Implementation 57
5.1.3 Examples 60
5.2 The Add Heuristic 63
5.2.1 Algorithm Description 63
5.2.2 Algorithm Complexity 65
5.3 The Drop Heuristic 65
5.3.1 Algorithm Description 65
5.3.2 Algorithm Complexity 66
5.4 The Cluster-Add Heuristic 66
5.4.1 Algorithm Overview 66
5.4.2 Motivation for Algorithm 67
5.4.3 Implementation 68
5.5 Comparison of Algorithms on Test Network 73

6 Line Layout and Terminal Assignment Algorithms **75**
6.1 Line Layout Algorithms 75
6.1.1 Kruskal's Constrained Minimum Spanning Tree Algorithm 75
6.1.2 The Esau-Williams Algorithm 77

6.1.3 Unified Algorithm 79
6.2 Terminal Assignment Algorithms 81
6.2.1 Alternating-Chain Algorithms 81
6.2.2 Extending the exchange-chain 82
6.2.3 The Assignment Problem 83
6.2.4 The Assignment Algorithm 85
6.2.5 Auxiliary Graphs 85
6.2.6 Dijkstra's Algorithm 87
6.2.7 Compressing the Auxiliary Graph 88
6.2.8 Algorithm Complexity 89

7 ROLAND's Graphical User Interface **90**

7.1 GUI Implementation Details 90
7.1.1 Adopting an Object Oriented Approach 90
7.1.2 Platform Independent Implementation using zApp 91
7.1.3 zApp's Features 91
7.1.4 Writing an Application in zApp 91
7.2 ROLAND's LAN Editor 92
7.2.1 High-level Overview 92
7.2.2 The NodeKeeper Class 93
7.2.3 CostMatrix Class 95
7.2.4 Relationship between the NodeKeeper and CostMatrix Classes . . . 98
7.3 Interfacing the Editor to the Algorithms 99
7.3.1 The Algorithm Class 101
7.3.2 The Algorithm Parameter Forms 101
7.4 Test Data Generator 102
7.4.1 Types of Data 102

8 Algorithm Performance **105**

8.1 Introduction 105

8.2 Linear Programming Code Experiment 106

8.3 Algorithm Execution Times 106

8.4 Cost of Solutions 107

8.5 Cluster-Add Algorithm Parameters 109

8.6 Conclusion 110

9 Conclusion **111**

9.1 Features of Project 111

9.2 Limitations and Future Work 112

A Concentrator Location Algorithms Pseudocode **114**

A.1 Centre-of-Mass Algorithm 114

A.2 Add Heuristic 118

A.3 Drop Algorithm 122

B Line Layout Algorithms Pseudocode **127**

B.1 Kruskal 127

B.2 Esau-Williams 130

B.3 Unified Algorithm 133

C Dynamic Concentrator Location Problem **135**

C.1 Introduction 135

C.2 Customer Demand 135

C.2.1 Equipment Type 136

C.3 Modelling variables 136

C.3.1 Decision Variables 137
C.4 Costs 137
C.4.1 Costs of Installing Facilities 137
C.4.2 Costs of Cable Expansion 139
C.5 Formulation of Problem 139
C.6 Tradeoffs 140

Bibliography **141**

List of Figures

1 The various worlds visited in this project 5

2 Hierarchical structure of telecommunications networks 6

3 The local access network 7

4 A Boston central office after an 1881 ice storm 8

5 Use of remote electronic devices in the access network 9

6 "Roadmap" of this dissertation 10

7 The modelling process 13

8 Dividing the access network into service areas 20

9 Structural Overview of a Network Design Tool [Ker93] 26

10 ROLAND's structure 28

11 An overview of ROLAND's LAN Editor 29

12 The ToolBar's node creation buttons 30

13 The ToolBar's remaining buttons 30

14 Screen-shot of ROLAND's LAN Editor 31

15 The Test Data Generator 32

16 The Concentrator Location Algorithm Parameter NoteBook Dialogue . . . 33

17 Adding a new algorithm to ROLAND 34

18 Bounding boxes and special flags for displaying the Centre-of-Mass Algorithm's results 36

19 The MV-network horizon-year planning problem 42

20 Solutions to LP-1 and LP-2 44

21 Solutions to LP-3 and LP-4 46

22 Tree Representation of the Branch-and-Bound Method 47

23 A decision tree for the MV-network horizon-year planning problem 48

24 (a) A convex function (b) A non-convex function 49

25 A lower bound of a function 50

26 COM example 61

27 Test network for the COM Algorithm 62

28 COM Algorithm's formation of 1 and 2 clusters 62

29 COM Algorithm's formation of 3 and 4 clusters 63

30 COM Algorithm's formation of 5 and 6 clusters 63

31 The Cluster-Add Heuristic 67

32 Test Network for Concentrator Location Algorithms 73

33 Solution produced by Add Heuristic 74

34 Solution produced by the Drop Heuristic 74

35 Solution produced by Cluster-Add Heuristic 74

36 Multipoint lines 76

37 CMST Problem Example 77

38 Result of applying Kruskal to 50 node test network 78

39 Result of applying Esau-Williams to 50 node test network 80

40 A terminal exchange 81

41 A more complete exchange 82
42 Original auxiliary graph [Ker93] 86
43 Compressed auxiliary graph 88
44 Structure of LAN Editor 92
45 The Distribution Point Dialogue Box 93
46 Concentrator and Site Dialogue Box 93
47 Link Dialogue Box 94
48 The NodeKeeper Class 95
49 Making an entry in a simple cost matrix 96
50 Gaps in the cost matrix 96
51 DP-Concentrator Cost Marix implemented with hashing 97
52 The TCCostPane Class 98
53 The Algorithm Class 100
54 Concentrator Location Parameter Dialogue 102
55 Granularity and Grouping: the shaded areas represent settlements that are typically situated near rivers 103
56 Test Data Generator Form 104
57 Random and clustered distribution points produced by the test data generator 104
58 Execution time of linear programming code 105
59 Execution times of Centre-of-Mass Algorithm and Add Heuristic 106
60 Execution time of Cluster-Add Heuristic 107
61 Relative costs of solutions produced by the Add and Cluster-Add Heuristics 108
62 Effect of `max_dist` and `num_clus` parameters on network cost 109

Chapter 1

Introduction

1.1 Motivation and Background

The giant communication networks of today are an expression of man's desire, if not need, to reach his fellow man. Since early history, the safe and economical delivery of messages across sometimes vast distances has aided the growth and downfall of kingdoms, the spread of cultures, and the formation of alliances. The impact of our generation's colossal lattices of transistors and optic fibres has seen MacCluhan's "Global Village" become more and more of a reality, while the enormous, ever-increasing financial investment in communication infrastructure, fueled by demand from all quarters, has placed an important emphasis on the efficient design and operation of these networks.

Rapid advances in telecommunications technology, in particular, has created new opportunities for modeling and optimizing the design of telecommunications networks. The complex and diverse nature of these networks poses several modeling challenges, and network designers are under pressure to choose the appropriate approach to solving their given problems.

1.2 Project Context

This dissertation presents the development of ROLAND : a program which provides network designers with an extendible palette of tools for producing economical network designs by solving various design problems relating to the *access* part of a telecommunications network.

Figure 1: The various worlds visited in this project

The local access network forms the all-important interface to the subscriber, and is the most cost-sensitive area of any telecommunications network [Row91], in which between 50% and 70% of capital and operation costs occur [Whi95]. In South Africa, the efficient planning of the telecommunications company Telkom's access network is vital for the economical provision of three to four million additional telephone lines by 2000 (*Vision 2000*). A fuller definition and description of the access network is given in the next section.

ROLAND integrates several approaches to access network design into an interactive environment which allows the designer to create network scenarios and explore various approaches to determining the most economical network topologies. These include interfaces to linear programming code, implementations of heuristics described in the literature, as well as new algorithms. The tool has been designed to reduce the effort required to add new algorithm implementations to the existing framework, and aids the network designer in comparing the solutions produced by the different algorithms.

The project draws together a number of worlds, as illustrated in Fig. 1. The Software Engineering and Human-Computer Interaction aspects of ROLAND form an important part of providing a facility that will be intuitive and useful to network designers. The various algorithms incorporated into ROLAND draw upon theory developed in the Operations Research domain, while the third "world" of telecommunications dictates the type of problems and their parameters we are trying to solve.

1.3 The Local Access Network

A concise definition of the access network is needed before examining the evolution from its rudimentary beginnings to the more high-tech networks of today. It should noted at this stage that the term *access network* will be used in this document to describe the *telecommunications local access network.*

1.3.1 Definition of the Access Network

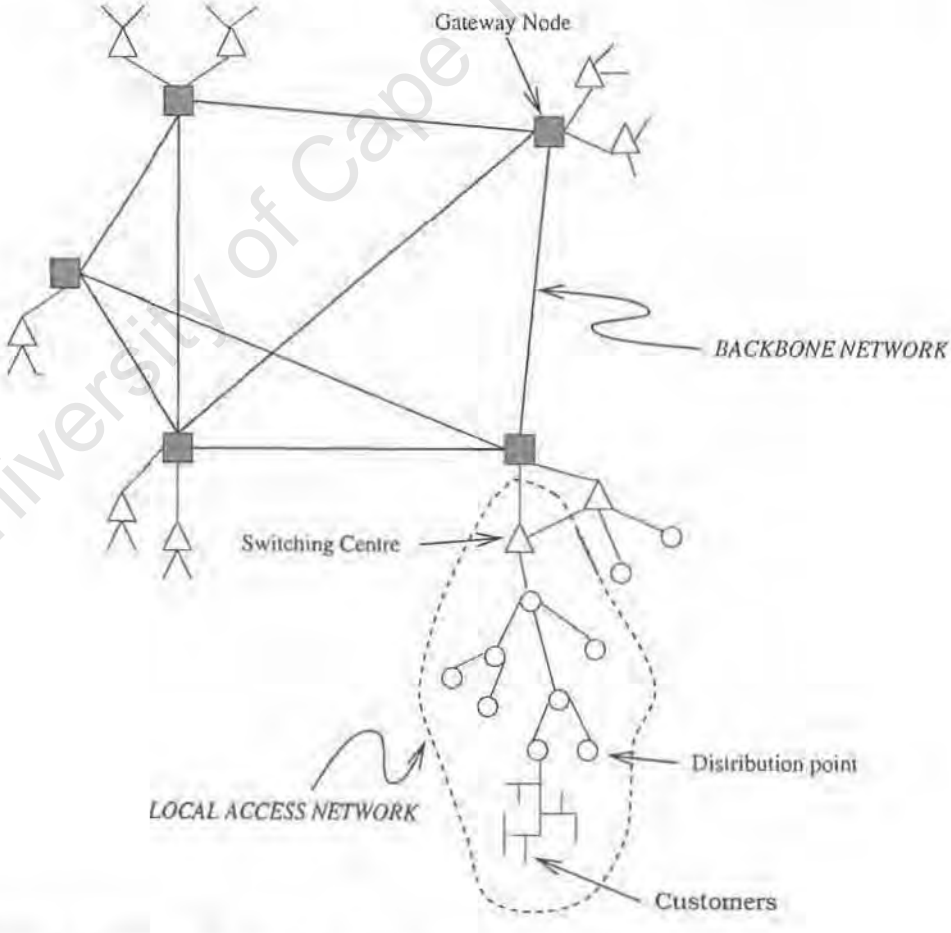

Figure 2: Hierarchical structure of telecommunications networks

The access network is hierarchical, which is typical of the various levels of a telecommunications system (see Fig. 2). It usually has a tree structure (i.e. each node of the network is connected to the local exchange via a unique path), and consists of three levels (see Fig. 3):

- **routes** - the section of the network containing all the customer nodes that communicate with the local exchange via a common link. Each route is divided into the feeder and distribution networks.
- **feeder networks** - connect the local exchange to intermediate nodes called *distribution points* (or *control points*). The number of distribution points assigned to a local exchange can vary from 20 to 200. The feeder network consists of cable types of varying gauges, or optic fibre, that are either mounted on poles, buried or installed in ducts.
- **distribution networks** - connect each distribution point to the customer premises. They tap into the feeder networks via lateral cables connected to distribution points.

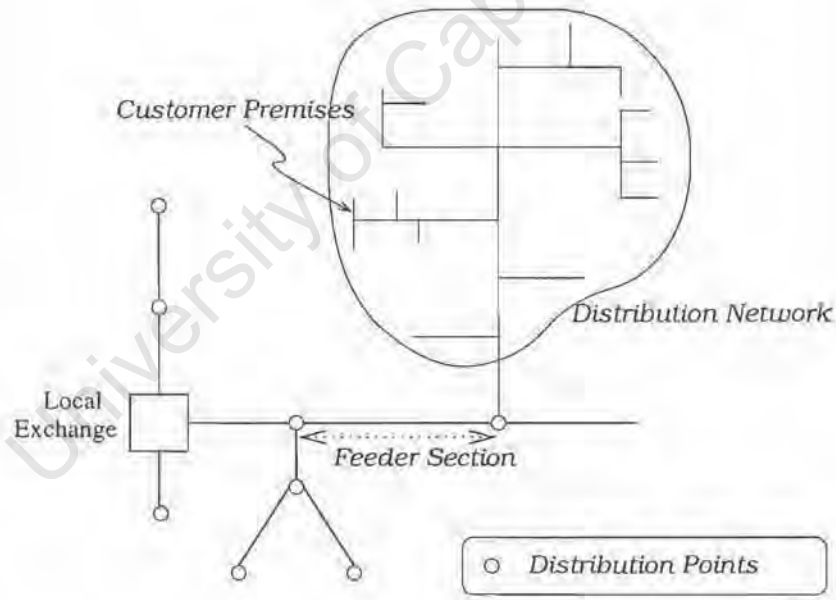

Figure 3: The local access network

The distribution networks are usually designed for *ultimate* demand, in order to exploit economies of scale and prevent disruption of service to customers resulting from the installation of new cables. The feeder networks, on the other hand, are designed to cater for *medium-term* demand. These networks therefore need to be periodically rescaled to accommodate demand growth and customer movement. The manner in which the feeder network is expanded to meet these additional needs is one of the problems that the algorithms incorporated into ROLAND try to solve. The installation of additional copper

lines from the local exchange to the subscriber is the simplest solution, but once *remote units* like concentrators are introduced into the network, the placement and capacity of the equipment can greatly influence the network cost.

1.3.2 Evolution of the Access Network

The modernization of the access network bears great strategic consequences for telecommunications companies, as the equipment deployed in this section of the network determines the services that can be provided to subscribers. This relationship has seen the steady advancement of telecommunications services, networks and technologies *globally* over the past two decades. In South Africa, however, a large section of the population are still in need of POTS (Plain Old Telephone Service), and although Telkom foresee these new subscribers soon wanting additional services, the provision of basic voice service is still the top priority.

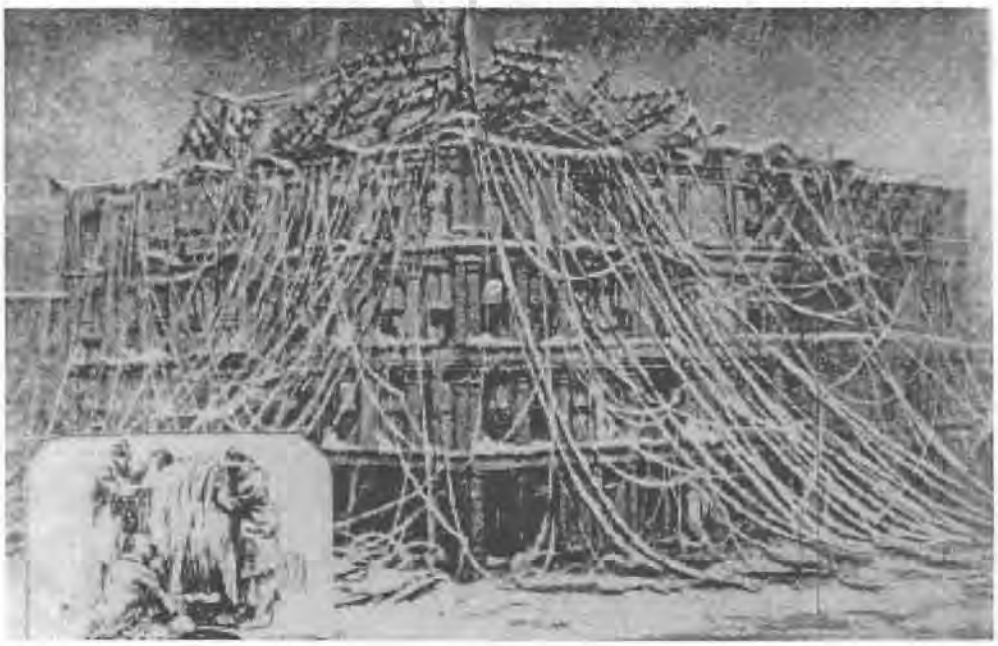

Figure 4: A Boston central office after an 1881 ice storm

Most *long-distance* telecommunication networks have almost completed the transition from analogue to digital switching technologies [BMSW91b]. The access network, however, has experienced relatively limited technological growth from the naive networks of the turn of this century.

The basic shape of today's access network is *essentially* the same as the networks that arose soon after Alexander Graham Bell first patented his invention of the telephone in 1876; in most cases, twisted copper pairs in cables connect the customer's home and a telephone company's local exchange. In the early days of telephone networks, the density of the overhead lines resulted in some interesting arrangements of wires (see Fig. 4).

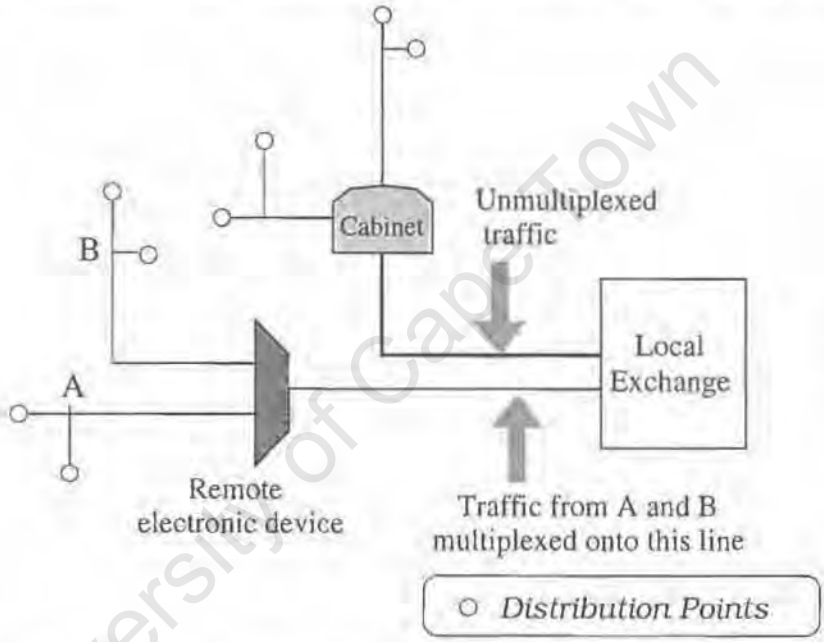

Figure 5: Use of remote electronic devices in the access network

Admittedly, better techniques for arranging the mass of twisted pairs flowing into the local exchange were developed, but it is only over the past few decades that the signal to be conveyed (moving to digital as opposed to analogue), combined with the method of transport (introduction of radio and optic fibre), has seen the traditionally copper-based networks change their shape. As a result of the new technologies now deployed in access networks, remote units (such as concentrators) have been introduced (see Fig. 5).These remote units allow traffic from a number of circuits to be compressed onto a channel whose capacity is lower than the sum of the input capacities. In many local exchanges today however, traffic still flows into the switch unconcentrated [SV90].

1.4 Dissertation Outline

Fig. 6 below gives the skeleton of this dissertation. In this chapter I have set the context of the project and provided a definition of the local access network of a telecommunications company. The layout of the remainder of the dissertation is as follows:

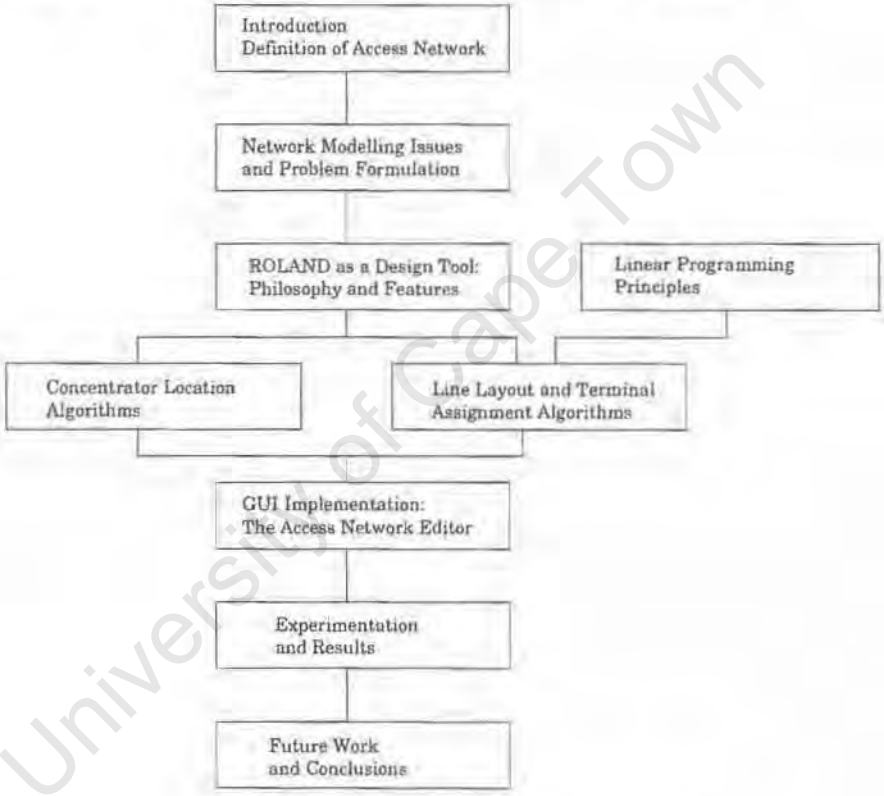

Figure 6: "Roadmap" of this dissertation

Chapter 2 starts by discussing some network modelling and design issues. It then gives an informal description of the three classes of problem that ROLAND helps to solve, namely the *concentrator location*, *line layout* and *terminal assignment* problems. A high-level summary of the types of algorithms found in the literature sets the context for the later chapters in which various algorithms and their implementations are described in detail.

Chapter 3 gives the philosophy behind ROLAND as a design tool, namely providing an environment in which new algorithm implementations can easily be added to equip

network designers with an extendible palette of tools. Using Kershenbaum's model for network design tools as a reference point, ROLAND's architecture: the nature of its various component's, plus the way that they interact, are described.

Chapter 4 introduces linear programming concepts such as duality and relaxation, and provides the formulation of the concentrator location problem as a mixed-integer programming problem. The branch-and-bound approach to solving the problem is discussed, and a relaxation algorithm for the *uncapacitated* case is also described informally.

Chapter 5 describes in greater detail the *concentrator location* algorithms integrated into ROLAND. The *Cluster-Add Heuristic* is introduced as a new hybrid algorithm, which was developed and added to ROLAND's suite of algorithms.

Chapter 6 describes the *line-layout* and *terminal assignment* algorithms packaged into the tool.

Chapter 7 gives a more detailed description of ROLAND's architecture, examining more extensively the classes introduced in Chapter 3.

Chapter 8 discusses a few results obtained by running ROLAND's algorithms on data generated by a test data generator. It also examines in greater detail the effect that changing the value of the Cluster-Add Heuristic's parameters has on the cost of the networks produced.

Chapter 9 presents the conclusions and suggests several areas for future research.

Chapter 2

Modelling and Optimizing Access Networks

In the previous chapter, we defined the telecommunications access network and saw how the modernization of the access network bears great strategic consequences for telecommunications companies, as the equipment deployed here determines the services that can be provided to subscribers. ROLAND as a design tool incorporates algorithms for the optimization of various types of access network design problems. In this chapter, we give a high level description of network models before providing an overview of the various algorithms presented in the literature for solving these problems. We also examine how the need for heuristics often arises when dealing with access network design problems.

2.1 Modelling a System

The essence of most network design tasks lies in the construction and use of a **model**. A model is an abstraction of reality. It refers to the underlying mathematical formulation of a particular system, which captures the various variables, constraints and quantities that are to be optimized. It might also be a scaled-down test network. The main reason for constructing a model is economic: it is not plausible to freely experiment on an actual network servicing paying customers. Also, modern telecommunications networks are becoming more and more complex, so a representative model is often needed to understand the system or explain it to others.

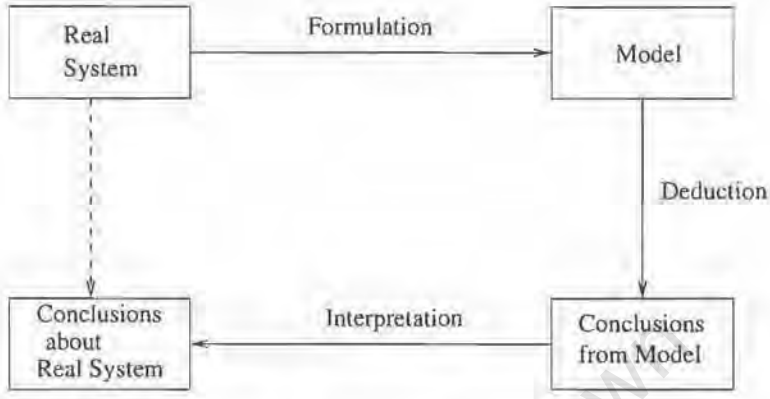

Figure 7: The modelling process

The process of modelling is illustrated in Fig. 7, and discussed next.

Model Formulation involves deciding what aspects of the real system can be ignored and which should be incorporated into the model. In many cases, this is a difficult decision. The modernization of access networks, as described in the previous chapter, has introduced new challenges in modelling these networks (and the subsequent optimization of the models) that did not arise in the traditional copper and analogue environment. These new "modern models" have created additional decision variables along with new trade-offs. For example, it is now possible to deploy remote devices such as concentrators as an alternative means to traditional copper cable expansion to increase network capacity.

A network can be modelled at various levels of abstraction, ranging from detailed engineering models of different technologies to higher level models that capture the broader pattern of network evolution. The latter scope of models include **economic** models which specify the capacity, location and timing of investment of different resources in the network. A *wide scope* of economic models exist, and several different models for access networks have been included in ROLAND to exhibit a range of modelling approaches and resulting optimization techniques. These are described later in this chapter. Economic models representing the most common class of access network design problems have been selected, as these will be useful in a broader context.

Deduction is process whereby the problem expressed by the model is solved. This might

involve solving a series of equations (if the model is a mathematical one), running an implementation of some heuristic on a computer or performing some geometrical calculations by hand. The choice of a particular method is dictated by the nature of the *model* being used. Whereas the formation of the model is a subjective process, this phase of the modelling process should not be based on people's opinions (although these opinions are sometimes inherently present in the solution method as we shall see when we discuss heuristics in Section 2.2.2). Provided that the assumptions are clearly stated and the various parameters well defined, the same model conclusions should be obtained by any party applying the formal rules of manipulation prescribed by the methods in use.

Interpretation is the next step in the process, where the conclusions drawn concerning the model from the *deduction* phase are translated into real-world conclusions. This process involves human judgement, and it is at this stage that designers must decide whether any of the aspects of the real system that were omitted during the formation of the model turned out to be in fact important. The process of determining whether a model represents the real system or not is known as *validation*, and a *valid* model is seen to provide useful information in a particular context.

The process of choosing an appropriate model for a particular system is often a difficult task. The correctness of the models is of paramount importance, as investment in the (physical) implementation of the networks is vast, to say the least. A bug in an algorithm resulting in an incorrect, yet feasible-looking design, may only be detected once the network is built. It is therefore almost always better to choose a simple solution method over a more "intelligent" one, unless there is a great difference in the quality of solution or efficiency. In choosing a model, the following ten principles [PRS76] are useful to bear in mind.

- Do not build a complicated model when a simple one will suffice
- Beware of modelling to fit the solution technique
- The deduction phase of modelling must be conducted rigorously
- Models should be validated prior to implementation
- A model should never be taken too literally

- A model should neither be forced to do, nor criticized for failing to do, that for which it was never intended
- Beware of overselling a model
- Some of the primary benefits of modelling are associated with the process of developing the model
- A model cannot be any better than the information that goes into it
- Models cannot replace decision makers

2.1.1 Network Models

Let us now draw our attention to the specific case of modelling networks, and in particular, telecommunications access networks. Optimally designing a network in order to satisfy a given set of requirements (such as the desired level of security, traffic-levels or specific technical requirements), while still minimizing the total cost of the network, is a common problem in the Operations Research world. It arises in a number of situations:

- goods transportation networks [KSK95][KH63]
- computer networks [FC72]
- centralized teleprocessing networks [WWB78][KC74][EW66]
- energy systems [LH89]
- telecommunications networks [SC95][LZC87]

A general graph model for network design problems

A simple graph model can represent the general problem of designing an optimally economic network subject to certain constraints. The set of nodes $X = \{1, 2, \ldots, N\}$ may represent breweries or some other factory, power plants, switching centres or concentrators. These nodes want to communicate with each other in some way (e.g. exchange bottles of beer, conduct telephone conversations, etc.)

In order to cater for this communication, links need to be established between selected pairs of nodes. These links may be in the form of roads, telephone lines or high-voltage electric cables. They can be viewed as edges or arcs in the case of an undirected or directed graph respectively. This graph then represents the structure or *topology* of the network, and is referred to the graph *associated* with the network under consideration.

An edge connecting nodes i and j is denoted by $u = (i, j)$. If U represents the set of all edges, the associated graph is denoted by $G = [X, U]$.

This is an obvious and natural representation of the structure of the graph, and numerous well-established graph algorithms can be used to test for various properties of the network. For instance, it is possible to check whether the network can survive the failure of any of its links by testing for the 2-edge connectivity.

In addition to capturing the structural nature of the network, it is also necessary that a model include information about these additional two parameters:

1. some measure of the *intensity of communication* to be established between nodes (be it the number of bottles of beer to be delivered to various shops from a factory or the number of voice channels on a telephone link).

2. some measure of the *amount of resources* available on the various links of the network (in the case of a beer distribution network, this could refer to the capacity of the trucks delivering the cases of beer).

In some *dynamic* models, the notion of time is also captured and factors such as the lifetime of equipment, interest rates on capital, growth of demand and other practical considerations such as the time to install equipment have to be taken into account. In addition, these models should ensure that resources are provided in step with the increase in demand and in the optimal increments. For instance, choosing large increments benefits from the economy of scale and results in fewer interruptions to service, but may lead to some of the resource going spare. The provision of resources in **small** increments on the other hand gives higher unit costs and greater interruption to service, but has the advantage that less of that resource is wasted.

Starting with this basic graph model of a network, Minoux [Min89] gives an excellent cross-section of the types of network design problems that exist in the literature (and in practice),

and the reader is referred to that work for a description of some of the more general network problems that exist but are beyond the scope of this thesis.

2.2 Approaches to Designing Networks

Before proceeding to describe the different economic models and associated algorithms that have been implemented and integrated into ROLAND, it would be useful to examine the various ways that networks can be designed optimally, and say *when* and *how* models are actually used in the design process. We shall also examine why a computer should be used at all in the design task.

2.2.1 The "Hands-on" Approach

Surprisingly, a number of networks are still designed manually with no formal rules or techniques being used. In the case of access network design, I have observed how engineers with years of experience in the field are used to decide where pieces of equipment should be placed and when this should occur. Although certainly the most flexible of all techniques, this approach suffers from a number of drawbacks in that the process is rarely quantitative, and decisions are made subjectively, and sometimes inconsistently. If the designer actually produces a successful design, it difficult to repeat the process in similar circumstances. Often, the design task involves too many variables for the designer to consider all the possible alternatives, and some form of automated tool could be useful to provide clues as to where equipment should be positioned.

2.2.2 Heuristics

Heuristics are like rules of thumb that are selected on the basis that they will aid solving a particular problem. Simon [Sim61] uses the term "heuristic" to denote: *"any principle or device that contributes to the reduction in the average search to a solution"*. They often embody good ideas that are discovered from design experience, that can be made quantifiable and repeatable.

Several alternative heuristics can be applied to the same problem, and the one yielding the best solution chosen. This means that it is possible to refine heuristics, keeping those "rules

of thumb" that work well, and discarding others that do not.

In an automated system like ROLAND, many alternative approaches can be tried on the same problem, and an objective choice based on performance can be made. New heuristics can be added to the system, allowing a team of designers to share ideas.

Some heuristics are applied in specific contexts, while other are more general and are applicable in many domains. A widely used heuristic that is useful across many types of networks is the **greedy algorithm**. Some of the algorithms described in later chapters make use of this technique.

The greedy algorithm is based on the idea that inexpensive networks tend to contain cheap links. When confronted by a series of choices, the greedy algorithm always chooses the best possible choice at each stage. This generally produces a good solution, but not necessarily the *optimal* one. It is possible that the optimal solution does not contain some of the least expensive links.

Heuristics are a valuable tool because they allow feasible solutions to computationally-exhausting problems to be found in a reasonable amount of time. They allow good solutions to be found without having to explore the entire solution space, by considering only those solutions that possess certain characteristics (indicating they belong to a good solution). Even in the case where it is feasible to explore the entire solution space, heuristics are often used to obtain feasible solutions quicker. This may be useful particularly in the early phases of the design process where the exploration of different scenarios is more important than obtaining refined solutions. Kuehn *et al.* [KH63] view heuristic programming as "an approach to problem solving where the emphasis is on working towards optimum *solution procedures* rather than *optimum solutions*".

2.2.3 Formal Optimization Techniques

Finding an optimal solution to a problem involves exploring each of the alternative solutions and choosing the best one. The set of all possible solutions is known as the solution space and even for seemingly small problems, the solution space can grow to be extremely large. Consequently, the exhaustive exploration of the solution space is rarely a feasible exercise. This can be illustrated by a simple example. Consider a network consisting of only 20 nodes. This has $(20 * 19)/2 = 190$ potential links between every node in the network. This

gives rise to 2^{190} possible different networks.

Finding the "best" solution in practice means minimizing or maximizing some objective function, which is expressed in terms of the design variables. In network design problems, cost is a common objective function, but others might include reliability, or quality of service.

The best value of the objective function (be it the maximum or minimum) is known as the **optimum**. There is a class of algorithms, which are guaranteed to produce optimal solutions, but only for a certain well-defined class of problems known as *linear programming problems*. Chapter 4 of this dissertation is dedicated to linear programming fundamentals, and examines this class of problems in greater detail.

2.3 Designing Access Networks

2.3.1 Feeder Network Expansion

As noted in the previous chapter, the *feeder* sections of access networks are designed to cater for *medium-term* demand, and periodically need to be rescaled to accommodate demand growth and customer movement. It is not surprising therefore to find that the emphasis in the literature is on models for determining how the capacity of the current feeder network should be expanded to support the growth in demand for voice and other types of service.

Expanding the capacity of feeder cable capacity is one approach to catering for an increase in the service demand. In modern access networks, the installation of remote electronic devices such as concentrators represents a complementary approach to the feeder capacity expansion problem. The placement of concentrators in the access network is similar to the terminal layout problem encountered in centralized teleprocessing systems common in the 1960s and 1970s, with the central computer analogous to the local exchange, and the distribution points with the terminals. This problem typically consists of three phases:

1. selecting the number and positions of the concentrators : the *concentrator location* problem;

2. assigning each terminal a specific concentrator : the *terminal assignment* problem;

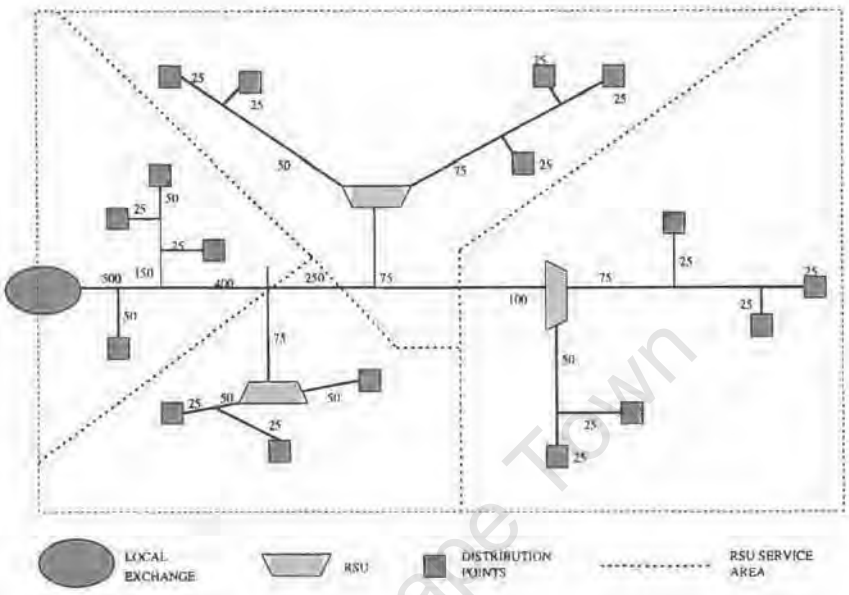

Figure 8: Dividing the access network into service areas

3. deciding how to connect each terminal to its assigned concentrator: the *line layout* problem.

2.3.2 Traditional Approach to Access Network Design

Traditionally, access network design has been manual process, in which the area under consideration is divided into various *service areas* according to the location of the distribution points (see Fig. 8). The locations of the distribution points is decided from a map of the forecasted demand for the area. Concentrators are then located in the centre of these service areas.

The decisions relating to the *type of technology* (e.g. copper, wireless, optic-fibre, etc.) to deploy in the access network and *how* to arrange the various components is a process that is dependent on many factors. It involves so many variables that traditionally this has resulted in experienced network designers making these decisions using past design experience and rules of thumb. The creation of a model to capture all these variables is a highly optimistic task, and the solution of such a model would be even more ambitious. Recent research into the use of a knowledge-based system for determining appropriate technology for an access network service area is being conducted by [Ruc96], and has confirmed the difficulties

in forming a useful model to capture the many constraints and variables inherent in the problem.

2.3.3 Access Network Models

As discussed earlier, the informal, ad-hoc design of networks suffers from a number of drawbacks, and the formalization of these techniques would help network designers to have a more consistent approach to their task. The use of different models for different aspects of the design phase is one solution. The models included in ROLAND are *economic* models and do not consider the technical aspects of the equipment being deployed. The choice of technology will admittedly determine the nature (such as the capacity) of the devices placed in the network, but as such, these economic models are generic and independent of the type of technology. Models and their associated solution methods for solving all three classes of this problem are integrated into ROLAND.

A common approach in the literature is to first determine the concentrator locations and terminal assignments using a single model, known as the *Capacitated Concentrator/Plant Location Problem* (CPL). This is a special case of the so-called Plant Location Problem, which involves picking a subset of an *a priori* specified set of plant locations so as to minimize the cost of production and shipment of specified quantities to a set of given customers [Min89]; the plants in this case, we associate with concentrators.

Other types of models presented in the literature include:

- the *service section or switching section connection model*, in which the distribution points are partitioned into a subsets called *service sections* which contain a number of potential concentrator sites (Luna *et al.* [LZC87]), and
- *tree network models*, whose solution methods use recursive procedures to exploit the tree structure of the network. Helme *et al.* [HJS88] present such a model, allowing multiple processors in series.

These models, amongst a few others, are summarized in Table 1.

Model type	*Restrictions and Features*	*Solution Method*
Centralized teleprocessing design model*	*Restrictions* : Point-to-point connection from concentrator to switching centre; single service type	.
capacitated concentrator location*	*Restrictions* : no bifurcated routing; *Features* : Selected design has a double star topology	optimization-based and local improvement heuristics
terminal layout*	*Restrictions* : Only a single cable size can be installed; *Features* : Terminal layout model usually applied with fixed concentrator location	Optimization-based approaches
Switching centre connection model * (Luna *et al.* [LZC87]	*Restrictions* : Prespecified service sections, potential concentrator sites in each section, and routing from distribution points to concentrator sites; single service type *Features* : Can incorporate proximity restrictions; exactly one concentrator site selected for each service section	Shortest path - local improvement heuristic
Tree network model without backfeed+	*Restrictions* : No backfeed *Features* : Can accommodate multiple processors in series	Dynamic programming
Network design model with layered network*	*Restrictions* : Cannot accommodate customer proximity *Features* : Fixed and (linear) variable processor and cable costs; conversion ratios for different processor types are compatible; can accommodate multiple customer services and multiple processors in series	dual ascent
Tree covering model+	*Restrictions* : Contiguity assumptions; no bifurcated routing; *Features* : Can incorporate proximity restrictions	Polynomial solvable for new networks; Lagrangian Relaxation and dynamic programming

Table 1: Telecommunication Access Network Planning Models : * refers to models applicable to general networks, while + refers to models limited to tree networks

2.4 Solution Methods included in ROLAND

2.4.1 Line Layout Problem

The connection of terminals to their assigned concentrator is non-trivial in the case of multidrop or multipoint lines where a number of terminals are allowed to share the same line to the concentrator. Several algorithms for determining the layout of such lines are integrated into ROLAND:

Kruskal: The multidrop line layout problem finds the minimum spanning tree of a graph, with additional constraints on the subtrees connected to the root (i.e. the concentrator to which we are connecting the terminals). This problem is known as the *Constrained* Minimal Spanning Tree Problem (MST) [KvS72]. A greedy approach, similar to that suggested by Kruskal for the unconstrained case, can then be used to solve the line layout.

Esau-Williams: The greedy approach suffers from the fact that terminals situated far from the concentrator (which are relatively expensive to connect directly) are sometimes left stranded when their neighboring nodes are merged into subtrees that "fill up". Esau and Williams [EW66] suggest a solution to this problem by associating a

trade-off function with each link that gives preference to terminals situated relatively far from the concentrator.

Unified Algorithm: By generalizing the data structures used in Esau-Williams, Kershenbaum and Chou developed a Unified Algorithm that encompasses a family of widely used algorithms, including Kruskal, Prim and Vogel's Approximation Method [KC74].

2.4.2 Terminal Assignment Problem

There are two approaches that can be used to solve the terminal assignment problem:

greedy approach: Here, terminals are greedily assigned to the nearest concentrator with remaining capacity. The problem with this approach is that terminals can, in many cases, be stranded and forced to form costly connections with concentrators situated far away.

semi-greedy (or alternating chain) approach: Provided that the sum of the concentrators' capacities exceeds the sum of the terminals' demands, it is possible to obtain an optimal solution to the terminal-assignment problem by determining chains of reassignments that occur when an economically-beneficial assignment is prevented from occurring due to a concentrator whose capacity is exceeded.

The greedy algorithm certainly runs faster than the semi-greedy one, but the loss in quality of solution motivated the inclusion of an alternating-chain algorithm in ROLAND's family of solution methods.

2.4.3 Concentrator Location Problem

A number of approaches exist for solving the concentrator location problem. The methods that ROLAND adopts are listed below:

Linear Programming: The concentrator location problem can be formulated as a (0,1)-integer programming problem and solved optimally using linear programming techniques [ER66][WWB78]. This formulation and a discussion of the linear programming

techniques used to solve the problem are given in Chapter 4. Due to the complexity of the problem (NP-hard), exact solutions can only be found for a relatively small number of nodes, typically less than fifty. ROLAND includes an interface to the linear programming application, **lp_solve**, which uses *branch-and-bound* to solve the mixed-integer formulation.

Centre-of-Mass (COM) Algorithm: The Centre-of-Mass Algorithm identifies natural clusters of traffic [MS77]. It has the advantage of being computationally efficient, and is especially useful in identifying candidate sites when no potential locations have been provided.

Add and Drop heuristics: The classical Add Heuristic [KH63] is a greedy algorithm, which starts with all the terminals connected directly to the local exchange. The savings that can be obtained by installing a concentrator at each potential site are then calculated, and the concentrator which offers the most savings is installed. The algorithm proceeds until no more savings can be obtained, or all the concentrators have been installed. The Drop Heuristic works in the same manner, but in the opposite direction, starting with all the concentrators installed.

Relaxation Algorithm: The three heuristics described above guarantee no quality on the solutions they produce. By relaxing some of the constraints of the concentrator location problem, Erlenkotter developed an algorithm, for the *uncapacitated case*, which places a lower bound on the quality of solution it produces [Erl78]. Guignard and Spielberg present a relaxation algorithm for the *capacitated case* [GS79].

Cluster-Add Heuristic: We have developed a new hybrid algorithm which combines COM and the Add heuristic to form an algorithm which first installs concentrators located near "centres-of-mass". This decreases the number of iterations needed by the classical Add heuristic.

Chapter 3

ROLAND as an Adaptive Design Tool

ROLAND's architecture provides an extendible framework for the integration of multiple optimization algorithms. This chapter gives a motivation for ROLAND's various components, and describes each module's functionality, while the implementation details are left till Chapter 7. Kershenbaum's model for network design tools [Ker93] has been used as a reference point. A test data generator that was developed is also presented, before an example illustrating how new algorithm implementations can be incorporated into the tool's framework concludes the chapter.

3.1 Network Design Tools

The construction of a network design tool should take into account the needs, experience and capabilities of the system user [Som92]. In particular, the user should not be burdened with having to collect and process the inputs to the tool, and should be able to interact with the data at *all* stages of the design process (both before and after algorithms have been run on the network).

Kershenbaum [Ker93] presents a high level structural overview of what he believes a good network design tool should look like, in which the module's are organized into four areas - the front end, database routines, algorithms, and utilities. This model is shown in Fig. 9.

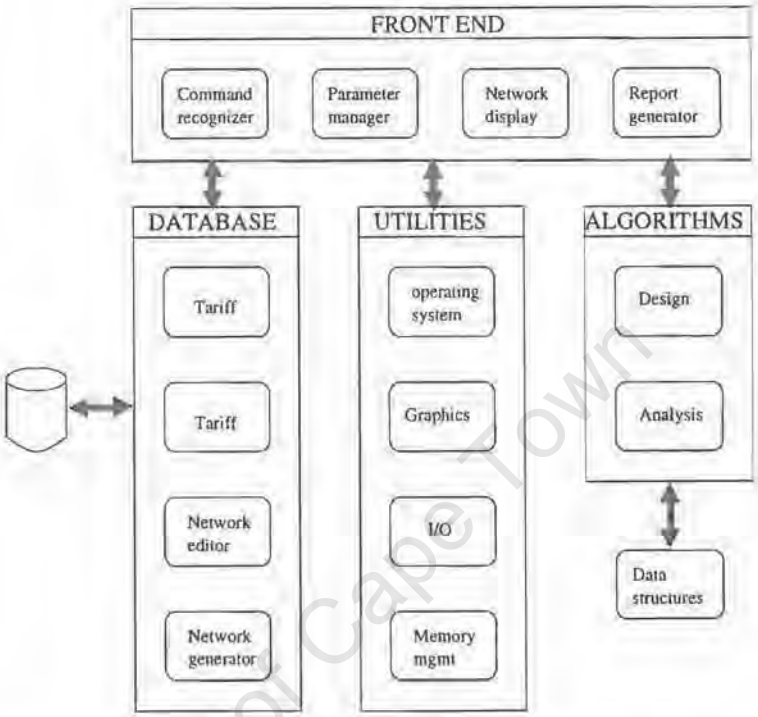

Figure 9: Structural Overview of a Network Design Tool [Ker93]

Front End The Front End is the module responsible for the input and output interfaces to the user. It presents the user with facilities such as the various menus for selecting functions, forms for displaying results, and an environment in which to edit networks.

The implementation of the front end is dependent on the tool's environment, both hardware and software. The capabilities of the end user's systems therefore needs to be considered, especially in terms of factors such as graphics capabilities.

The network display and editing capabilities also require careful planning by the tool designer. An example of a design choice is deciding on how a user should be able to modify a node's properties. A number of alternatives for selecting the node exist:

- move a mouse cursor over the node and click a button
- press an arrow key to scroll through a list of node ID's and press enter to select the desired node
- type the node ID

Many users prefer to use the mouse for graphical input, but each of these three

approaches has its advantages. For instance, the second method is most probably fastest where there are a small number of nodes, and if the list is alphabetical.

When designing the network editor that allows the user to specify nodes, links, and requirements, it should be remembered that few users have the patience to graphically enter a large network node by node. Some bulk editor needs to be provided, where a network can be specified textually. An interactive editor is good for editing a design: adding a few new nodes or moving nodes in a network.

Database modules When dealing with realistic problems, the design tool needs to be able to handle the bulk data needed by the design process. It is preferable to use existing databases, as these databases are more reliable and the need to re-enter input manually, is removed.

The database module should allow the user to maintain sets of files associated with each design problem. An interactive design tool such as ROLAND allows the user to explore many alternative designs to be explored, most of which are not successful. The alternative designs need therefore to managed and stored in some way.

Algorithms This module incorporates the algorithms used for analyzing or optimally designing networks. When implementing these algorithms, it is very difficult to verify their operation, and it is easy to introduce subtle bugs that yield reasonable, yet infeasible designs. There is no foolproof way of avoiding this situation, but when it comes to choosing between a simpler or more "clever" algorithm, it is almost always better to choose the simpler one, unless there is a substantial increase in the efficiency or quality of solution yielded by the more complex algorithm.

Another benefit of using simpler algorithms lies in the fact that these algorithms tend to rely less on the specifics of a particular network scenario and can be applied in other situations. The basic principles of network design, and the basic algorithms used, do not change significantly as the technology develops, although specific tradeoffs may change if, for example, one type of network component becomes much less expensive. This is another reason to describe the network design problems generically, in terms of basic principles and device characteristics, rather than in terms of specific devices which become outdated very quickly.

Taking this approach also leads to a reduction in the amount of code, with the same

code being used in number of different contexts. The code then also tends to be more reliable, as it is thoroughly tested. Using common data structures and operations in the implementation of algorithms also helps to achieve the goal of code reduction.

Utilities Kershenbaum describes the Utilities module as the collection of interfaces to the operating system that are machine-dependent. These include primitives to open and close files, graphics routines to draw points or lines, and auxiliary packages such as database management programs. These utilities are normally included in the programming language being used to implement the tool. It is important that efficient and reliable languages and packages be chosen that are widely available across computer systems and will not impede migration.

3.2 ROLAND's Architecture

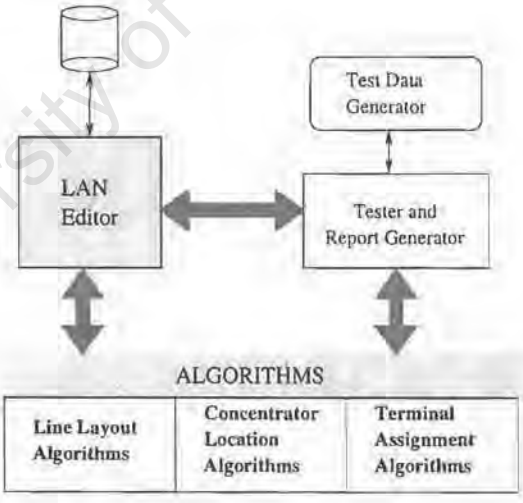

Figure 10: ROLAND's structure

Fig. 10 is a structural diagram of ROLAND, showing how the major components fit together and interact. ROLAND's Local Access Network Editor allows the user to interactively create and edit access network scenarios (e.g. the specification of subscriber demand and placement of equipment), and then apply algorithms to the specified networks. The test data generator is able to generate random data (either with uniform distributions, or clustering), while the Tester (which incorporates the Forms Module) allows the performance results

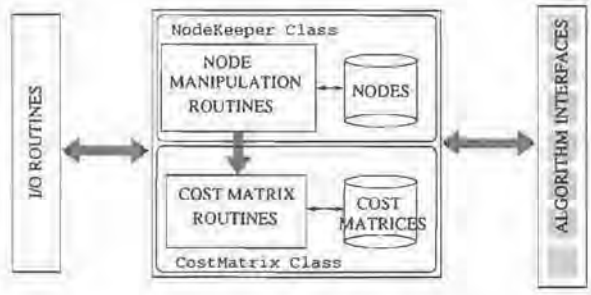

Figure 11: An overview of ROLAND's LAN Editor

(e.g. time taken and total network cost) of algorithms to be generated and displayed. The implementation details of these three components are described in detail in Chapter 7.

3.2.1 The LAN Editor

The basic architecture of the *LAN Editor* is illustrated in Fig. 11. The editor consists of two primary classes: class `NodeKeeper` and class `CostMatrix`. The cost function used by ROLAND to determine the values in the terminal-terminal and terminal-concentrator cost matrices is presently based on Euclidean distance. A new more descriptive function can easily be added to the `CostMatrix` class if need be.

Common data structures and the associated functions for displaying and manipulating a few common network nodes are included in the `NodeKeeper` class, and any new network objects that need to be represented should be placed here. Utilities for saving and loading network scenarios are also included. The tool thus caters for the specification of networks textually (as recommended by Kershenbaum), as well as provided the more convenient interactive editing capabilities.

A screen-shot of the LAN Editor is shown in Fig. 14. ROLAND has been written using the zApp Class Library, which allows it to be compiled across a number of platforms. ROLAND allows multiple editor windows (known as access network views) to be open, which is useful when comparing the application of alternative algorithms on the same network. In addition, ROLAND also allows users to:

- create, manipulate (edit properties, move, select etc), and delete distribution points, links and concentrator nodes. The toolbar buttons displaying these options are shown in Fig. 12 and Fig. 13.;

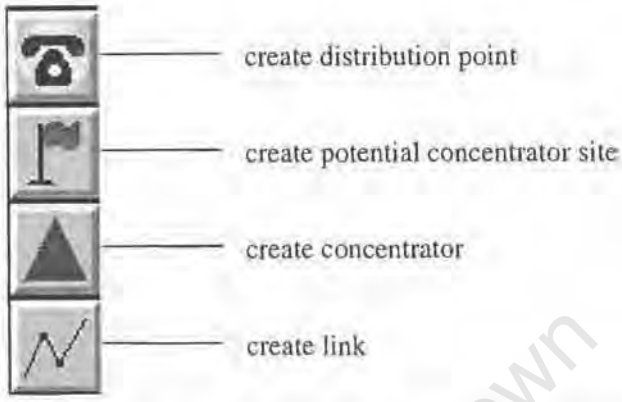

Figure 12: The ToolBar's node creation buttons

- place a map or other bitmap image in the background;
- save and load access network data;
- copy the current access network to another LAN editor window;

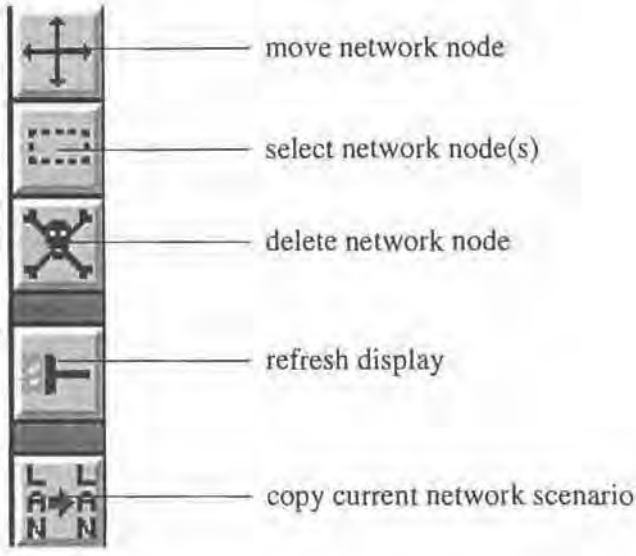

Figure 13: The ToolBar's remaining buttons

Presently the tool does not cater for the representation of any geographical or topological data, as this was deemed beyond the scope of the project. An interface to a Geographical Information System (GIS) would allow such data to be stored and displayed. Work done in

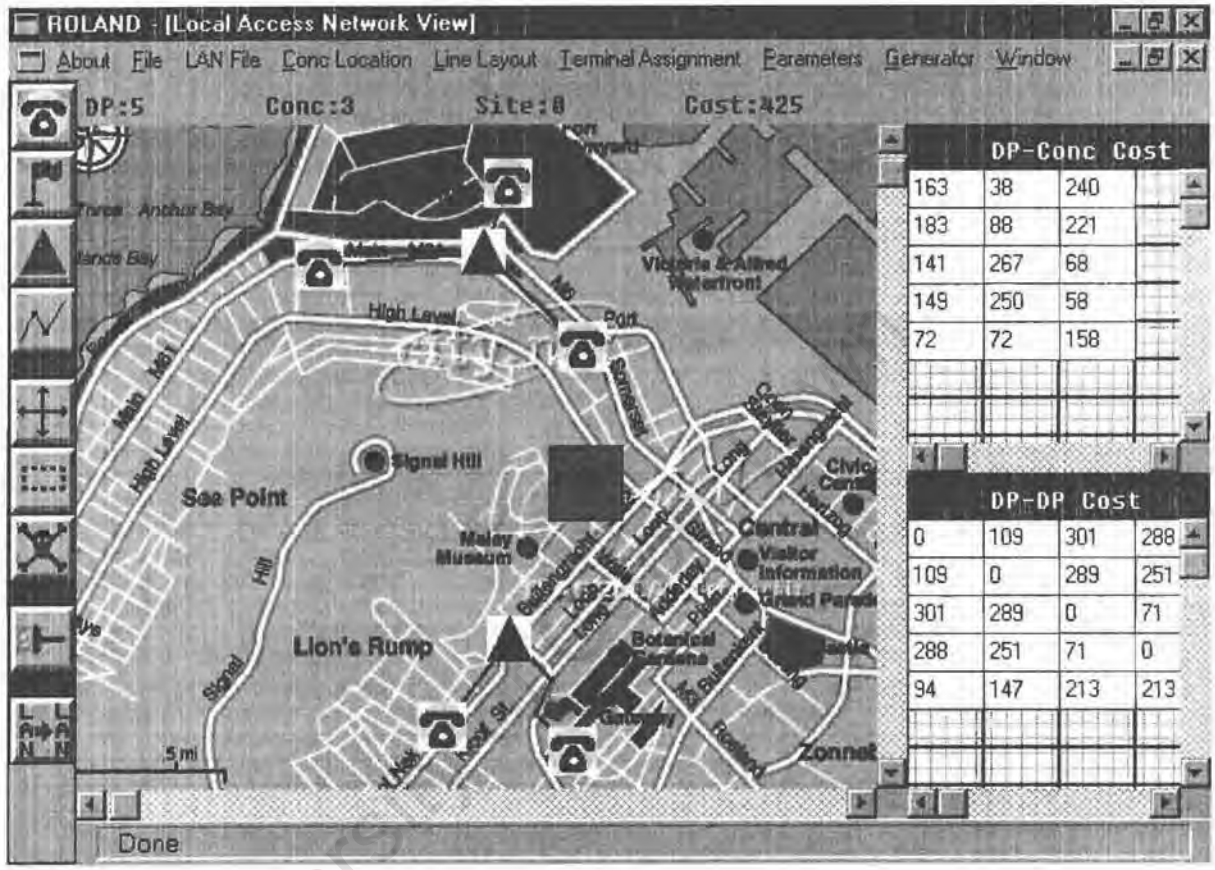

Figure 14: Screen-shot of ROLAND's LAN Editor

the area of using a GIS as part of a knowledge based system for determining the position of equipment can be found in [PS97].

3.2.2 The Algorithm Module

The Algorithm module consists of three main parts: the common data structures (with their associated operations) which are shared between algorithm implementations, initialization routines and the actual implementations themselves.

The common data structures typically include lists, vectors or stacks that may be used by multiple algorithm implementations. Similar algorithms may require certain data structures to be initialized in the same way, so a collection of generic initialization routines is justified. During a typical initialization, relevant information about the current network (which is

required by the algorithm in question) is transferred from the `NodeKeeper` class into the appropriate data structures.

Re-using code by making use of generic data structures and choosing simpler algorithms over more complex ones makes the implementations easier to understand for someone wishing to modify an algorithm.

3.2.3 Test Data Generator and Form Module

Test Data Generator

Figure 15: The Test Data Generator

It was hoped that real test data would be made available for submitting to ROLAND's algorithms, but unfortunately this was not forthcoming. A *test data generator* was therefore developed to produce certain types of test data for assessing algorithm behaviour and performance. Both random and clustered data (or a combination of both) can be generated and the resulting data can be viewed in a separate pane before converting it into a corresponding network in a LAN Editor window. The test data generator's window is shown in Fig. 15. Test data can be read or saved to file, and printed. The tool also allows test data to be converted into an actual network in a LAN Editor window.

Form Module

The *Form module* comprises the collection of the various dialogue boxes (and their associated code) used by ROLAND's LAN Editor, Test Data Generator, for specifying algorithm parameters and for reporting algorithm performance. As new implementations are added, changes or extensions may need to be made to certain forms. Some forms have been designed with future change in mind; for instance, the dialogue for specifying a concentrator location algorithm's parameters is a *notebook* dialogue (see Fig. 16): as new implementations are added to the tool, new "pages" for the implementations' parameters are added to the dialogue.

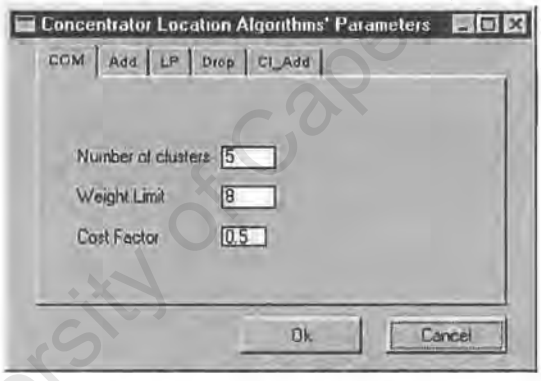

Figure 16: The Concentrator Location Algorithm Parameter NoteBook Dialogue

3.3 Extending ROLAND

ROLAND's three major components have been designed with the goal of allowing new algorithm implementations to be easily and gracefully integrated into the tool's existing framework. The use of an object-oriented programming language (C++ in ROLAND's case) and a user interface class library (zApp by Roguewave) facilitates the re-use of code and assists in achieving this goal of extensibility. The benefits of using C++ and zApp are described in greater detail in Chapter 7. After examining the need for extensibility in a design tool, this section shows the various steps that need to be followed when adding a new algorithm to ROLAND, and concludes by presenting a case study illustrating the application of these steps.

3.3.1 Finding the right path to the best solution: the need for extensibility

One of the major difficulties with using an existing implementation of an algorithm to solve these types of problems is the loss of the ability to explore alternative solutions. Certain algorithms are naturally better suited to certain types of data. Typically, implementations allow the user to change various algorithm parameters, but the ability to investigate solutions to a wide and diverse range of possible network scenarios demands a more general approach. The telecommunications and operations research literature appears to offer few suggestions in this regard.

Experience in visualization suggests that the ability to interact with the solution process helps in discovering the best solution method [Car95]. Being able to steer an algorithm while it is running is one possibility. Another approach is to make a range of different algorithms available, with which the user can generate a number of solutions to analyze and compare. This is the approach that ROLAND takes; as new algorithms are developed and implemented, they can simply be added to the existing framework.

3.3.2 Adding a new algorithm

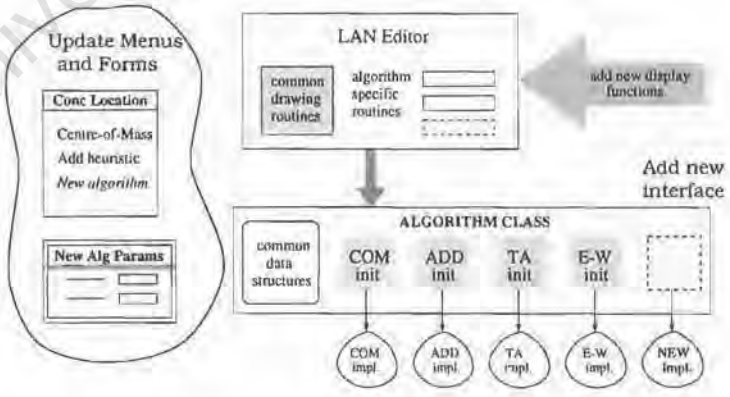

Figure 17: Adding a new algorithm to ROLAND

Fig. 17 shows how new implementations of algorithms can be incorporated into the existing frame-work. In most cases only a few, relatively small alterations have to occur to existing code. The new algorithm can make use of the pool of common data structures in the `Algorithm` class; any new structures needed by the algorithm can be bundled with its

implementation class. Most drawing routines (e.g. the display of a newly installed concentrator) are found in the existing routine library, but in some cases (e.g. the Centre-of-mass algorithm described in Chapter 5, where clusters of points need to be displayed), customized routines may need to be written.

3.3.3 A case study

The steps described in the previous section are now illustrated by means of an example. We examine the amount and nature of the coding that needed to take place to include a new algorithm into ROLAND's `Algorithm` class. The algorithm we have chosen to illustrate this is known as the Centre-of-Mass Algorithm and is described in Chapter 5.

Coding the algorithm

The implementation of the COM Algorithm makes use of some common data structures for storing the demands and positions of each distribution point. *Heap objects* are used to used to store the distances from each distribution point to all of the others.

Writing the initialization procedure

The initialization procedure calls the smaller routines responsible for initializing the distribution point data structures and cost matrix specifying the "distance" between each pair of distribution points. The values of the algorithm parameters also need to be fetched from the relevant Dialogue object.

Modifying NodeKeeper class

The COM algorithm suggests possible locations for concentrator sites, and these locations need to somehow be represented on the screen. A list of co-ordinate pairs was added to the `NodeKeeper` class for this purpose.

Modifying the display and mouse-event handling code

The algorithm works by identifying clusters of distribution points, with the centres of each cluster being proposed as a potential concentrator site. After running to completion, the

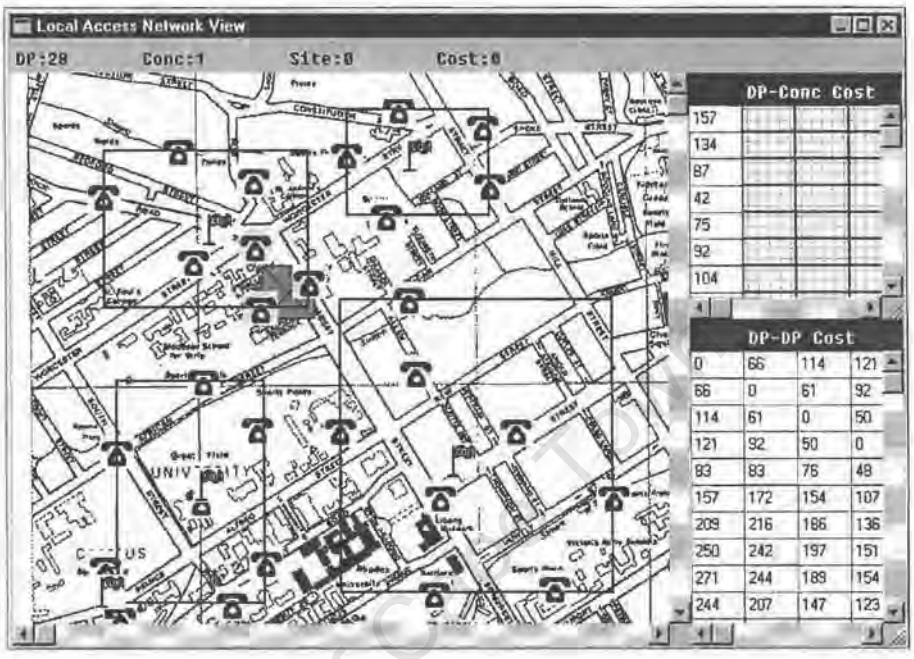

Figure 18: Bounding boxes and special flags for displaying the Centre-of-Mass Algorithm's results

clusters are displayed by drawing bounding rectangles around each one, with a special COM flag being placed in the middle of each rectangle. A new function was written to draw these bounding boxes, while the function responsible for displaying the various objects on the network editor's canvas was extended to draw the COM flags. The *mouse-click event handler* needed to be modified to check for any mouse-clicks on the special flags as this event triggers the installation of a new concentrator at this point.

Adding new parameter form and menu items

A new form for allowing the user to specify the COM Algorithm's parameter needed to be added to the *Notebook Dialogue* discussed earlier and shown in Fig. 16. This was created using a resource editor. Items also needed to be added to the editor's menu that allow the user to invoke the COM algorithm or alter its parameters by displaying the parameter form.

Chapter 4

Linear Programming Fundamentals

Determining an optimal network design often involves forming and solving a *mathematical* model of the problem. Linear programming refers to a specific class of these models. This chapter introduces certain linear programming fundamentals that are needed to understand the formulation of the concentrator location problem as a mixed-integer linear programming problem. Various solution techniques are also explored.

4.1 Linear Programming

4.1.1 Definitions

Many programming problems are concerned with the allocation of scarce resources (materials, time, capital etc.) in the "best" possible way so that profits are maximized and costs minimized. The term "best" implies the existence of a number of alternatives, and this "best" choice is usually found by solving a mathematical problem. *Linear programming*, in particular, refers to a class of programming problems that meet the following conditions [NW88]:

1. The decision variables used in the problem formulation are non-negative.
2. The criterion for selecting the "best values" can be expressed as a linear function of the decision variables. This function is usually referred to as the *objective function*.

3. The operating rules governing the process (i.e. specifying the limitation of the various resources) can be expressed as a set of linear equations and/or linear inequalities. This set is usually referred to as the *constraint set.*

4.1.2 Integer Programming

Integer programming deals with problems of maximizing or minimizing a function of many variables subject to a set of equality and inequality constraints, where *integrality restrictions* exist on one or more of the variables. If we impose the constraint that the function to be maximized and the constraints are linear, we arrive at the class of problems known as *linear mixed integer programming problems.* These discrete optimization models are able to represent a rich variety of problems, including the concentrator location problem introduced in Chapter 2.

Formally, we can write the *linear mixed integer programming (MIP)* problem as follows:

$$max\ \{cx + hy : Ax + Gy \leq b, x \in Z_+^n, y \in R_+^p\}$$

where :

- Z_+^n is the set of non-negative integral n-dimensional vectors;
- R_+^p is the set of non-negative real p-dimensional vectors;
- $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$ are the variables or unknowns.

An instance of the problem is specified by the tuple (c, h, A, G, b) where:

- c is an n-vector
- h is a p-vector
- A is a m by n matrix
- G is a m by p matrix
- b is an m-vector

This problem is called *mixed* because of the presence of both integer and real variables.

The set S = $\{x \in Z_+^n, y \in R_+^p, Ax + Gy \leq b\}$ is called the *feasible* region , and $(x, y) \in S$ is called a *feasible* solution. An instance of a problem is said to be *feasible* is the set S is not empty.

The *objective function* in this case is $z = cx + hy$ and a feasible solution (x^0, y^0) for which the objective function is as large as possible, i.e.

$$cx^0 + hy^0 \geq cx + hy \qquad \forall (x, y) \in S$$

is called an *optimal solution*. The value of the objective function for $cx^0 + hy^0$ is then called the *optimal value* or *weight* of the solution.

Sometimes, a feasible instance of MIP may not have an optimal solution; the instance may in fact be *unbounded* which means that for any $\omega \in R^1$ there is a point $(x, y) \in S$ such that $cx + hy > \omega$. The notation $z = \infty$ is usually used to denote an unbounded instance. Every feasible instance of a MIP either has an optimal solution or is unbounded [NKT89].

The *linear (pure) integer programming (IP)* problem:

$$max\{cx : Ax \leq b, x \in Z_+^n\}$$

is the special case of MIP in which there are no continuous variables, while the *linear programming problem* (LP):

$$max\{hy : Gy \leq b, y \in R_+^p\}$$

is the special case of MIP in which there are no integer values.

It is often the case that the integer variables are used to represent logical relationships and are thus constrained to be either 1 or 0. We then obtain the *0-1 mixed-integer linear programming (0-1 MIP)* problem. Here, $x \in Z_+^n$ is replaced by $x \in B^n$, where B^n is the set of n-dimensional binary vectors.

4.2 Examples of Two Models

4.2.1 Concentrator Location Problem

This section formulates the concentrator location problem as a *0-1 mixed-integer linear programming problem*, where the installation of concentrators is represented by the use of (0,1)-integers [ER66][WWB78]. This general formulation is known as the *capacitated plant location problem* (CPL).

It should be kept in mind that any mathematical formulation of a network-design problem is almost always an approximation. It is usually the case that the model can not take into account *all* the constraints or cost factors belonging to a particular problem [LH89] and factors such as the availability of land, existing infrastructure and road networks are usually not captured by most models.

Modelling variables

- set V=$\{v_1, v_2, \ldots, v_m\}$ of possible concentrator locations
- set of terminals $U = \{u_1, u_2, \ldots, u_n\}$
- Define the following *binary* variables:

$$x_{ij} = \begin{cases} 1 & \text{if } u_i \text{ is connected to } v_j \\ 0 & \text{otherwise} \end{cases} \qquad y_j = \begin{cases} 1 & \text{if a concentrator is installed at site } v_j \\ 0 & \text{otherwise} \end{cases}$$

Objective Function

The **objective function** to be minimized is the *total cost function*:

$$C = \sum_{j=1}^{m} f_j y_j + \sum_{i=1}^{n} \sum_{j=1}^{m} c_{ij} x_{ij} \tag{1}$$

where:

- c_{ij} is the cost for connecting u_i to v_j,

- f_j is the cost for installing a concentrator at v_j (including the high level cost for connecting v_j to CPU).

Subject to the **constraints** :

$$\sum_{j=1}^{m} x_{ij} = \beta_i \qquad \forall i \tag{2}$$

$$\sum_{i=1}^{n} x_{ij} \leq \bar{\alpha}_j y_j \qquad \forall i \tag{3}$$

$$\tag{4}$$

where:

- β_i is the degree of connectivity at terminal i and is equal to one in the ordinary plant location or the star-star network problem
- $\bar{\alpha}_j$ is the "capacity" of the concentrator that may be installed at v_j.

4.2.2 Medium-Voltage Network Planning Problem

We now consider an example from a non-telecommunications world, namely the *Medium-Voltage Network Planning* (MVNP) problem, which is similar to the concentrator location problem. Fig. 19 shows a typical network; the installed primary stations and substations are shown by solid squares and circles respectively, while the optional or potential ones are represented by dashed lines. Solid lines represent existing cables, while dashed lines are potential electrical cables.

The aim here is to choose the most economical combination of new lines and stations to meet the given demand for electricity, with the optimal dimensions of the equipment being selected. In these types of problem, there could be a temptation to use linear cost functions to represent the cost of components such as the substations and lines, and apply traditional linear programming technique the problem.

In reality, however, the fixed costs of equipment like poles or the cable-trench excavation cannot be included in the cost functions, as these costs are typically non-linear or step functions.

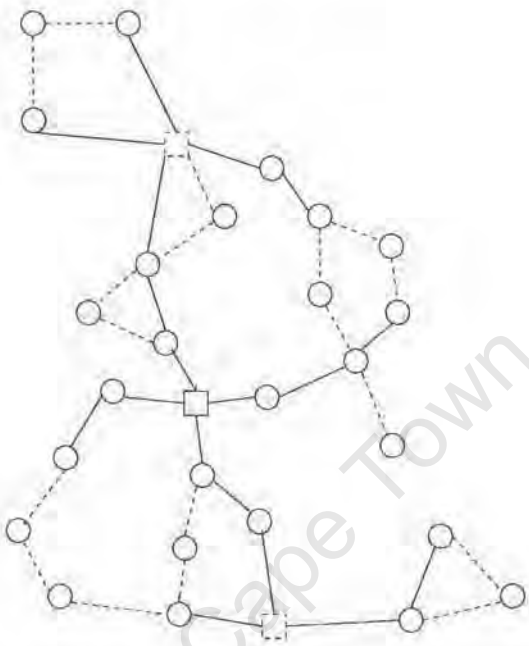

Figure 19: The MV-network horizon-year planning problem

This is particularly evident in the case of optional components where the costs are either incurred or not. This problem is in fact an example of another 0-1 mixed-integer programming problem.

Here, the choice of options in the network can be expressed by using decision variables δ_i and γ_j: δ_i is unity if a substation i is built or zero if it not built, while γ_j is unity if line j is installed or zero otherwise. A set of alternative networks exists for all the possible combinations of δ_i and γ_j (i.e. the solution space). One of these combinations defines the optimal network configuration.

4.3 Solution of Integer Programming Problems

For smaller network sizes, the CPL and MVNP problems, formulated as 0-1 mixed-integer programming problems, can be solved exactly using techniques such as branch-and-bound or implicit enumeration [S69]. As the network size grows however, such approaches become intractable and heuristic algorithms need to be adopted. Several heuristics for solving this problem are discussed in Chapter 5, but here we present a discussion on the solution techniques that can be feasibly applied to smaller integer programming problems, culminating

in a high level description of the *Branch-and-Bound* Algorithm.

One practical approach to solving both pure and mixed integer problems is to ignore the integer restrictions initially and solve the problem as one would a linear program. If the LP optimal solutions results in fractional answers for some of the integer variables, these can be truncated or rounded-off to get an approximate optimal integer solution. For instance, if there are two integer variables x_1 and x_2 that yield fractional values of 2.8 and 4.3, then the four possible integer solutions (3,4), (2,4), (3,5) and (2,5) should be considered.

The true optimal integer solution may in reality not correspond to any of these integer solutions that are yielded by rounding or truncating. It is possible in our example for x_1 to have an optimal value less than 2 or greater than 3. In order to obtain the true optimal solution, *all* the possible integer values of x_1 that are smaller and greater than 3.5 need to be considered.

When the problem contains a large number of integer variables, some systematic method is needed for considering all the possible combinations of integer solutions obtained from the LP optimal solution. When dividing the problem into subproblems, it is obviously desirable that the number of alternatives whose costs need to be compared be effectively reduced. The Branch-and-Bound Algorithm is a method that does this in the most efficient manner. We give an indication of this algorithm by working through a small example in the next section.

4.3.1 Branch-and-Bound Algorithm

The division of the main problem into subproblems forms the basis of the *Branch-and-Bound Algorithm*; a brief description that is intended merely to give a flavour of this approach now follows. This is the method used by the optimization code **lp_solve** (written by Dr Berkelaar from the Eindhoven University of Technology) which is used by ROLAND to obtain exact solutions to the concentrator location problem. A fuller description of the application of the Branch-and-Bound algorithm to this type of problem is discussed in [ER66] and [Sö69].

To illustrate the basic principles of the branch-and-bound method, we consider an example MIP from [PRS76]:

$$
\begin{aligned}
\text{Maximize:} \quad & Z = 3x_1 + 2x_2 \\
\text{Subject to:} \quad & x_1 \leq 2 \\
& x_2 \leq 2 \\
& x_1 + x_2 \leq 3.5 \\
& x_1, x_2 \geq 0 \text{ and integer}
\end{aligned}
$$

Initial Step

The MIP is first solved as a linear program by ignoring the integer restrictions on x_1 and x_2. We call this linear program LP-1, and a graphical representation of a solution is shown in Fig. 20. The optimal solution is $x_1 = 2$, $x_2 = 1.5$, with the maximum value of the objective function z_0 being 9. We do not however have an optimal solution for the MIP problem, as x_2 has a fractional value. It is important to note that the optimal integer solution cannot have an objective function value bigger than 9, as the addition of new constraints to a linear program cannot improve its original optimal value. We have in effect an *upper bound* on the maximum value of Z. This concept is elaborated in the next section where we discuss *Relaxation*.

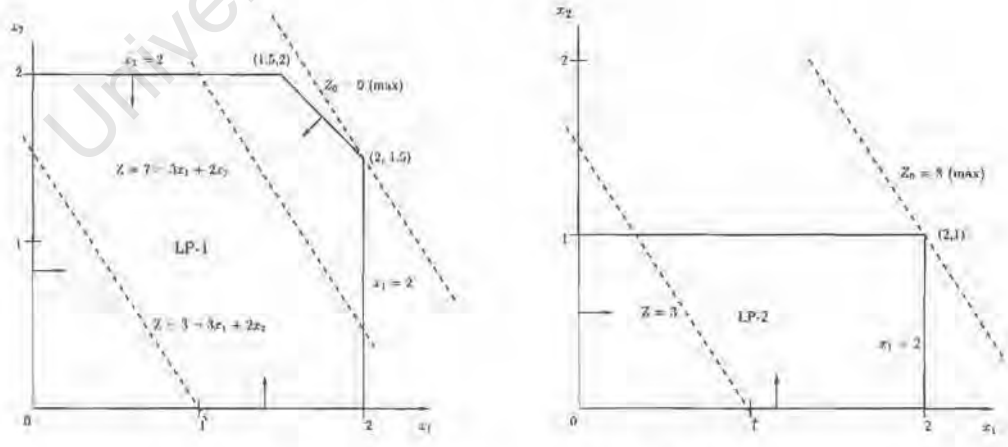

Figure 20: Solutions to LP-1 and LP-2

The next step: branching out

The next step in our example is to examine integer values of x_2 which are larger or smaller than 1.5. We do this by adding a new constraint : either $x_2 \leq 1$ or $x_2 \geq 2$ to LP-1. This results in two new linear programs:

LP – 2	**LP – 3**
Maximise $Z = 3x_1 + 2x_2$	Maximise $Z = 3x_1 + 2x_2$
Subject to : $x_1 \leq 2$	Subject to : $x_1 \leq 2$
$x_2 \leq 2$	$x_2 \leq 2$
$x_1 + x_2 \leq 3.5$	$x_1 + x_2 \leq 3.5$
new constraint: $x_2 \leq 1$	new constraint: $x_2 \geq 2$
$x_1, x_2 \geq 0$	$x_1, x_2 \geq 0$

The feasible regions of LP-2 and LP-3 are shown in Fig. 20 and Fig. 21 respectively (the feasible region of LP-3 is just the straight line AB). These regions satisfy the following:

1. The optimal solution to LP-1 is infeasible to both LP-2 and LP-3 and the old fractional optimal solution will therefore not be repeated.
2. Every feasible integer solution to the original problem is contained in one of LP-2 or LP-3. None of the feasible solutions are lost due to the creation of two new linear programs.

The two new problems have the following solutions:

Optimal Solution to LP-2 (see Fig. 20): x_1 =2 and $x_2 = 1$ ($Z_0 = 8$). There is therefore a feasible (integer) solution to the MIP problem, and even though LP-2 may contain other feasible solutions, these cannot yield objective functions with values smaller than 8. We have a lower bound on the maximum values of Z for the mixed integer program. We computed the upper bound earlier to be 9, so we cannot call LP-2 the optimal solution without first examining LP-3.

Optimal Solution to LP-3 (see Fig. 21): $x_1 = 1.5$ and $x_2 = 2$ ($Z_0 = 8.5$). This solution is not feasible as x_1 takes on a fractional value. We observe also that maximum Z

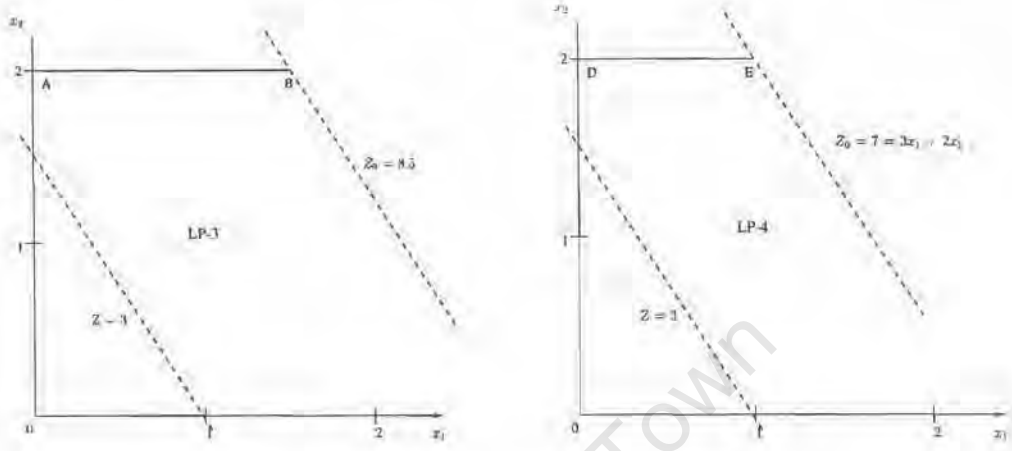

Figure 21: Solutions to LP-3 and LP-4

value (8.5) is larger than the lower bound of eight. We need to therefore check if there is an integer solution in the feasible region of LP-3 whose Z-value is larger than eight. This leads to the creation of two new linear programs LP-4 and LP-5 which have the following solutions:

Optimal Solution to LP-4: $x_1 = 1$ and $x_2 = 2$ ($Z_0 = 7$)(see Fig. 21). The solution if feasible, but less than the lower bound of eight that we previously established.

LP-5 is infeasible: This is clear when one sees that that the conditions $x_1 \geq 2$ and $x_1 \leq 2$ fixes x_1 at 2, while $x_2 \geq 2$ and $x_2 \leq 2$ fixes x_2 at 2. The condition $x_1 + x_2 \leq 3.5$ can therefore not be met.

The integer solution we obtained while solving LP-2, namely x_1 =2, x_2 =1 and $Z_0 = 8$, is the optimal integer solution to the mixed integer problem.

Graphical representation of Branch-and-Bound

The sequence of linear programming problems that results under the branch-and- bound procedure can be represented by a tree structure (usually referred to as a *decision tree*) as shown in Fig. 22. Node 1 in the graph represents LP-1. From node 1, we *branch* to node 2 (which represents LP-2) by adding the constraint $x_2 \leq 1$ to LP-1. Since we obtain an integer optimal solution for LP-2, no further branching from this node is needed; we say that node 2 has been *fathomed*. Branching for $x_2 \geq 2$ from node 1 results in node 3 (representing

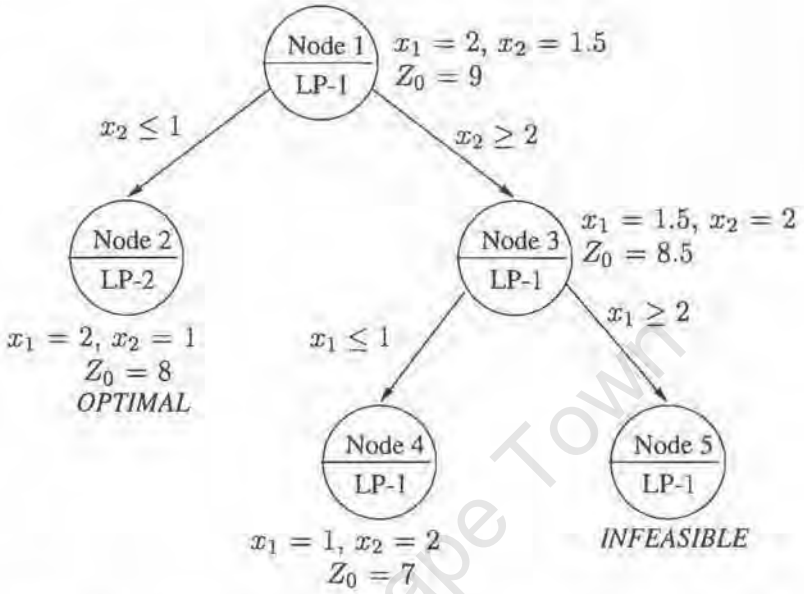

Figure 22: Tree Representation of the Branch-and-Bound Method

LP-3). Since LP-3 yields a fractional optimal solution, we need to branch further from node 3, using the variable x_1. This yields nodes 4 and 5, which are both fathomed since LP-4 has an optimal integer solution, while LP-5 is infeasible. The best integer solution obtained at a fathomed node (in our case, node 2) is chosen as the optimal solution. Fig. 23 shows the decision tree for the electricity network example presented in the previous section, where a node N_k of the tree represents a combination of fixed decision variables δ_i and γ_j.

4.4 Relaxation

Before proceeding with an overview of Relaxation (in the linear programming sense of the word), it is necessary to define a few terms that we shall use in later discussion.

Convexity : An important class of problems is when both the solution space and objective function are *convex*. A solution space is said to be **convex** if for any two points, a and b, inside this space, all the points on the line connecting a and b are also inside the space. In a convex space, we can move from one feasible solution to another in small steps, without leaving the feasible region. This opens up the option of using simple, incremental methods for searching for the optimal solution.

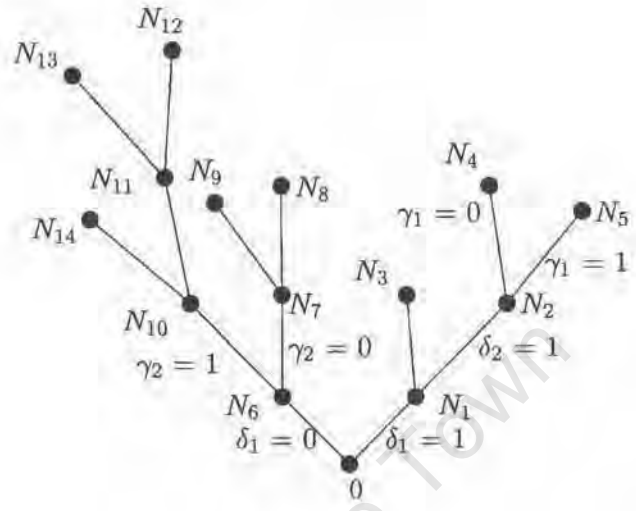

Figure 23: A decision tree for the MV-network horizon-year planning problem

A convex function has the property that for any point c, between a and b, $f(c)$ lies below the line connecting $f(a)$ and $f(b)$. This concept is illustrated in Fig. 24. The notion of convexity extends to functions of many variables, with a line being then interpreted as a multi-dimensional plane. A minimum of a convex function can be found by means of a local search. This means starting at any point and moving in a direction where the function decreases. These methods are known as *descent methods*, and remove the need to search through all the possible values in the solution space.

Concavity : If $f(c)$ lies above the line connecting $f(a)$ and $f(b)$ for all c between a and b, it is said that f is **concave**. For example, the function shown in Fig. 24(b) is concave on the interval from a to b and convex on the interval from b to c. It is neither concave nor convex over the entire interval. The maximum of a concave function can be found in a manner similar to the minimum of a convex function.

It is interesting to note that *linear* functions are in fact both convex and concave. As such, their maxima and minima are relatively easy to find. Integer programming problems, on the other hand, have solution spaces that are not convex, but are isolated points at integer coordinates. None of the points on the line connecting two adjacent integers are in the solution space.

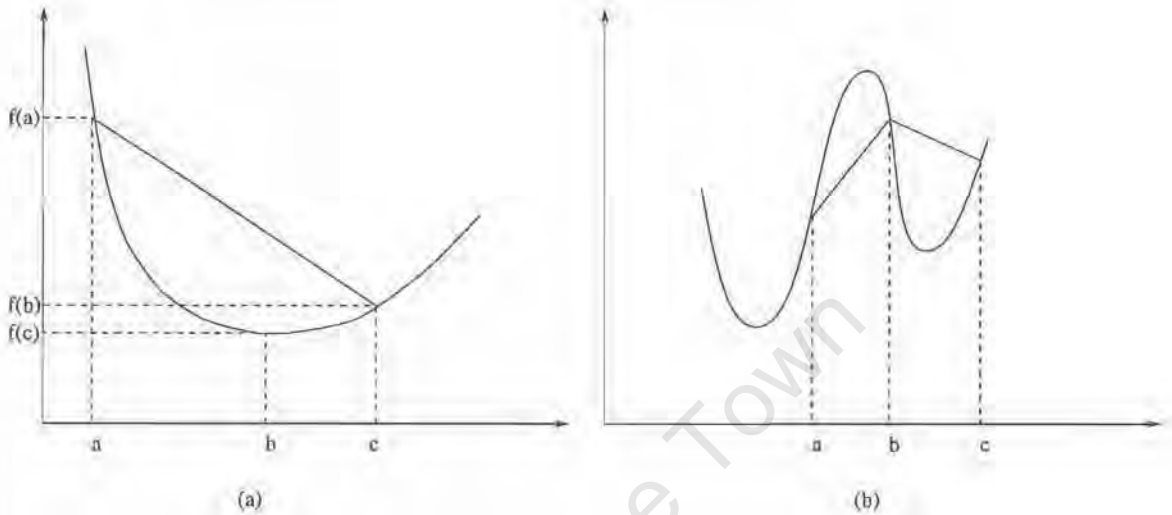

Figure 24: (a) A convex function (b) A non-convex function

Sometimes a problem, P, which is difficult to solve, is closely related to another problem P' which is much easier to solve. The related problem P' may merely be P with some constraints removed, or it may keep the original constraints of P and adopt a more well-behaved (possibly concave) objective function. P' can then be solved in the hopes of obtaining a solution (or good approximation to a solution) to problem P.

A good example of this is the case of the multipoint line problem, where there is a restriction on the number of node permitted in each subtree rooted at the central site. If we relax the problem by lifting the restriction on the number of nodes in each subtree, the problem of finding a set of trees of minimum total length can easily be found using a greedy algorithm. The solution can then be adjusted to satisfy the original constraint.

If we are able to form P' by replacing the original objective function $f(x)$ by a well-behaved function $g(x)$ having the property:

$$g(x) \leq f(x) \qquad \forall x$$

we say that $g(x)$ is a **lower bound** for $f(x)$.

Consider $f(x)$ and $g(x)$ in Fig. 25. Function $f(x)$ is non-convex while function $g(x)$ is a convex function. $g(x)$ is also a lower bound for $f(x)$ on the interval shown. If we try to find the minimum of $f(x)$ using a descent method, we could end up in the valley at x=x_1 or we could find the global minimum at x = x_3.

obtained for the lower bound and feasible solution coincide. Also, if the optimal solution to the relaxed problem can be found, the optimal solution to the actual problem is also found.

In the present case, the optimal solution to the relaxed problem is easy to obtain. Once given the values for p_{ij}, we merely assign each terminal to the closest concentrator, where the "distance" is defined as the sum of c_{ij} and p_{ij}.

Tightening the lower bound poses an interesting problem. By assigning large values to the p_{ij} selected in the solution (provided that their sum for a particular concentrator j does not exceed d_j), the value obtained for the relaxed problem is increased. Making a particular p_{ij} large, however, decreases its chances of being picked, as $c_{ij} + p_{ij}$ is less likely to be smallest for a given terminal i. The penalty assignment algorithm is described in the next section.

Penalty Assignment Algorithm

The problem of assigning the penalties p_{ij} can be thought of informally as "selling" a concentrator to the terminals, starting with all the p_{ij} set to zero. Each terminal is visited and asked how much it is willing to "pay" for concentrator j. If no terminal offers anything, the concentrator won't be selected. A terminal closer to another concentrator, will offer nothing for concentrator j, as it has nothing to gain from it being selected. A terminal i that is closest to concentrator j, however, will pay $c_{ik} - c_{ij}$ where c_{ik} is the cost of connecting i to the next nearest concentrator, k. We can set $p_{ij} = c_{ik} - c_{ij}$, knowing that it will not be wasted and will appear in the final solution.

p_{ij}'s are assigned until all the terminals are forced to pay something for the closest concentrator. It may be possible at this point to increase some of the p_{ij} as $\sum_i p_{ij}$ may not equal d_j for some concentrators. If more p_{ij} is assigned to a concentrator j which is not closest (in terms of $c_{ij} + p_{ij}$) to terminal i, it is a waste as terminal i will then not select it. It is therefore necessary to also assign some penalty to other concentrators in order that $c_{ij} + p_{ij}$ is still tied for the closest distance.

Consider the matrix specified by p_{ij}. It is desirable that only one p_{ij} is assigned per row (i.e. for each terminal), as only one penalty will actually be picked. As few as possible p_{ij} per row should therefore be assigned. On the other hand, we can assign a total of d_j penalties for each column j (corresponding to the cost of concentrator j). In other words, a tradeoff exists, and it is sometimes beneficial to assign several p_{ij}'s in each row, in order to

prevent some of the penalty going to waste. Kershenbaum adopts the following procedure: first assign one p_{ij} per row, then two, then three and so on.

The following example from [Ker93] illustrates the procedure. The matrix below gives the costs of connecting four terminals to four concentrators. We assume that the cost of each concentrator is 5. In general, the number of terminals versus the number concentrators, as well the cost of each concentrator, differs.

Original values :

		Conc				
		1	2	3	4	Value
t	1	5	9	2	6	2
e	2	3	1	4	1	1
r	3	5	8	9	7	5
m	4	9	3	2	4	2
		5	5	5	5	10

The procedure begins by setting each terminal's `Value` to be equal to the smallest value in each row (which specifies the cost of connecting each terminal to its nearest concentrator). The sum of the `Values` (10 here) is certainly a lower bound on the cost of the solution.

We then try to assign a single p_{ij} in each row:

- Set $p_{13} = 3$, making `Value` = 5 for terminal 2 and `spare` 2 for concentrator 3.
- in the second row, there is a tie for the best p_{ij}, so this would involve a double adjustment which we are not allowed to make yet. We move straight to the third row.
- Set $p_{31} = 2$, `Value` = 9 in row 3 and `spare` = 5 for concentrator 1.
- Set $p_{43} = 1$, `Value` = 3 in row 4 and decrease `spare` to 1 for concentrator 1.

This results in the lower bound being increased to 16.

After the first iteration (level = 1):

		Conc				
		1	2	3	4	Value
t	1	5	9	5	6	5
e	2	3	1	4	1	1
r	3	7	8	9	7	7
m	4	9	3	3	4	3
		3	5	1	5	16

We now consider assigning *two* p_{ij} per row.

- In row one, increase p_{11} and p_{13} by 1; `Value` increases to 6 and `spare` decreases to 2 and 0 for concentrators 1 and 3 respectively.
- Row 2 and 3, `Value` is increased to 3 and 8 respectively.
- `Value` is row 4 cannot be increased, as this requires `spare` from concentrator 3, which has already exhausted its `spare`.

The lower bound is now increased to 20.

After the second iteration (level = 2):

		Conc				
		1	2	3	4	Value
t	1	6	9	6	6	6
e	2	3	3	4	3	3
r	3	8	8	9	8	8
m	4	9	3	3	4	3
		1	3	0	2	20

We now consider assigning three p_{ij} in a row:

- Row 1 needs `spare` from concentrator 3 (which has been exhausted), so its `value` cannot be increased.

- In row 2, the last spare unit can be used to push the value up to 4.
- After this, no further changes can be made.

The lower bound now 21, as shown in the table below.

		Conc				
		1	2	3	4	Value
t	1	6	9	6	6	6
e	2	4	4	4	4	4
r	3	8	8	9	8	8
m	4	9	3	3	4	3
		0	2	0	1	21

In this example, concentrators 1 and 3 have in effect "sold" themselves by exhausting all their spare. If we select both these concentrators and assign each terminal to the closer concentrator, terminals 2 and 3 are assigned to concentrator 1, while terminals 1 and 4 are assigned to concentrator 3. This yields a total cost of 22.

The lower bound is therefore fairly tight, and we are *assured* that the solution is within 1 of being optimal. It is possible to consider all the subsets of concentrators with no remaining spare, but this is an exponential procedure, and while feasible for small numbers of concentrators, some form of heuristic is normally adopted in practice. ROLAND uses the Drop Algorithm (described in Chapter 5) to make a selection from the set of concentrators that have no spare remaining.

The linear programming approach is only feasible for smaller network sizes, and heuristics are needed for larger networks. Heuristics for solving access network design problems are described in the next two chapters.

Chapter 5

Concentrator Location Algorithms

The linear programming solution of network design problems is only feasible for smaller network sizes, and some form of heuristic is usually needed to achieve solutions to larger network problems. This chapter describes in greater detail four heuristics that help to determine good locations for the installation of equipment such as concentrators in the access network. Short descriptions of these were given in the second chapter; here we expand on these descriptions and give the implementation details of the Centre-of-Mass, Add and Drop heuristics. A new hybrid algorithm known as the Cluster-Add Heuristic is also presented.

5.1 The Centre-of-Mass (COM) Algorithm

5.1.1 Algorithm Description

The centre-of-mass algorithm identifies natural clusters of traffic, and is a simple yet computationally efficient method of locating concentrators, especially in the case where potential concentrator sites have yet to be identified.

Suppose each terminal, i, has coordinates (x_i, y_i) and traffic-based weights, w_i.

We begin with each terminal in a cluster by itself. Clusters close to each other are then merged replacing two clusters, i and j with new cluster k, which is located at the centre-of-mass (COM) of i and j:

$$x_k = \frac{w_i x_i + w_j x_j}{w_i + w_j}$$

$$y_k = \frac{w_i y_i + w_j y_j}{w_i + w_j}$$

The clusters i and j are then removed from the set of clusters, and cluster k is added in their place. The following restrictions also apply:

- There exists a desired weight, W, for a cluster, which should not be exceeded. There *may* also exist a minimum weight for a cluster.
- There is a limit, D, on the distance separating two clusters that may be merged. This distance is defined to be the distance between their centres-of-mass.
- There exists a desired number of clusters, N. Merging should cease once this limit is reached.

These constraints are in reality contradictory and compromises may need to be made. When deciding on what clusters to merge, the intuitive approach is to continually merge the two clusters that are closest together. In the next section, we present an implementation of the COM Algorithm that adopts this approach.

5.1.2 Implementation

The following data structures and variables are used in Kershenbaum's implementation of the COM Algorithm [Ker93]:

- an array `cost` represents the cost of connecting each pair of terminals. The Euclidean distance between the two points is used for this purpose, but a more accurate function (e.g. some step function) could be used for this purpose.
- each terminal has a weight based on the traffic to and from it, as well as an x and y coordinate.

- a variable `nClus` is the limit on the number of clusters. The algorithm stops if it succeeds in forming `nClus` clusters.
- A cluster with a weight above `Wlimit` is restricted in this implementation from being formed.
- `Cfac` controls the maximum cost (distance) between clusters to be merged
- The algorithm computes `Cmax`: the maximum distance (cost) between terminals
- `Cthresh`, the threshold on cost allowed between two merging clusters, is then set. `Cthresh` is equal to `Cmax` times `Cfac`.
- the algorithm outputs a vector, `Cassoc`, listing the cluster associated with each terminal.

The algorithm sets up a heap of neighbours of each node. This assists when finding the next nearest neighbour of each node during processing. A heap `SHeap` is initialized to contain the current nearest neighbour of each node. The values in this heap are held in a vector `sel_val` and are the cost to the nearest neighbours of each node.

As the algorithm proceeds, neighbours are considered and popped off the individual nodes' heaps. `SHeap` is updated to reflect this. As nodes (then clusters) are merged to form clusters, one node ceases to exist, and the other becomes the head of the merged cluster. `cNum[t]` helps to find the cluster number associated with node t. It holds the number of another node in the same cluster. The actual cluster number can be found by following the chain of `cNum`'s until the cluster head is found (i.e. until `cNum[h]` $=$ `h`). It is possible to shorten these chains by collapsing them, which we shall see shortly. Nodes no longer existing pop to the top of the neighbour heaps and `sHeap`, where they are recognized by the fact that `cNum[c]` ≤ 0 and removed from further consideration.

The current closest pair of nodes are tested to see if they can be merged and if it found feasible (i.e. the nodes involved both exist and distance and weight constraints are not violated), the merge proceeds and a new cluster is formed at the centre-of-mass of the merging clusters. Pseudocode for the main COM Algorithm and the procedures used to access and perform merges can be found in Appendix A.

It should be noted that if the clustering produced by this algorithm is to be kept, another algorithm would be run to pick the actual centres of each cluster. This is due to the fact that the algorithm blindly positions new clusters at the centres-of-mass of groups of terminals, which in general might not lie at a feasible geographical location.

Tarjan's Method

When keeping track of what cluster a terminal belongs to, Tarjan [KT81] suggests keeping a pointer to another node in the same cluster, and have one node designated as the head of the cluster (pointing to itself). Initially, every node is in a cluster by itself and consequently points to itself. When a terminal i is merged with a cluster j, i is made to point to j. As the algorithm progresses, and cluster i (consisting of more than one terminal) with head j, and cluster k with head l, need to be merged, k is made to point at l. The merge thus involves following two chains of pointers until the heads of both clusters are found. The shorter the chain of pointers, the less work this involves. Tarjan presents a way of collapsing the chains *as they are traversed*, thereby shortening them. He suggests that a function, `FindComponent`, be implemented as follows:

```
index <- FindComponent( node, *next )
   int next[]

   p = next[node];
   q = next[p]
   while ( p != q )
      next[node] = q
      node = q
      p = next[node]
      q = next[p]
   return (p)
```

Tarjan showed that by collapsing the chain by one link (by adjusting `next` to point one level higher, rather than collapsing it completely or not at all, the overall complexity of testing

and updating `next` is $O(n+m)$ where n is the number of nodes and m is the number of links tested.

5.1.3 Examples

This 10 terminal example is taken from [Ker93](see Fig. 26), and for simplicity, all terminals have been given a weight of 1 (which is not required by the algorithm). The desired number of clusters is 2 and there is a weight limit of 4, with a cost factor of 0.5. The longest link (6,1) has a cost of 87, which results in a cost threshold of 43.

Node	X	Y	Weight
1	31	19	1
2	45	13	1
3	59	92	1
4	22	64	1
5	86	55	1
6	95	78	1
7	98	63	1
8	39	44	1
9	27	38	1
10	48	85	1

Weight limit = 4 Cost factor = 0.5 No. of clusters = 3

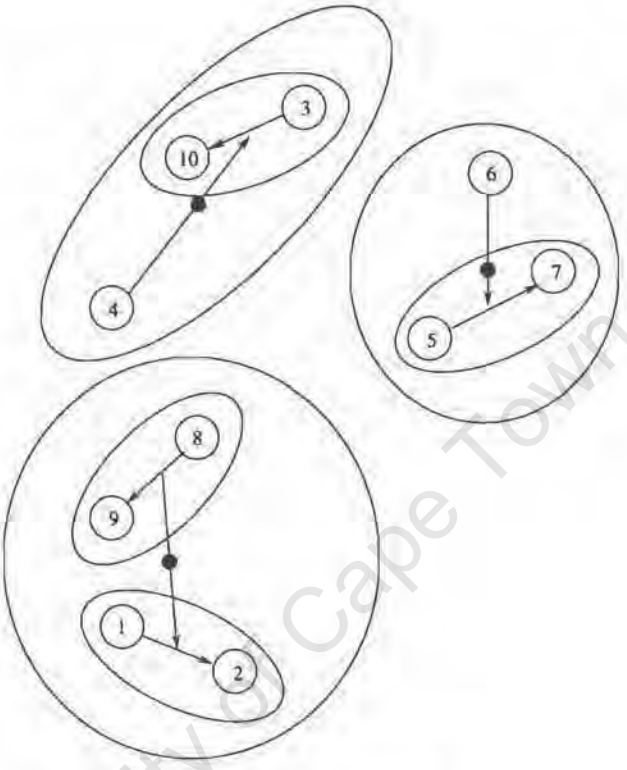

Figure 26: COM example

Merges considered:

Node	Neighbour	Cost	X	Y	Weight
9	8	13	33	41	2
10	3	13	53	88	2
7	5	14	92	59	2
2	1	15	38	16	2
7	6	19	93	65	3
9	1	22	-	-	-
6	-	24	-	-	-
2	9	25	35	28	4
9	-	25	-	-	-
8	-	26	-	-	-
2	4	38	-	-	-
10	4	39	42	80	3

The algorithm first considers merging nodes 9 and 8 (the distance between these two nodes is 13). This merge does not violate any of the constraints and the two nodes are merged. The next four merges are also accepted. The merge (9,1) is not accepted, however, due to the fact that node 1 is no longer independent and part of a cluster. The (2,4) merge is rejected due to the fact that this would violate the weight restriction (forming a cluster of weight 5). The final clusters are shown in Fig. 26.

To further illustrate how the COM Algorithm can be used to identify varying numbers of clusters, consider the sample network shown in Fig. 27. The clusters produced for varying values of `nClus` are shown in Fig. 28 to Fig. 30

Figure 27: Test network for the COM Algorithm

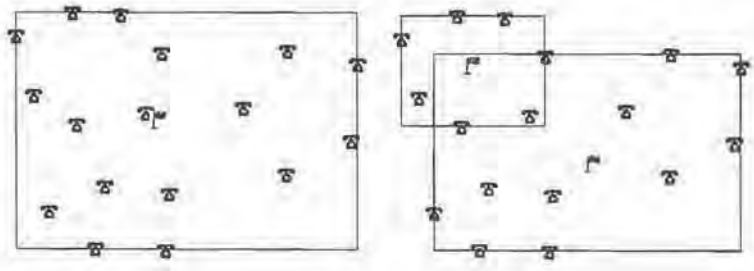

Figure 28: COM Algorithm's formation of 1 and 2 clusters

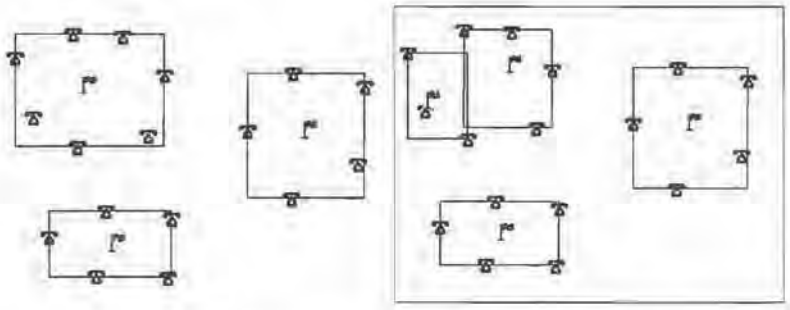

Figure 29: COM Algorithm's formation of 3 and 4 clusters

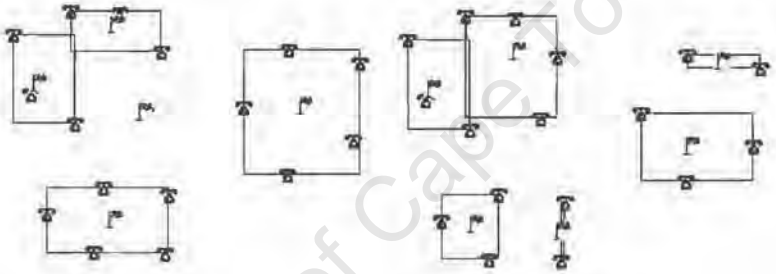

Figure 30: COM Algorithm's formation of 5 and 6 clusters

5.2 The Add Heuristic

5.2.1 Algorithm Description

The Add Heuristic is a greedy algorithm. It starts with all the concentrators connected directly to the central node (switch) and then evaluates the savings (reduction in cost) that can be made by placing a concentrator at each potential site. The concentrator which offers the most savings is then selected.

The central switch is regarded by the algorithm as a concentrator with infinite capacity located at the origin of the coordinate system. We then have the following costs:

- c_{ij} : the cost of connecting distribution point i to concentrator j for $j > 0$.
- c_{i0} : the cost of connecting distribution point i to the central switch.

Assuming that there are no capacity constraints on the concentrators, the savings s_j associated with each concentrator j are:

$$s_j = \sum_{i \in I(j)} (c_{ij} - c'_i) - d_j$$

where:

- d_j is the cost of installing a concentrator at site j
- c'_i is the cost of connecting the section of the network containing i to the central switch. Initially, all the terminals are associated with the centre, so this cost is c_{i0}, but as terminals become associated with concentrators, this value becomes the cost of connecting the concentrator associated with i to the switch.

I(j) is the set of all the terminals that can achieve savings by moving to j, i.e. it costs less to be associated with j than their current concentrator. At each step of the algorithm, the concentrator which saves the most money is chosen and the most cost-efficient terminals are associated with it. The algorithm proceeds by greedily selecting the best concentrators, until no concentrator that achieves any savings can be found. This is achieved by creating a heap of concentrators, with the concentrator providing the biggest savings at the top of the heap.

When implementing the Add Heuristic, it must be decided whether to allow terminals to move from one concentrator to another as new concentrators are installed. If terminals are allowed to move, the case may arise where a concentrator that was selected earlier is no longer justified, as few or no terminals are now associated with it. Kuehn and Hamburger, who introduced this algorithm in 1962 in the context of determining locations of warehouses in large-scale distribution networks, first encountered this problem. They used a *Bump and Shift Routine* to eliminate (bump) any warehouse which is no longer economical because some of the customers originally assigned to it are now serviced by warehouses located subsequently [KH63].

If we choose not to allow terminals to move, the case may arise where a particular concentrator chosen early on captures terminals that are closer to another concentrator not yet chosen. One solution is to run a terminal-assignment algorithm after the Add Heuristic has finished, but this suffers from the flaw that certain concentrators may not be selected by

the Add Heuristic, due to the fact that they do not lie close enough to terminals that are still free.

In practice, it has been found that if the capacity constraints are fairly loose, it is better to allow terminals the freedom to move [WWB78]. In this case, the concentrators selected early on tend to grab too many terminals, but can still justify their selection even if some terminals subsequently move. On the other hand, if the capacity constraints are tight, then the movement of terminals should be prohibited or at least discouraged. This can be done by associating a penalty with a move, so the savings of moving a terminal i from a concentrator j to another concentrator j would be calculated as $s_{ij} = c_{ij} - c_{ik} - P$, where P is some penalty associated with the move.

5.2.2 Algorithm Complexity

After a concentrator is selected, the c'_i change, and the savings of the uninstalled concentrators have to be recalculated. This means that each terminal has to be examined and selecting the concentrator with the best savings means that we have to compare each s_j. Selecting a concentrator is therefore $O(TC)$ where T is the number of terminals and C is the number of potential concentrator sites. As we can select at most C concentrators, the overall complexity of this algorithm is $O(TC^2)$.

5.3 The Drop Heuristic

5.3.1 Algorithm Description

An alternative greedy approach to the concentrator location problem is to start with *all* the potential concentrator sites selected and drop the concentrators that result in the biggest financial savings. The algorithm is very similar to the Add Heuristic, and at each stage the savings that can be made by dropping each concentrator are evaluated, and the one that saves most is dropped. As in the Add Heuristic, the savings made at each stage decrease as the algorithm progresses, until the algorithm terminates when no more savings can be made.

The implementation of the algorithm presented by Kershenbaum is similar to the Add Heuristic, but does have some differences. Once again, a heap of concentrators is made,

based on each concentrator's savings. The main differences arise in the *capacitated* case where evaluating each concentrator requires a *terminal assignment algorithm* to be run. An optimal terminal assignment algorithm is described in the next chapter. The algorithm starts by solving a terminal assignment problem with all the concentrators present, and then for each concentrator, a terminal assignment problem for the network minus that concentrator is solved, in order to determine which concentrator should be dropped.

5.3.2 Algorithm Complexity

A heuristic approach to the terminal assignment problem is $O(TC)$ for T terminals and C concentrators. The entire algorithm works out to be $O(TC^3)$, which is only computationally viable for small networks. It makes sense, there to limit the Drop Heuristic to the *uncapacitated* case. Since the Add and Drop heuristics yield comparable results in terms of quality of solution, it does not make sense to use the Drop Heuristic in the capacitated case, due to its added complexity.

5.4 The Cluster-Add Heuristic

The author has developed a new hybrid algorithm which combines the COM and Add heuristics to form an algorithm which first installs concentrators located near "centres-of-mass". This decreases the number of iterations needed by the classical Add Heuristic. In Chapter 8, we look at how the performance of this algorithm compares with the other concentrator location algorithms.

5.4.1 Algorithm Overview

Fig. 31 shows the basic principles behind the algorithm: concentrator sites situated a distance less than δ (a given parameter chosen according to the particular context) from a centre-of-mass are installed first before calling the Add Heuristic with a diminished number of potential sites. The terminals belonging to a centre-of-mass located near a concentrator site are assigned to that concentrator, provided that they are sufficiently close to that site. In the figure, site s_2 is situated sufficiently close to a centre-of-mass, so it is installed, and three terminals to its left belonging to that cluster are assigned to it. The three circled

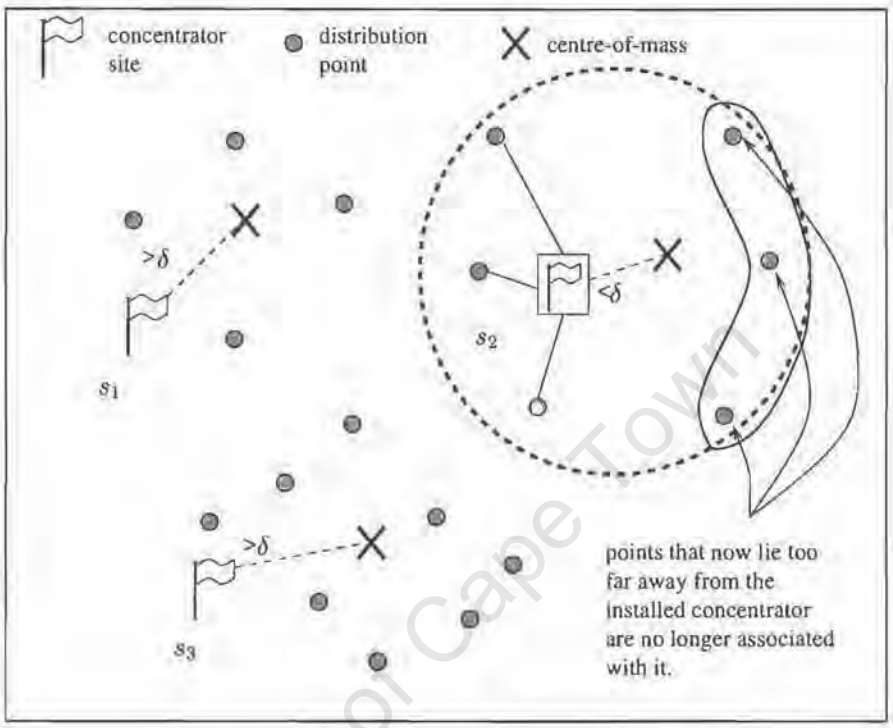

Figure 31: The Cluster-Add Heuristic

terminals to its right are *not* associated with this concentrator, as their distance from the site is too great.

5.4.2 Motivation for Algorithm

Concentrator location algorithms such as the Add and Drop heuristics are provided with lists of potential concentrator sites that are considered for installation in the network. These potential network sites still need to be determined by the network designer. One algorithm that can be used for this purpose is the COM Algorithm, but as we have seen the centres-of-mass that are revealed are not guaranteed to fall on physically feasible locations. While the Cluster-Add Heuristic does not "suggest" potential sites, it does recognize the fact that sites are favourably situated when located near the middle of clusters of demand.

The time sacrificed for the initial call of the COM Algorithm, is redeemed to a degree by a reduction in the number of sites that need to be considered by the Add (or another) heuristic. Fixing nearby terminals to these "demand- centred" concentrators, reduces the number of

terminals that are involved in the terminal-assignment phases of the Add Heuristic. The degree to which the number of sites is cut down is dependent on the degree of clustering in the locations of the distributions points, as well as the parameters of the COM and Add heuristics.

Given the location of points of demand in a network, an initial investigation is likely to involve the use of some algorithm such as COM to determine the situation of clusters of demand. The Cluster-Add Heuristic still displays the location of centres-of-mass in the network, and offers the added benefit of being able to determine whether any existing concentrators are located near centres of demand.

5.4.3 Implementation

We now examine the implementation of the Cluster-Add Heuristic that is included in ROLAND, giving the pseudocode representations of the primary procedures.

Algorithm Parameters

There are three parameters associated with the Cluster-Add Heuristic:

- δ : concentrator sites closer than δ to a centre- of-mass are installed
- *term_dist* : terminals associated with a centre-of-mass are only associated with a nearby concentrator if they lie within this distance of the concentrator.
- *min_surplus* : a large cluster of distribution points may need to be serviced by more than one concentrator. If a concentrator is already installed near a centre-of-mass and additional demand greater than `min_surplus` still exists, additional concentrators may be installed, provided they lie within δ of the centre-of-mass.

The Cluster-Add Heuristic in effect combines the COM and Add heuristics, and as such is also reliant on these two algorithms' parameters. Chapter 8 looks at the effect the choice of these parameters has on the cost of the solution produced.

COM_Install Procedure

This procedure is called before invoking the Add Heuristic (or any other concentrator location algorithm) with the reduced concentrator site list. It is excecuted after the COM Algorithm has just been run and recorded the clusters that it has formed in the array `Cassoc[]`.

```
Procedure COM_install
Accepts:
   num_term,                          // number of terminals
   num_site,                          // the number of concentrator sites
   cost[num_term][num_term],          // the cost matrix
   weight[num_term],                  // the demand of each terminal
   x[num_term], y[num_term],          // the coordinates of each terminal
   installed[num_COMs]                // records which COMs have been serviced
begin
   for each potential concentrator site, s
   begin
      int com = check_dist(s);
      if (com != -1 )
      begin
         if (installed[com]){
            //  check whether demand for another concentrator exists

            if (surplus[i] > MIN_SURPLUS)
              install_conc(i, com, installed)
         }
         else install_conc(i, com, installed)
       end
    end
end
```

Check Distance Procedure

This procedure tests to see whether a given concentrator is sufficiently close to any centre-of-mass identified by the Centre-of-Mass Algorithm.

```
Procedure check_dist
Accepts:
   c          // index of the concentrator site to be evaluated
   COM[]      // array of centres-of-mass suggested by COM Algorithm
Returns:
   Integer    // index of a nearby centre-of-mass
              // if this site is not near a COM then it returns -1
begin
   int COMindex := -1
   // we loop through all the suggested COMs
   for each centre-of-mass, i
   begin
      int distance = Euclidean (c.position, COM[i].position)
      if (distance < DELTA)
      begin
         if (COMindex != -1) // another close COM has already been found
            // check whether this COM is closer or not
            if ( Euclidean (c.position, COM[COMindex]) > distance){
               COMindex := i
            // otherwise the previously found COM is closer
         else COMindex := i
      end
   return COMindex
end
```

Install_Conc Procedure

This procedure simply indicates that the specified centre-of-mass has been "serviced" by a concentrator site and calls connect_terminals to associate nearby terminals with the newly installed concentrator site.

```
Procedure install_conc
Accepts:
   conc            // index of concentrator to install
   COMindex        // index of nearby centre-of-mass
   installed[]     // array of flags indicating which centres-of-mass have
Returns:
   void
begin
   installed[COMindex] := true
   connect_terminals(conc, COMindex)
end
```

Connect_Terminals Procedure

This procedure is responsible for associating terminals belonging to a cluster with a newly installed concentrator that is situated close to the centre-of-mass of the cluster. Recall that an array Cassoc[nt] holds the indices of the clusters associated with each terminal.

```
Procedure connect_terminals
Accepts:
   conc         // index of a concentrator that has just been installed
   COMindex     // index of nearby centre-of-mass (suggested by COM algorithm)
Returns:
   void
begin
   int termindex[nt]   //  indices of terminals asssociated with COM
   int num_ cluster_terms := 0 // number of terminals associated with concentrator
   int conc_term_dist[nt] // distances between conc and the  associated terminals
```

```
for each terminal, t
begin
   if (Cassoc[t] == COMindex)
   begin
      termindex[cluster_terms] := t
      conc_term_dist[cluster_terms] = Euclidean (t.position, conc.position)
      num_cluster_terms++
   end
end
// Now we can begin associating terminals with the concentrator
if (cluster_terms < Ccap[conc]) // Ccap holds the capacity of each conc.
// we can connect all the terminals to the concentrator
begin
   for ct := 1 to num_cluster_terms
   begin
      if (conc_term_dist[ct] < TERM_DIST)
      make_connection(ct, conc);
   end
end
else // we associate only the closest terminals with conc
begin
   // set up a heap of terminals based on their distances to the conc
   heap tHeap.init(num_cluster_terms, conc_term_dist)
   int num_conncected := 0
   while (num_connected < Ccap[conc])
   begin
      int closest_term := tHeap.pop()
      if (conc_term_dist[closest_term] < TERM_DIST)
         make_connection(closest_term, conc)
      num_connected++
   end
end
end
```

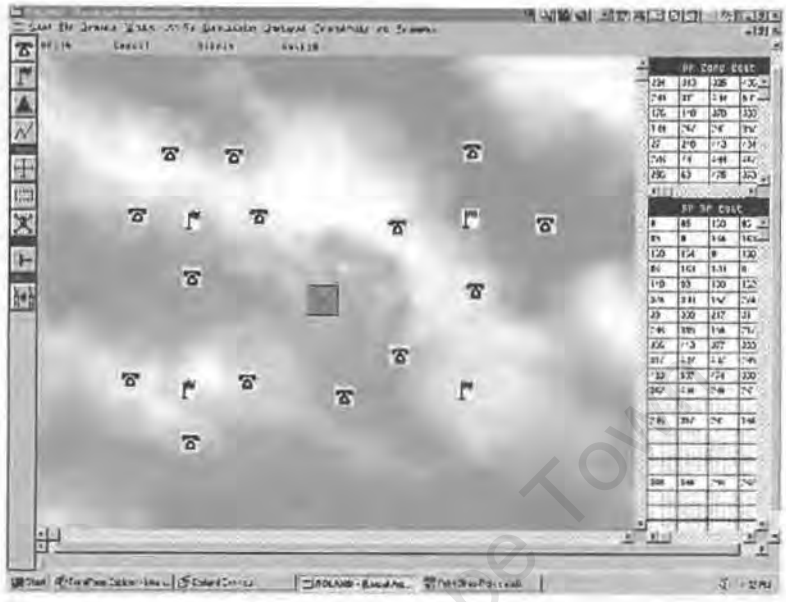

Figure 32: Test Network for Concentrator Location Algorithms

5.5 Comparison of Algorithms on Test Network

To illustrate the operation of the Add, Drop and Cluster-Add heuristics in ROLAND's environment, we shall use the example network shown in Fig. 32. The network has 14 distribution points and four potential concentrator sites.

Add Heuristic: with a concentrator capacity of four and transfer penalty of zero, the solution shown in Fig. 33 was yielded. The cost of the resulting network is 1488.

Drop Heuristic: the network shown in Fig. 34 was produced. The cost of the network is 1514, as opposed to the lower solution cost of 1488 produced by the Add Heuristic. Remember that an implementation of the Drop Heuristic for the *uncapacitated* case has been used. Once again, only three of the four concentrator sites have been selected for installation.

Cluster-Add Heuristic: the resulting network is shown in Fig. 35. The cost of the solution is 1510 (compared with the 1488 produced by the Add Heuristic and the 1514 produced by the Drop Heuristic).

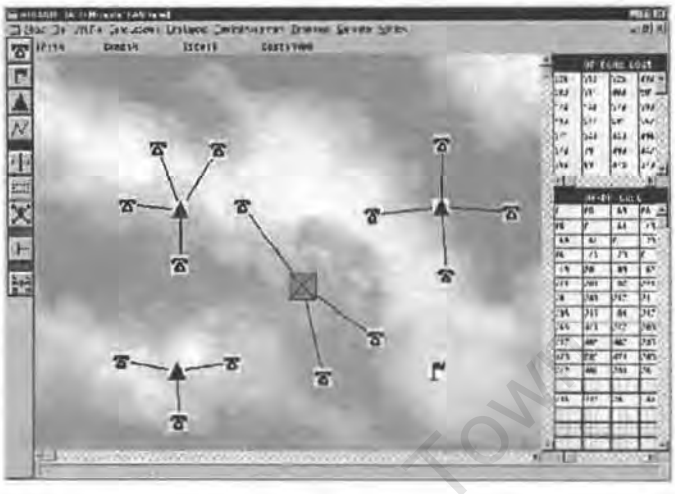

Figure 33: Solution produced by Add Heuristic

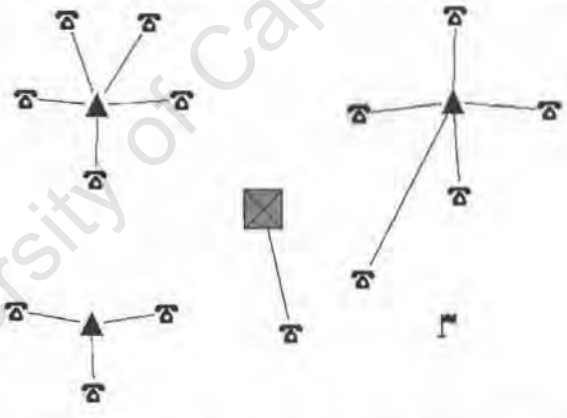

Figure 34: Solution produced by the Drop Heuristic

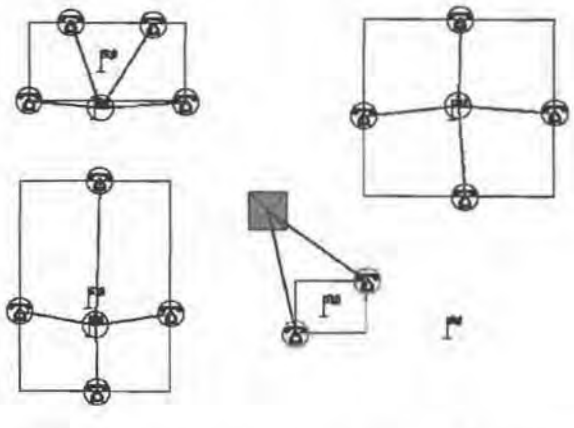

Figure 35: Solution produced by Cluster-Add Heuristic

Chapter 6

Line Layout and Terminal Assignment Algorithms

In Chapter 2, we introduced the problem of designing minimum-cost multidrop lines which connect remote terminals to a concentrator. Optimal solutions can be obtained by using linear integer programming and methods such as branch-and-bound (see Chapter 3). As in the case of the concentrator location problem, these approaches are not feasible for most practical problems, and a need for some form of heuristic approach arises. This chapter presents the implementation details of two line layout heuristics, and a third algorithm which unifies these two heuristics is also described. It concludes by looking at the implementation of an optimal *terminal-assignment algorithm* which determines the best way to connect a set of terminals (such as distribution points) to a set of concentrators.

6.1 Line Layout Algorithms

6.1.1 Kruskal's Constrained Minimum Spanning Tree Algorithm

The problem of connecting a set of distribution points to a set of concentrators is known as the *Constrained Minimal Spanning Tree Problem.* This is an extension of the classical Minimal Spanning Tree Problem (MST), with an additional constraint on the size of the subtrees connected to the root. In this context, the root of the tree represents the local exchange. Nodes in the tree connected to the root represent concentrators, and the subtrees

(rooted at these nodes) constitute the networks connecting distribution points to these concentrators (see Fig. 36). CMST can not be solved exactly using linear programming. It has been shown to be *NP*-complete [GJ79] and while algorithms do exist for solving smaller size problems exactly [Gav85], heuristics need to be employed for larger problems. It is these heuristics that we focus on in this section.

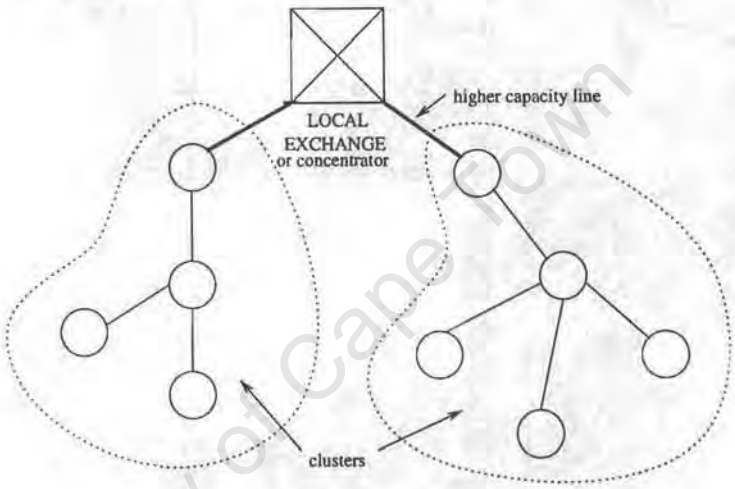

Figure 36: Multipoint lines

Simple and efficient algorithms like Kruskal's greedy algorithm exist for solving MST. These algorithms can be extended to solve CMST by adding a parameter restricting the subtree size. A description of Kershenbaum's implementation of this extension to Kruskal follows. In the description, no graph representation is assumed.

The algorithm starts by initializing some structure (called `link_struc`) that holds the candidate edges, and a heap `link_heap` is initialized with these edges to enable us to repeatedly select the next best candidate edge.

A structure `comp_struc` that keeps track of what component each node is in is also initialized. The procedure `FindComponent` is used to determine which component a node belongs to. A similar approach to that used by the COM Algorithm presented in the previous chapter can be adopted. This structure should also keep track of the total weight of each component, as this information is required when determining whether it is legal to merge two components or not. A parameter `Wmax` specifies the maximum component weight, and the `TestMerge` procedure is used for this purpose.

The algorithm proceeds by popping links off `link_heap` and testing whether this link connects nodes in different components. If this is the case, and the components can be merged without violating the restriction on the component size, then the link is included in the solution, and the two components are merged. The procedure `Merge` accomplishes this.

Pseudocode for the various procedures is provided in Appendix B.

Worked Example

The example shown in Fig. 37 is taken from [Ker93]. It is assumed that each node has a weight of 1 and `Wmax` has been set to 3. The algorithm greedily selects the links (1,3), (1,2) and (0,1). The links (2,4) and (3,5) are then rejected, as this would result in a violation of the capacity constraint of 3. (4,5) and (4,0) are then added to complete the network, resulting in a final cost of 41. The greedy algorithm does not yield the optimal solution and a lower cost feasible solution does exist. For example, (0,1), (1,3), (0,2),(2,4),(4,5) represents a solution with a cost of 36.

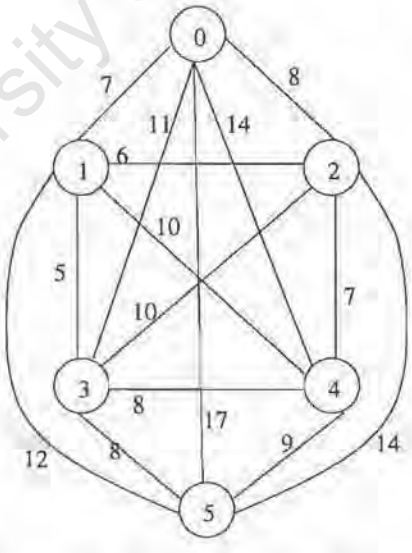

Figure 37: CMST Problem Example

6.1.2 The Esau-Williams Algorithm

The main drawback of the greedy approach to solving CMST is the fact that in choosing the least expensive link at each stage in the algorithm, nodes far from the local exchange

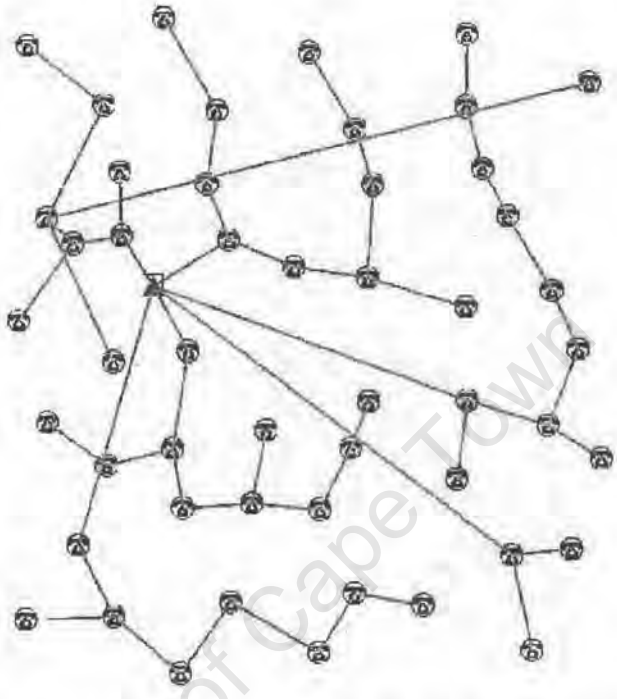

Figure 38: Result of applying Kruskal to 50 node test network

can be left stranded, resulting in the inclusion of expensive links in the solution. On an example network of 50 nodes (constructed using ROLAND), with a weight limit of 10 on the cluster size, Kruskal yielded the network shown in Fig. 38, with a cost of 6535. Note the presence of three long links stretching from the right side of the network to the concentrator that is represented by the triangle.

One way of combating this situation, is by associating a tradeoff function, t_{ij}, with each link. This function enables the algorithm to pay more attention to nodes far from the centre, giving preference to the inclusion of links incident to these nodes. This function is defined as:

$$t_{ij} = c_{ij} - c_{c_i}$$

where c_{ij} is the cost of connecting node i to node j and c_{c_i} is the cost of connecting the component containing node i to the centre. c_{c_i} is initially the cost of connecting node i directly to the centre, but as i is joined to a component containing other nodes, c_{c_i} changes and is specified by:

$$c_{c_i} = \min_{k \in c_i} c_{kC}$$

where c_{kC} is the cost of connecting k directly to the centre. The algorithm starts with all the nodes connected directly to the centre, and then examines whether it is possible to lower the network cost by exchanging direct links to the centre with cheaper links between nodes, thereby connecting nodes with each other. When no more cost-saving exchanges can be found, the process terminates.

Esau-Williams uses the same `TestComponent` procedure as that used by Kruskal's greedy algorithm. The `Merge` procedure is extended to keep track of the cost of connecting the resulting component to the local exchange. The pseudocode for these two procedures, as well as the main algorithm, is given in Appendix B.

Returning to the problem encountered by Kruskal in Fig. 38, applying Esau-Williams to the same network results in fewer "long" links being included in the solution resulting in a more economical network, with a cost of 6021 as opposed to 6535.

6.1.3 Unified Algorithm

Kershenbaum and Chou have developed a *unified algorithm* that embodies several algorithms by generalizing the concept of a component weight [KC74]. These algorithms include Kruskal and Esau-Williams, as well as Vogel's Approximation Method (VAM) and Sharma's Algorithm (introduced in Chapter 2). It has shown to be very effective in that the designs obtained are always as good, if not better, than the individual algorithms applied separately. In addition, hybrid algorithms can be created by adjusting the algorithm's parameters.

The Unified Algorithm is a modified form of Kruskal's (and hence Esau-William's) algorithm, which uses generalized data structures and functions. By setting a "toggle" in the algorithm code, different heuristics, which use these generic structures, can be applied. In addition to the use of a structure to keep track of each component being formed (weight, size, member nodes, etc.) and the heap used to find the best neighbour for each node, the following generalized functions are used:

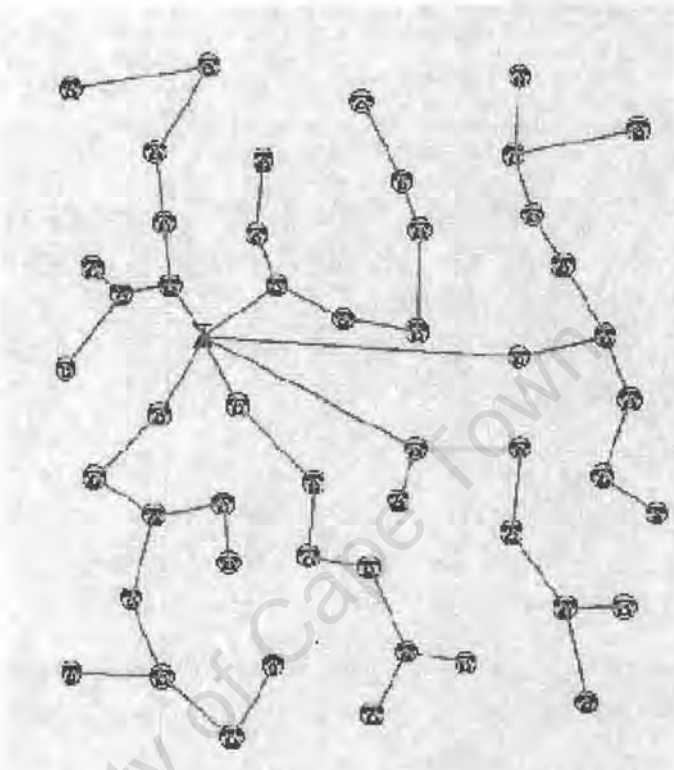

Figure 39: Result of applying Esau-Williams to 50 node test network

V-Function : This function is used to associate a trade-off function with each component. This governs the order in which each link is considered by the algorithm. By using different `V-Functions`, it is possible for the Unified Algorithm to realize different heuristics:

- *Kruskal* : V is set 0 for all components
- *Esau-Williams* : V is set to the distance from the component to the centre. This value is updated as components are merged.

CompStructInit, NeighbourStructInit : These two functions are responsible for initializing the structures that keep track of the components being formed. Depending on the algorithm that has been selected, these structures may be initialized in a different way.

FindBestNeigh, FindNextNeigh : These two functions are used to return the best neighbour for a specified node. `FindNextNeigh` is also used to remove the neighbour

selected from further consideration. In the case of *Esau-Williams* this involved popping a heap. In general, however, it is possible to use other structures besides heaps for keeping track of neighbours.

FindComponent : This function is passed a particular node and returns the cluster associated with that node

ResetV : This function modifies the value of V associated with a cluster once two clusters have been merged together.

Pseudocode for the algorithm can be found in Appendix B.

6.2 Terminal Assignment Algorithms

In Chapter 2, we introduced the terminal assignment problem and briefly described how a *greedy* terminal assignment algorithm works. While greedy algorithms usually perform reasonably well in practice, they can result in solutions with some expensive assignments of terminals, especially when capacity constraints are tight. *Semi-greedy* or *alternating-chain* algorithms produce more economical solutions than their greedy counterparts, and a description of this class of algorithms follows. Kershenbaum's implementation of an alternating-chain algorithm, which has been included in ROLAND's family of algorithms, is presented.

6.2.1 Alternating-Chain Algorithms

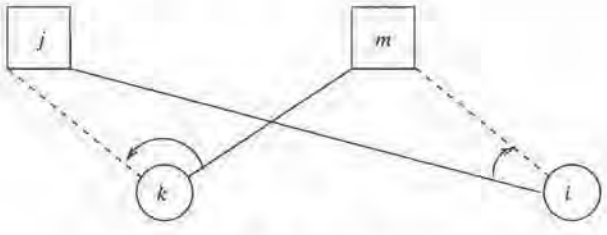

Figure 40: A terminal exchange

The principle behind this family of algorithms rests on the ability to exchange the assignment of a pair of terminals. The simplest case is illustrated in Fig. 40, where if terminal i is

assigned to concentrator j, and terminal k is assigned to concentrator m, and the following condition holds:

$$c_{ij} + c_{km} > c_{im} + c_{jk}$$

we should swap the assignments to lower the cost of the network.

A similar case exists if certain assignments are disqualified due to capacity constraints. In this case, exchanges can sometimes be discovered that result in feasible solutions.

The exchange above can be made because w_i, terminal i's demand is greater than the demand of terminal j, w_j, and concentrator has the capacity spare to accommodate w_i, while j doesn't.

This principle can be used to improve a greedy algorithm that leaves some terminals unassigned. These outstanding terminals can be assigned to concentrators without spare capacity (slack), and then exchange them to concentrators with slack. This technique does not guarantee an optimal solution, but does help to rectify the major problems associated with greedy terminal assignment algorithms.

6.2.2 Extending the exchange-chain

The major problem with adopting this "exchange approach" lies in the cost of performing an exchange, which is $O(T^2)$. Greedy algorithms, on the other hand, are O(TC). For small network sizes, this is not too serious, but for larger network sizes it might be necessary to identify a subset of "giver" and "taker" concentrators, to limit the number of exchanges that are performed.

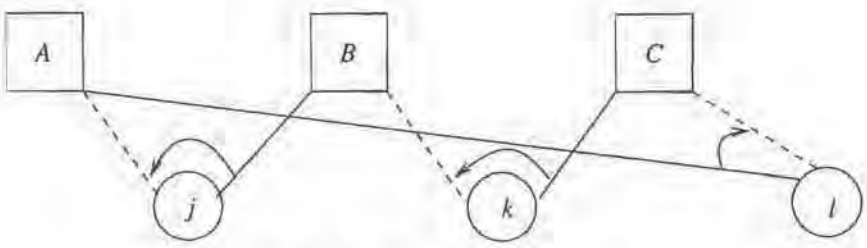

Figure 41: A more complete exchange

In Fig. 41, terminal i is connected to concentrator A which is relatively far away. It is desirable to assign i to concentrator C, but C's capacity has been exhausted. Moving

terminal k to B would make room for i at not much expense as B is close to k. There is not enough spare capacity at B to accommodate k, however. Terminal j can be moved to concentrator A either because there is spare capacity or due to the fact that relocating i would create spare capacity. Exchanging a terminal between any of the two concentrators does not make sense in isolation, but the three way exchange results in a more favorable assignment of the terminals.

Extending the notion of terminal exchange to these *chains* of terminals further increases the complexity with $O(T^k)$ exchanges being possible among k terminals. By making a few key observations and adopting an orderly approach to the problem, a provably optimal solution can be obtained. The algorithm is based on these few observations:

1. In the best case, every terminal will be assigned to its nearest concentrator. Only when the capacity of a concentrator is exceeded, will terminals need to be moved elsewhere.

2. If a terminal is assigned to a nearby concentrator, the only reason to move it is if the move will make room for another terminal that would need to make an even greater detour if it is not permitted to connect to this concentrator.

3. Given an incomplete optimal solution with k terminals already assigned, an optimal solution with $k+1$ terminals assigned can be found by finding the best way to add the $k+1$st terminal to the k terminal solution. This might mean that some of the first k terminals need to be moved.

The step from going from a solution with k terminals to a solution with $k+1$ terminals is known as an *augmentation*. The sequence of reassignments that need to occur to go from the one solution to the other is known as an *augmenting path*. The trick of finding the best solution at each stage lies in not having to consider all the T^C possible combinations of assignments.

6.2.3 The Assignment Problem

The terminal assignment problem has the same structure as a classical problem known as the *Assignment Problem*. This can best be illustrated by considering a matrix containing

positive numbers. The objective is to select elements of the matrix, so as to maximize the total of the numbers selected. At most one value per row, however, may be selected. A solution corresponds to a set of numbers, with at most one value selected from each row. Note that any subset of the solution is also a solution. Any solution S_2 that contains more elements than another solution S_1 must contain an element in a row that is not included in S_2. In fact, this is a problem defined on a *matroid.* We shall not examine matroids in any detail here; the interested reader is referred to [Law76]. There are several useful theorems surrounding matroids, not least of these being that *a greedy algorithm is guaranteed to find an optimal solution to a problem if the solutions to a problem form a matroid.*

Returning to our matrix problem, a greedy algorithm selecting the largest element in each row, is guaranteed to find an optimal solution. If we add a further constraint that at most one element from each *column* may be selected, a greedy algorithm can no longer guarantee an optimal solution.

For example, given the matrix:

```
4 8 9
7 6 8
8 2 3
```

if we greedily pick the 9 in the first row, the best solution we can obtain is a total of 23, as opposed to the optimal solution which chooses the three 8s, resulting in a value of 24. In fact, the solutions to the problem are the intersection of two matroids. It can be shown that augmenting path algorithms can find optimal solutions to problems defined on the intersection of two matroids [Law76].

If you think of the rows as people and the columns as jobs, the problems translates into assigning people to jobs: the *Assignment Problem.* The terminal assignment algorithm has the same structure and corresponds to assigning terminals to concentrators. The constraints now are that no more than W terminals (the capacity of the concentrators) may be chosen in each column and that one terminal is picked in each row (i.e. that each terminal has been assigned to a concentrator). The restriction is now on the number of terminals picked (as opposed to the sum of their weights), and involves *minimizing* the cost of assigning terminals to concentrators. This problem is still defined on the intersection of two matroids, and an

augmenting path algorithm is guaranteed to find the optimal solution. In the next section, we describe such an algorithm as presented by Kershenbaum [Ker93].

6.2.4 The Assignment Algorithm

The algorithm assumes:

- All the terminals have the same weight of 1.
- A concentrator has a capacity of W terminals

A solution exists if and only if $T \leq WC$. In addition, a feasible solution exists if an assignment may be found which does not violate the capacity constraints. This can be tested immediately and if no such solution exists, the algorithm can be halted.

If a feasible solution can be found, the algorithm continues by trying to assign the terminals to their nearest concentrator. If this does not assign more than W terminal to any concentrator (i.e. the capacity constraint is not exceeded), the optimal solution has been found, and the algorithm has terminated.

If we have not been able to assign all the terminals to their nearest concentrator, and only k have been able to be assigned, we have a *partial solution* which is optimal for the k terminals. We now need to find an optimal augmentation. This is achieved by constructing an **auxiliary graph** as shown in Fig. 42.

6.2.5 Auxiliary Graphs

An auxiliary graph is defined as follows:

1. S and F are two special nodes representing the start and end of all of the augmenting paths. The optimal augmentation is the shortest path from S to F.
2. All the other nodes in the graph are labelled xY, representing the connection of terminal x to concentrator Y. Squares in the graph labelled xY represent that x is currently assigned to Y, while circles represent that x is not currently assigned to Y.

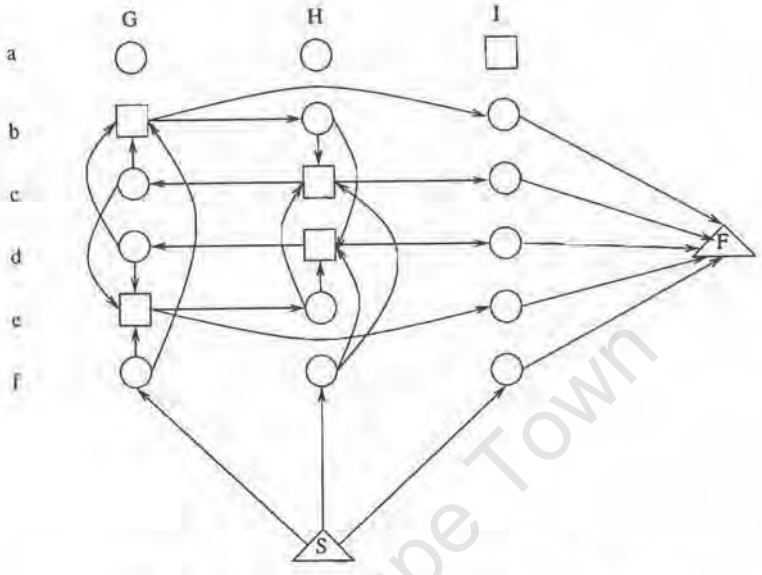

Figure 42: Original auxiliary graph [Ker93]

3. An arc (S, xY) exists for all concentrators from the start node,S, to each (unassigned) terminal, x. This corresponds to the assignment of terminal x to concentrator Y. The length of this arc represents the cost of connecting x to Y.

4. An arc (x_1Y, x_2Y) exists for all x_1 **not** currently assigned to Y, all x_2 currently assigned to Y, and all Y currently full. These arcs correspond to reassigning x_2 to another concentrator to make room for terminal x_1. The length of such an arc is $-c(x_2Y)$, as making the decision to move this terminal results in a reduction of x_2Y.

5. An arc (xY, xY_2) exists for all terminals, x, currently assigned to a full concentrator Y, and for all other concentrators Y_2. These arcs correspond to reassigning terminal x to concentrator Y_2, and the length of such an arc is $c(xY_2)$. Note that Y_2 may or may not be full.

6. An arc (xY_2, F) exists for all terminals, x, currently assigned to a full concentrator Y and for all other concentrators Y_2 that are not full. These arcs correspond to ending the augmentation, by reassigning x to Y_2, and the cost of such arcs is 0.

It should be noted that there are no arcs to or from terminals that are assigned to a concentrator that is not full. These terminals do not need to be moved, and may only later be moved into an augmentation path as their assigned concentrators fill up.

Every path in the augmentation graph from S to F corresponds to a valid augmenting graph. A number of paths exists from S to F in the graph. Following one of these paths results in a new solution with an additional terminal assigned and no concentrators overloaded. After each augmentation, the auxiliary graph needs to be reconstructed. This may result in some previously uninvolved terminals (formerly associated with concentrators that were not full) becoming involved as their assigned concentrators reach full capacity. The shortest path from S to F corresponds to the *optimal solution.*

As each terminal is added to the solution, the new auxiliary graph is constructed, and a shortest path algorithm is applied to the graph to determine the shortest augmentation path through the graph from S to F. Dijkstra's classical algorithm [Dij59] for determining the shortest path between two nodes in a graph is presented in the section.

6.2.6 Dijkstra's Algorithm

Let (V,E) be a weighted digraph. Let A be a vertex. The objective is to find, for each vertex x, the distance, $d[x]$, from A to x and the shortest path from A to x. We assume that for y and z in V, $w(y,z)$ is defined by:

$$w(y,z) = \begin{cases} 0 & \text{if y = z} \\ \infty & \text{when (y,z)} \notin E \\ \text{weight of the edge from y to z, otherwise} & \end{cases}$$

We define *pathto*(z) to be the list of vertices in the shortest current path from A to z.

```
begin
    for each x ∈ V do
    begin
        d[x] := w(A,x)
        pathto(x) := A
    end
    Mark vertex A
    while (unmarked vertices remain that are a finite distance from A) do
    begin
        x := one of the unmarked vertices who distance from A is minimal
```

```
            Mark vertex x
            for each unmarked y ∈ V such that (x, y) ∈ E do
                begin
                    d' := d[x] + w(x,y)
                    if d' < d[y] then
                      begin
                          d[y] := d'
                          pathto(y) := pathto(x),x
                      end
                end
        end
end
```

6.2.7 Compressing the Auxiliary Graph

The auxiliary graph has $O(TC)$ nodes and $O(WTC)$ edges. An augmentation is therefore at least $O(WTC)$ and possibly as bad as $O(T^2C^2)$. It is possible, however, to compress the auxiliary graph into $m + 2$ nodes, where m is the number of currently full concentrators. Fig. 43 shows the compressed auxiliary graph for the graph shown in Fig. 42. We do not describe this process here, but the interested reader is referred to [Ker93] for more details.

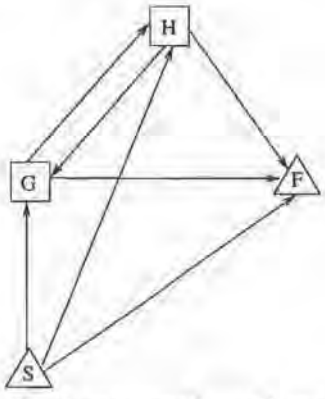

Figure 43: Compressed auxiliary graph

6.2.8 Algorithm Complexity

The number of nodes in the compressed auxiliary graph can never exceed C, so the shortest path algorithm can be completed in $O(C^2)$. Setting up an augmentation graph requires $O(TC)$ operations, and the backtracking and augmentation requires only $O(C)$ operations. T is usually larger than C and dominates.

Chapter 7

ROLAND's Graphical User Interface

In Chapter 3, ROLAND's architecture was presented and a brief description of the various components was provided. Here, we focus on the lower level implementation details of the GUI, with the emphasis being on the classes that make up the Local Access Network *Editor*, concluding with a description of the *test data generator* implementation.

7.1 GUI Implementation Details

7.1.1 Adopting an Object Oriented Approach

ROLAND is written in C++ and has been designed using an *object-oriented* approach. As emphasized in Chapter 2, the aim of ROLAND's design has been the production of a modular environment, which lends itself to easy extension or modification. The benefits of *encapsulation* and *specialization* that an object-oriented approach affords a program greatly helps to achieve this aim. New algorithms can easily be added to the tool, by making additions and slight modifications to a few classes.

7.1.2 Platform Independent Implementation using zApp

Before proceeding to examine the major classes constituting ROLAND's GUI, we briefly discuss the class library used in the GUI development, namely **zApp**, a portable C++ application framework developed by Rogue Wave [Rog85]. Applications developed with zApp are completely portable across a variety of operating systems including Windows, Win32, OS/2 and UNIX X/Motif.

7.1.3 zApp's Features

zApp's class library provide the following features:

- *a complete window hierarchy* catering for application windows, child windows, Multiple Document Interface (MDI) parents, dialogues and other types of windows;
- *a collection of standard controls* such as buttons, bitmaps, list and combo boxes and edit lines;
- *menus and toolbars*, including pulldown, popup and cascading menus;
- *events* are encapsulated into an event handing subsystem, removing the need of matching operating system messages;
- *a collection of graphical objects* such as pens, brushes and fonts.

The various graphical elements used in ROLAND's GUI are derived from zApp's base classes.

7.1.4 Writing an Application in zApp

While not wanting to focus on the nitty gritty low-level GUI implementation details here (the reader is referred to [Rog85] for a comprehensive introduction to zApp's class hierarchy), a brief discussion on the construction of an application using zApp is needed to provide the context for later discussion.

A zApp application's architecture is based on the typical *event-driven model*. It is the responsibility of the programmer to write routines that react to events such as the movement of the mouse of the clicking of a mouse button.

In general, the user interface portion of a program can be broken into three components:

1. *events* that are an application's input (e.g. keyboard input or mouse movements)
2. *event handlers* that are functions which process events. Event handlers are associated with graphical objects such as windows, menus, scroll bars and buttons.
3. *the event dispatcher* that directs the various events to the appropriate event handler for processing

zApp breaks a window down into two parts, namely a *frame* and a *pane*. Frames are unable to produce any graphical output, and are like a painting's frame in that they are "empty" in the middle and they need to contain another object to produce any output. Various window *panes* exist for the display of text or graphics.

7.2 ROLAND's LAN Editor

7.2.1 High-level Overview

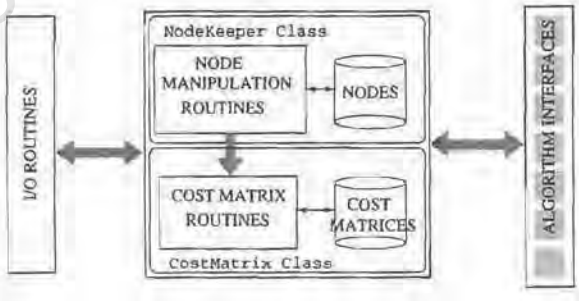

Figure 44: Structure of LAN Editor

The basic architecture of ROLAND's Local Access Network Editor is shown in Fig. 44. The `NodeKeeper` class is responsible for the storage and manipulation of the various network elements. The *CostMatrix* class keeps track of the cost of connecting nodes, and updates this information as nodes are moved in the network. We shall now proceed to examine these two classes in greater detail.

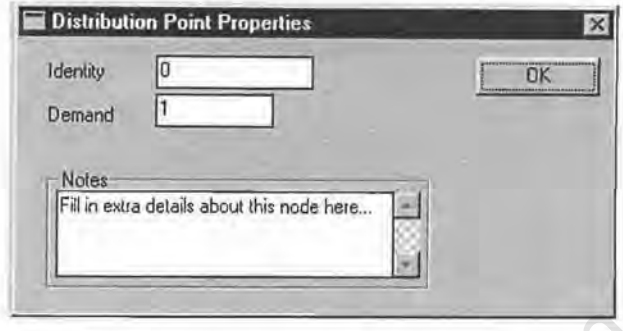

Figure 45: The Distribution Point Dialogue Box

7.2.2 The NodeKeeper Class

Types of Network Node

There are basically four types of nodes that need to be represented in any network scenario created in ROLAND:

1. A distribution point : a source of demand for telephone lines in the network
2. an installed concentrator : some remote device connected to the local exchange that is able to "service" a number of distribution points' demand
3. a potential concentrator site : a physical location that seems to be a good situation for positioning a concentrator

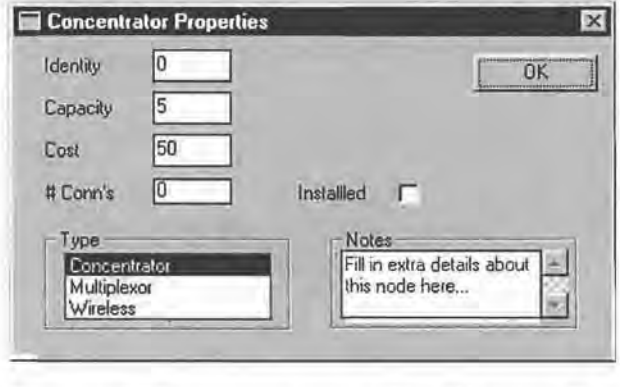

Figure 46: Concentrator and Site Dialogue Box

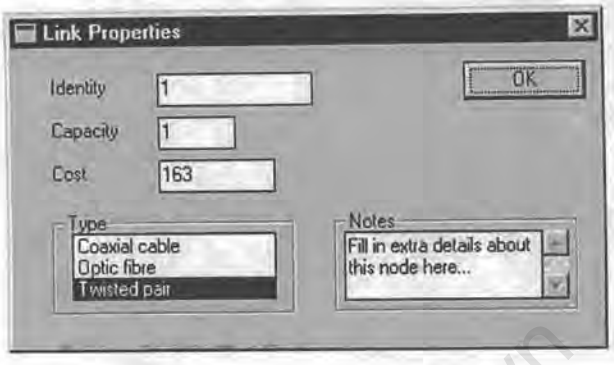

Figure 47: Link Dialogue Box

4. a link : some means of connecting a distribution point to a concentrator or another distribution point, be it copper cables or wireless communication.

The four classes modelling these four network components have been designed with the view of providing generic "technology-independent" structures, that can be used to represent different actual devices. Associated with each of these objects is a particular form used to capture the information about these network elements. The properties associated with each of these nodes are well illustrated by the dialogue windows used to initialize the objects (see Fig. 45, Fig. 46 and Fig. 47).

The concentrator and concentrator site nodes share the same class. All that distinguishes these two types of node is a flag *installed*, indicating whether the concentrator has been installed or not.

Storing the network elements

Fig. 48 shows a graphical representation of the `NodeKeeper` class. Four linked lists are used to store the four different types of network elements, and the `NodeKeeper` class is responsible for the maintenance of the lists of network elements. As various nodes are created or destroyed, they are added to or removed from the relevant list. As concentrator sites are installed, or concentrators uninstalled, entries are moved between the concentrator and site lists. The class also contains the methods required for saving or loading a network scenario from file.

The addition or removal of a network element does not only effect the `NodeKeeper` class;

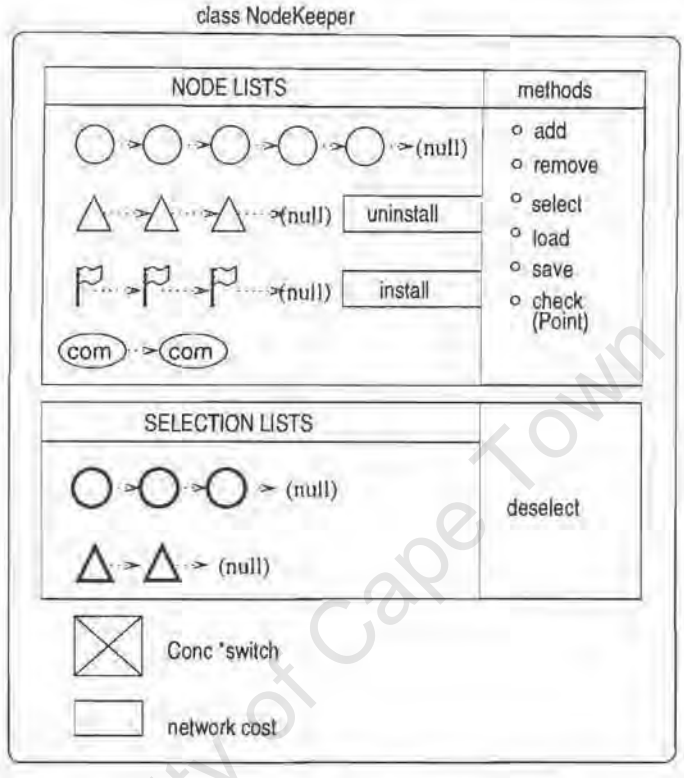

Figure 48: The NodeKeeper Class

entries also need to be added or removed from the cost matrices. We discuss the relationship between the `NodeKeeper` and `CostMatrix` classes after examining the latter in greater detail.

7.2.3 CostMatrix Class

Cost Matrix Representation

If we wanted just to be able to add entries into our cost matrix, an implementation would be relatively simple. We could simply keep count of the number of distribution points and concentrators we have added to the network. We would associate DP 0 with row zero of the cost matrix, DP 1 with row one, and so. Similarly, concentrator 0 would be associated with column zero, concentrator 1 with column one, and so on (see Fig. 49).

This approach becomes deficient as soon as we start to remove concentrators or distribution points from the network. To illustrate the problems encountered, consider the simple network and associated cost matrix shown in Fig. 50.

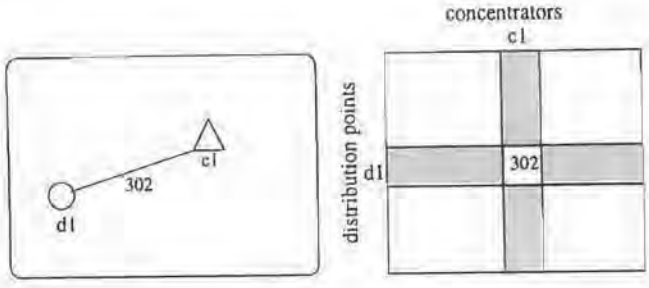

Figure 49: Making an entry in a simple cost matrix

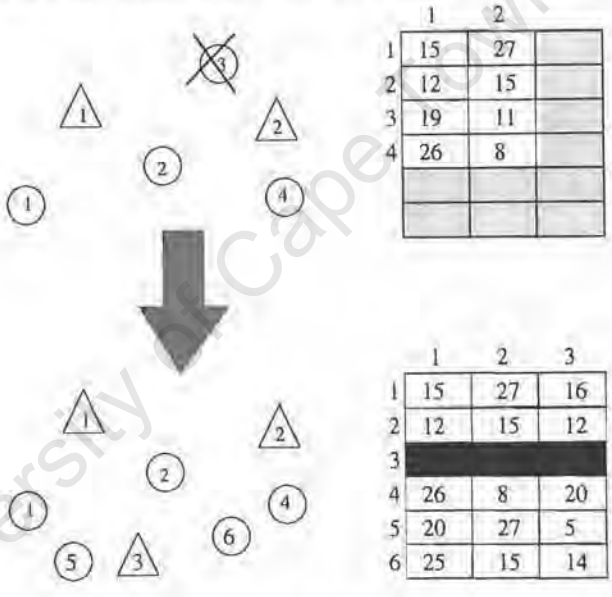

Figure 50: Gaps in the cost matrix

As entries are deleted from the network, the column or row associated with them in the matrix, are left unoccupied - never to be filled. One solution would be use a *dynamic array* approach, with some form of compaction technique that maintains the mapping between the columns and rows in the matrix and their associated distribution points or concentrators. This would require keeping a set of indices linking each distribution point and concentrator identification number to the relevant row or column number. As deletions occur, a compaction routine would be run to removes the vacated spaces left in the two-dimensional array. This can be achieved by copying the entries in the sparsely populated array into a smaller array which has no "gaps" between its entries. In addition, the index tables would need to be updated to reflect the association of new row and column numbers to certain concentrators or distribution points. This approach becomes costly however as the network

grows.

Hash Tables

The solution ROLAND adopts to the above numbering problem is through the use of hash tables. A *hash function* is used to map a node's identification number onto a column or row number in the cost matrix array. If that spot is already occupied by some other element, another vacant spot is found by means of techniques such as coalesced chaining. The dimensions of the two-dimensional array used to store the cost matrix should be set to the maximum possible number of network nodes that will be inserted in the network. Fig. 51 shows this scheme.

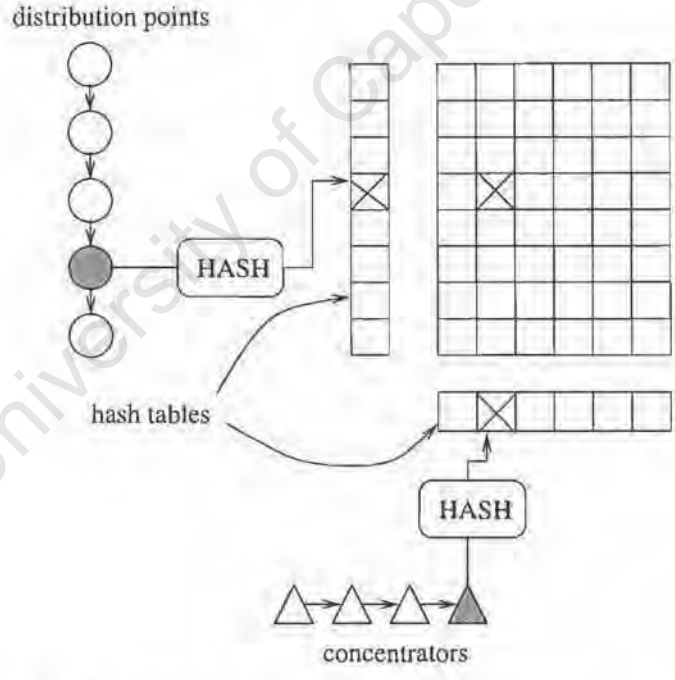

Figure 51: DP-Concentrator Cost Marix implemented with hashing

Removing a node from the network once again results in a vacant row or column in the matrix. This empty space is once again occupied when a node whose identification number hashes onto the same value as the deleted element, is added to the network. A disadvantage of this approach is that for small networks, a large section of the cost matrix array will not be used. It is hence desirable to set ROLAND's `MAX_DP` and `MAX_CONC` parameters not to be unrealistically large.

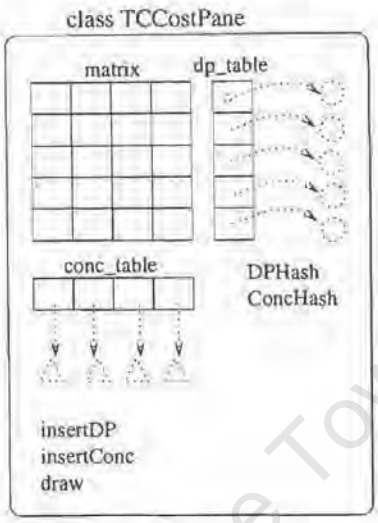

Figure 52: The TCCostPane Class

Cost Matrix Pane

ROLAND contains both concentrator location and line layout algorithms. This necessitates the inclusion of a cost matrix storing the cost of connecting the distribution points to the concentrators, as well as a cost matrix capturing the cost of connecting distribution points to each other. These two cost matrices are separated into two separate classes: `TCCostPane` for the former matrix, and `TTCostPane` for the latter. As the names of these classes imply, they are derived zApp *pane* windows and are responsible for both the graphical display of the cost matrix information, as well as the management of the underlying storage structures. Fig. 52 shows the `TCCostPane`; `TTCostPane` is almost identical, and calls routines common to the two classes.

7.2.4 Relationship between the NodeKeeper and CostMatrix Classes

When nodes are added or removed from the network, or moved from one position to another, the relevant changes need to be made to the cost matrices:

Creation or deletion of network elements: Creating a new distribution point, means that a new row has to be inserted in the d.p.-concentrator cost matrix, representing the cost of connecting this new node to each of the concentrators. A new row *and*

column need to be added to the d.p.-d.p. cost matrix. Inserting a new concentrator or concentrator site in the network, requires that a new column be added to the d.p.-concentrator cost matrix, recording the cost of connecting each distribution point to this newly inserted concentrator or site. When `NodeKeeper`'s *addDP*, *addConc* or *addSite* methods are called, the relevant methods (*insertDP* or *insertConc* belonging to `TCCostPane` and `TTCostPane`) for inserting the required elements in the cost matrices are called.

Movement of network elements: Each time a node in the network is moved from one point to another, this changes the distance from this node to every other node in the network. As our entries in the cost matrices are based on Euclidean distances between nodes in the network, movement of nodes requires a modification of entries in:

- the relevant *row* in the d.p.-concentrator cost matrix and *row* and *column* in the d.p.-d.p. cost matrix if a distribution point is moved;
- the relevant *column* in the d.p.-concentrator cost matrix if a concentrator or site is moved.

The *event handler* associated with mouse movements is invoked every time the mouse is moved, and if the *move-node* button is selected on the LAN Editor's toolbar, the *DPUpdate* or *ConcUpdate* methods of `TTCostPane` and `TCCostPane` respectively, are called to reflect the change in distances between network elements in the cost matrices.

7.3 Interfacing the Editor to the Algorithms

The ability to easily add new algorithm implementations to ROLAND's framework is one of the key goals of this project. In Chapter 3, we gave a high level description of the steps needed to incorporate a new implementation into the tool. In this section, we have a closer look at the various classes involved.

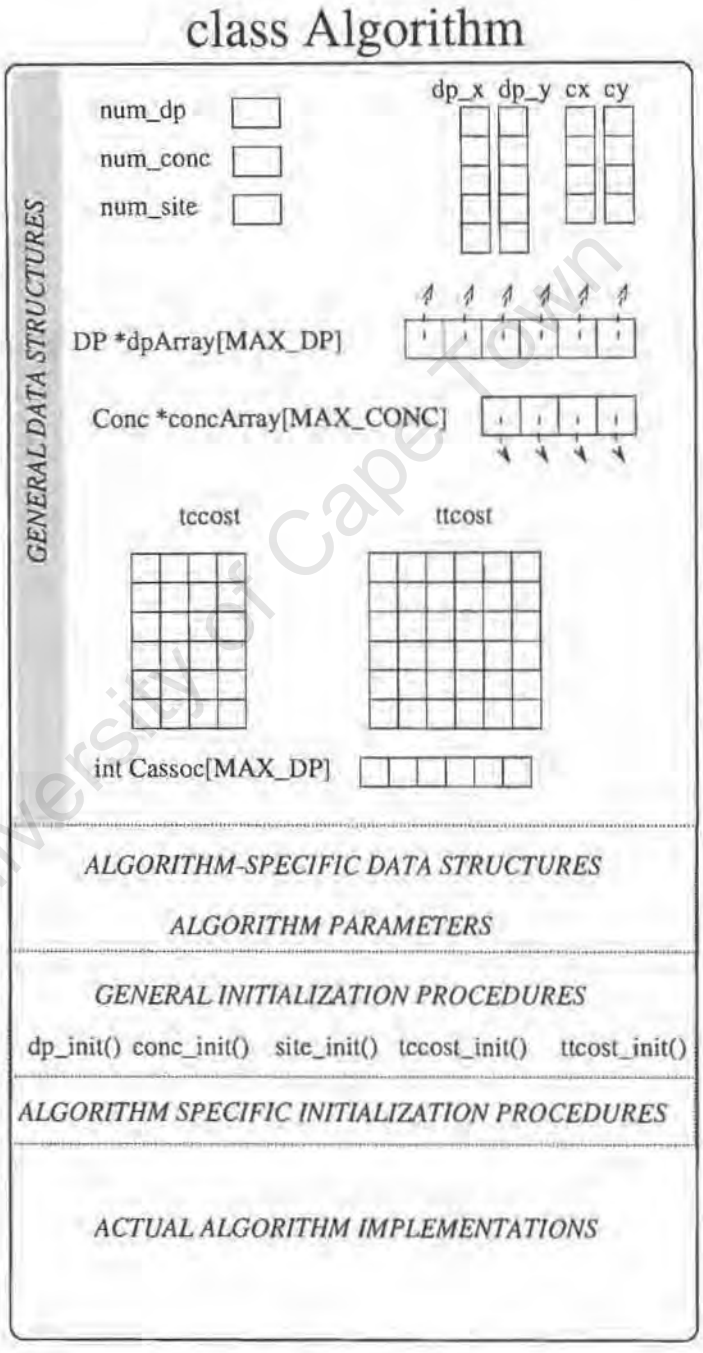

Figure 53: The Algorithm Class

7.3.1 The Algorithm Class

Algorithm implementations use different data structures to those used by the `NodeKeeper` class to store information about the elements in a network. Common data structures associated with algorithm implementations, such as arrays for storing the positions of distribution points or the cost of installing concentrators and data types for holding algorithm parameters, are contained in the `Algorithm` class. The methods associated with this class can be divided into three categories, namely *initialization* routines, the *actual algorithms*, and *"result-display"* methods.

An initialization method for each of the algorithms must be provided. This is called before the associated algorithm is invoked. Typically, such routines involve calls to simpler methods used to initialize the data structures associated with the various network elements. For instance, *initSite* is used to initialize structures used by the concentrator location algorithms to hold information about potential concentrator sites.

The *actual algorithm implementations* are the methods encapsulating the algorithm code. The methods presently incorporated into ROLAND all return the time taken to invoke the algorithm. After an algorithm is called, a call to another method to "transfer" the data from the algorithm's data structures to the structures used by the `NodeKeeper` object must be made. These methods usually also display the results produced by the algorithm. For instance the COM algorithm will generate new "centre-of-mass nodes" that are stored by the `NodeKeeper` object. As these nodes are reported, coloured boxes are used to temporarily display the clusters found, and the new centre-of-mass nodes are highlighted in coloured circles. As soon as the user clicks on the LAN Editor canvas, these lines disappear.

An *Algorithm* object is associated with each `LANPane` object. When an option specifying an algorithm is chosen from a LAN Editor's menu, the relevant initialization routine and algorithm are invoked by means of method calls to the `Algorithm` object.

7.3.2 The Algorithm Parameter Forms

Incorporating a new algorithm implementation into ROLAND requires the creation of a new form to allow the user to specify the algorithm's parameters. Relevant data types may also need to be added to the `Algorithm` class to cater for any new parameters introduced by the algorithm. ROLAND includes a "parameter form" class to facilitate the inclusion of

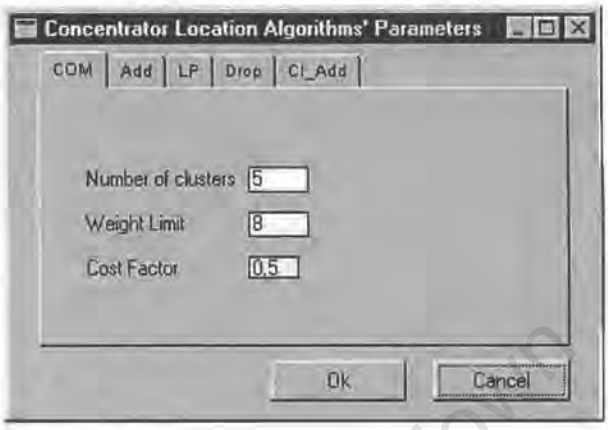

Figure 54: Concentrator Location Parameter Dialogue

new forms in the design tool. It is derived from zApps's `zNoteBookDialogue` class which allows multiple forms to be included together in a notebook form, where the relevant form can be chosen by means of a tag (see Fig. 54). New dialogues can be created using a resource editor, while the relevant data structures associated with each of the form's elements (e.g. text boxes) are provided by zApp's class library.

7.4 Test Data Generator

7.4.1 Types of Data

Due to the reluctance of telecommunications companies to release any real data for testing purposes, a test data generator was written to provide test cases for the analysis of algorithms. In deciding on what types of data to generate, Illenberger's model for common telecommunication demand distributions [Ill96] was considered. His model suggests what type of technology to use based on the *granularity* and *grouping* of demand (see Fig. 55). In summary, Illenberger uses three parameters to classify demand:

Density refers to the "intensity" of demand. This is proportional to the number of dwellings per square kilometer.

Granularity refers to the "granular" or "clustered" nature of the dwellings. In medium and low density area, this parameter has a great effect on the type of technology that should be used.

Grouping refers to the geographic grouping of dwellings due to topological constraints such as valleys or roads.

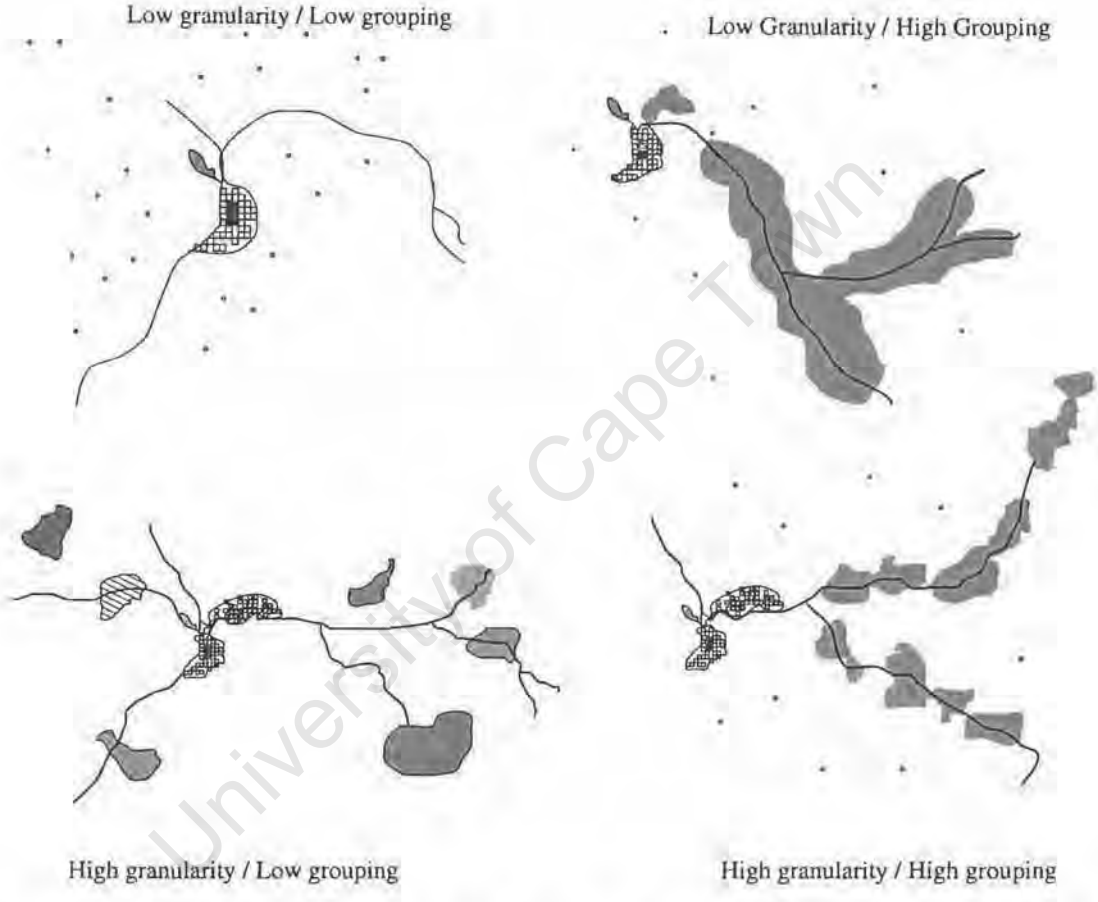

Figure 55: Granularity and Grouping: the shaded areas represent settlements that are typically situated near rivers

The test data generator form is shown in Fig. 56. It was decided to allow combinations of two types of distribution to be generated: (see Fig. 57):

random distribution : a random scattering of distribution points and concentrator sites can be generated by specifying that no clusters be formed, and setting the percentage of random points to be 100 percent.

localized distribution : Distribution points may also be generated in clusters of varying sizes. The proportion of points placed in these clusters (as opposed to being randomly

scattered), can also be determined by varying the percentage of random points generated. The size of the clusters generated is specified by a *degree-of-freedom* parameter that sets the maximum distance that a point may lie from a cluster's "centre".

As discussed previously, it is beyond the scope of this project to determine what type of technology to use in a particular situation. Should ROLAND be extended to include an interface to a system for suggesting appropriate network technology, it may be desirable to allow test data that takes into consideration topological features such as rivers and mountains, to be generated.

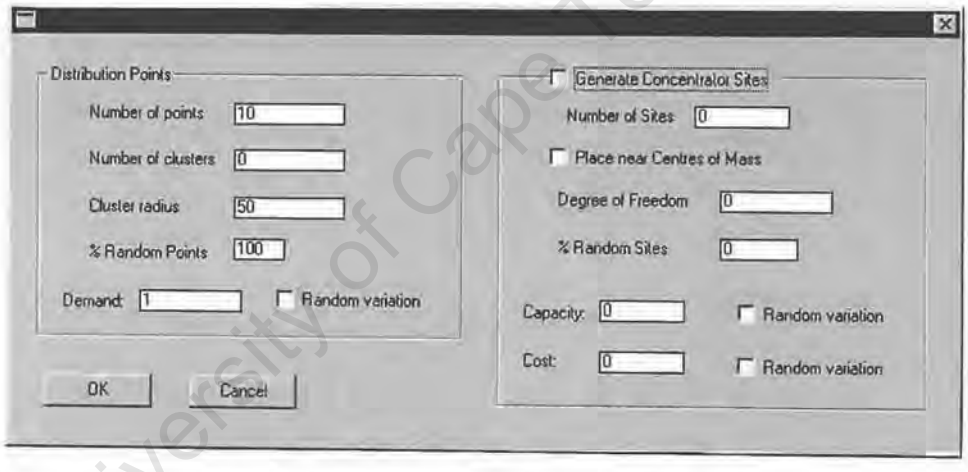

Figure 56: Test Data Generator Form

The experiments presented in the next chapter were all conducted using data that was generated by the test data generator.

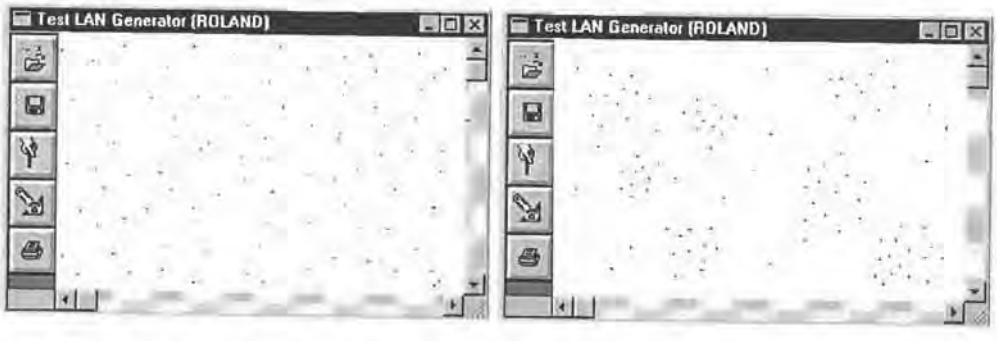

Figure 57: Random and clustered distribution points produced by the test data generator

Chapter 8

Algorithm Performance

8.1 Introduction

This chapter presents the results obtained from running the concentrator location algorithms on test networks generated by ROLAND's test data generator. The Cluster-Add Heuristic is compared to the Add Heuristic, and the effect that different values for this algorithm's parameters has on network cost, is also investigated.

All the experiments were conducted on a Pentium II 200MHz workstation with 64 MB RAM, running on Windows NT 4.0.

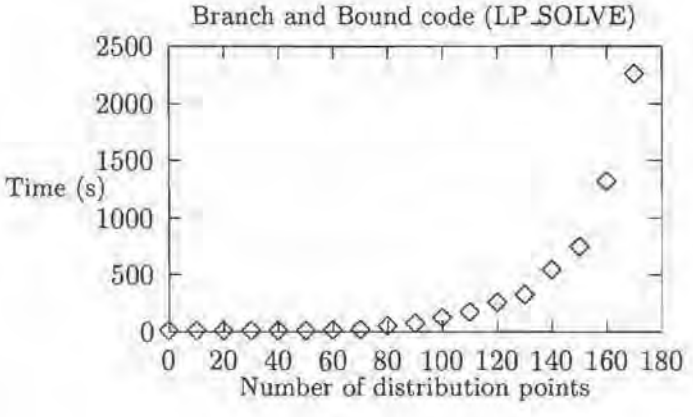

Figure 58: Execution time of linear programming code

8.2 Linear Programming Code Experiment

Fig. 58 shows the times taken for the shareware linear programming code LP_SOLVE to solve the mixed-integer problems representing increasingly larger networks (the number of concentrator sites increase linearly with the number of distribution points). As the number of nodes exceeds 130, the execution time increases sharply. This fact is highlighted by observing that for 130 nodes, it takes 533 seconds, while adding just 30 nodes to the network increases the time taken to obtain a solution to 2251 seconds. These results justify the need for heuristics to solve networks of more than 150 nodes.

8.3 Algorithm Execution Times

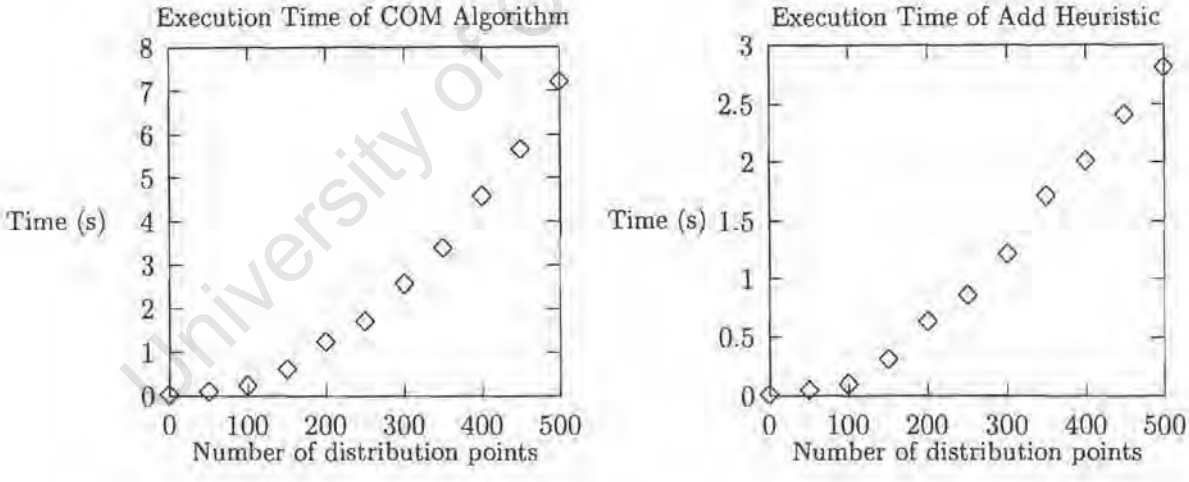

Figure 59: Execution times of Centre-of-Mass Algorithm and Add Heuristic

Fig. 59 shows the time taken to run the Centre-of-Mass Algorithm and Add Heuristic on varying size networks. Once again, the number of concentrator sites has been kept linear to the number of distribution points, with an average of ten distribution points to every concentrator. The cost of installing a concentrator has been set to 100. The COM Algorithm is $O(N^2)$, which explains the parabolic shape of its curve. The Add Heuristic, on the other hand, is $O(TC^2)$ (i.e. linear with respect to the number of distribution points), which explains why the curve is straighter, and the execution times are lower.

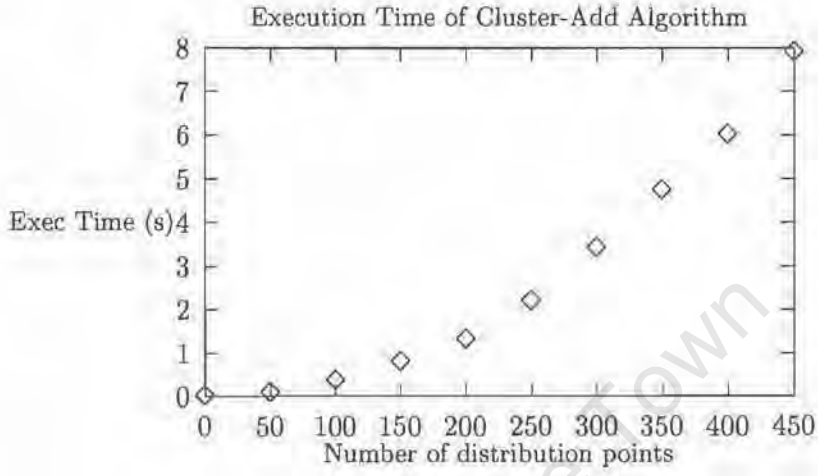

Figure 60: Execution time of Cluster-Add Heuristic

The Cluster-Add Heuristic requires that the Centre-of-Mass Algorithm be run on the network before determining which potential sites lie close to centres-of-mass. The times taken to execute the COM Algorithm dominates those of the Add Heuristic (bearing in mind the Cluster-Add Algorithm also reduces the number of distribution points and concentrators submitted to the Add Heuristic), which explains why the curves follows that of the COM Algorithm closely (see Fig. 60).

8.4 Cost of Solutions

Fig. 61 shows the relative performance of the Add Heuristic and Cluster-Add Algorithm on three types of data, namely:

Random: all the distribution points and concentrators are randomly distributed

Medium clustering: half of the distribution points are placed into clusters and the remaining half are randomly distributed. Concentrators are placed near the centres of these clusters.

High clustering: All the distributions points are placed into clusters, and the concentrators placed near the centres of these clusters. The cluster size chosen was 10 distribution points.

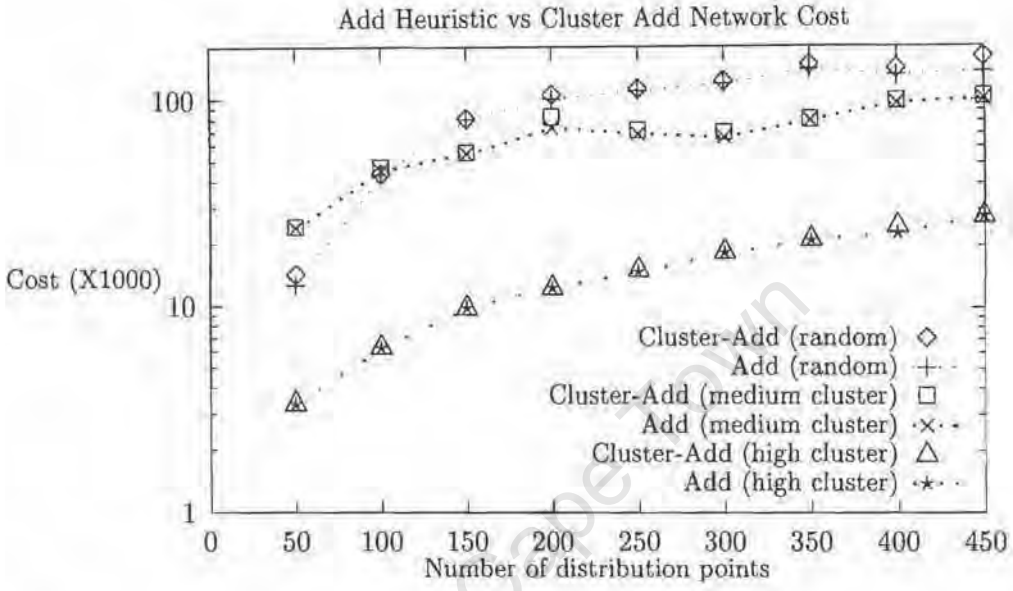

Figure 61: Relative costs of solutions produced by the Add and Cluster-Add Heuristics

The two algorithms perform almost identically for all three types of data. The Cluster-Add Heuristic adopts a greedy approach to associating distribution points to concentrators located near centres-of-mass. It may occasionally install a concentrator that is close to the centre-of-mass of a cluster that is relatively empty. Distribution points that would have been more economically serviced by other concentrators, may then become associated with this site. This could account for some of the cases where the Add Heuristic outperforms the Cluster-Add Heuristic.

It is interesting to note that the cost of the solutions obtained on the random data are higher than those obtained on the medium clustered data, and the medium clustered data yields higher cost networks than the more highly clustered data. This reflects the fact that is more highly clustered networks, the links to the concentrators are generally shorter, and less long expensive links from "remote" nodes to a concentrator are included.

8.5 Cluster-Add Algorithm Parameters

Whilst doing experimentation with the Cluster-Add Heuristic, it was observed that the choice of parameters for this algorithm could have a marked effect on the cost of solution obtained. In particular, the choice of the number of clusters (specified by the `num_clus` parameter) identified by the COM Algorithm and the `max_dist` parameter (specifying the maximum distance that a terminal may be from a concentrator if it is to be associated with it), has a significant if not predictable effect on the cost. The following two experiments illustrate this point, and the results produced are for a "medium-clustered" (as defined previously) test network of 300 distribution points, and 30 potential concentrators sites (see Fig. 62).

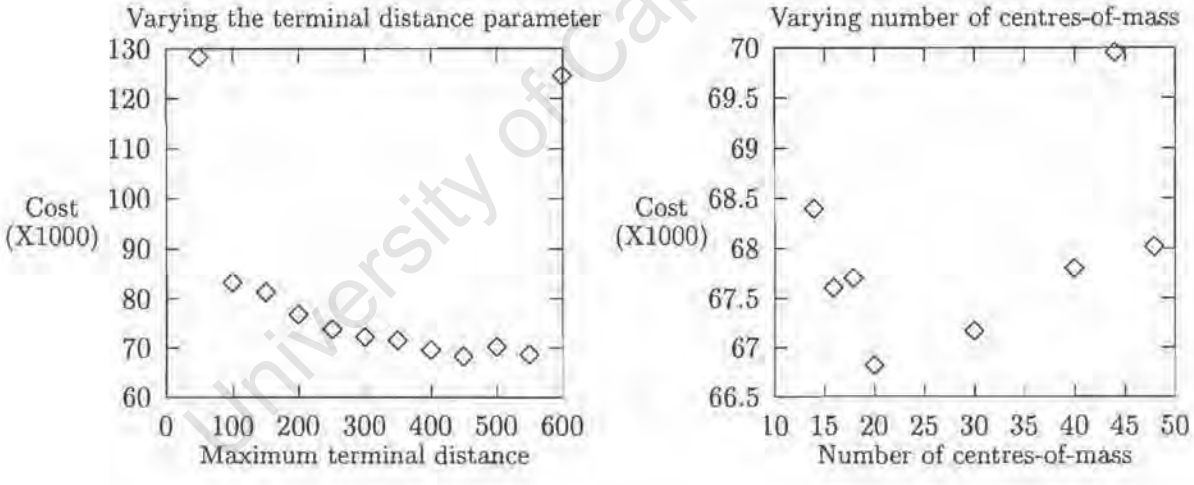

Figure 62: Effect of `max_dist` and `num_clus` parameters on network cost

Changing the terminal distance parameter : If the `max_dist` parameter is set too tight, very few distribution points are allowed to connect to a concentrator situated near a centre-of-mass, even though they may belong to the associated cluster. This means that they need to be serviced by other concentrators, resulting in the inclusion of potentially long and expensive links to more distant concentrators that elevate the network cost.

Setting `max_dist` too slack on the other hand, means that potentially expensive links

(that may be unnecessary) associate distribution points with a concentrator situated near a centre-of-mass. These distribution points then become prematurely fixed to a less favourable concentrator by the Cluster-Add Heuristic, whereas they may be able to be more economically serviced by closer concentrators.

Changing the `num_clus` parameter : The number of clusters chosen by the COM Algorithm determines the number of centres-of-mass that concentrators are tested with for locality. Varying this number was seen to have an effect on the cost of solutions yielded by the Cluster-Add Heuristic, although the relationship between the parameter and network cost could not be determined.

8.6 Conclusion

The results obtained by the few experiments that were run using ROLAND emphasized the need for some form of heuristic for solving concentrator location problems for networks comprising more than 150 nodes. The times taken to execute these heuristics were substantially shorter than the linear programming code. The new Cluster-Add Heuristic's execution times are dominated by time taken by the Centre-of-Mass Algorithm, and yields solutions comparable to those of the Add Heuristic. The choice of parameters for the Cluster-Add Heuristic were shown to be significant. Although no marked improvement in terms of network cost was obtained, ROLAND provided a convenient environment in which to conduct these tests, making use of the data provided by the test data generator.

Chapter 9

Conclusion

A network design tool ROLAND has been developed as a flexible and extendible environment in which to examine and evaluate a range of telecommunications local access network design algorithms. Three classes of algorithms are currently included in ROLAND, namely concentrator location, line layout and terminal assignment algorithms.

9.1 Features of Project

- An interactive GUI allowing designers to easily create network scenarios, using generic network components, has been developed.
- A number of existing telecommunications access network design algorithms, including heuristics and an interface to linear programming software, have been included in ROLAND. This allows the designer to explore a number of alternative approaches to obtaining an economical network design, while the three *classes* of problems covered by these algorithms, permits the exploration of three different aspects of access network design.
- The tool is useful for examining the behaviour and performance of design algorithms, and is presently applicable not only to *telecommunications* access network design problems, but also to other distribution networks (e.g. electricity distribution networks).
- An object-oriented approach to the system design has helped to create a modular piece

of software, where the GUI code is cleanly separated from the algorithm implementations. Commonly used data structures and routines have been placed in a separate class, aiding the development and integration of new algorithm implementations.

- By using the *zApp* class library, ROLAND can be compiled on a number of different platforms.
- A new hybrid concentrator location algorithm, the *Cluster-Add Heuristic*, was developed and integrated into ROLAND to demonstrate the development and inclusion of a new implementation into the tool. Although the algorithm did not offer any subtantial improvement in terms of performance, ROLAND provided a convenient environment in which to evaluate its behaviour.
- A test data generator that generates both random and clustered data was developed and allowed the algorithms to be tested on different types of test data.

9.2 Limitations and Future Work

- ROLAND has not yet been used in practice. Input from engineers and network designers at Telkom was assimilated at the beginning of the project. Additional suggestions were obtained from experts who viewed the work at two telecommunications conferences attended, and the final product was tested by an experienced telecommunications engineer. The tool still requires a longer period of evaluation to give an assessment of its practical value. The feedback obtained has suggested that the following improvements to the GUI be made:
  - adding the ability to manually modify the values held in the cost matrix.
  - adding the ability to view the network at varying levels of details (i.e. the ability to have a high-level view of the entire network and "zoom in" onto areas of interest).

  The latter restriction is a drawback in the current implementation of the tool, and results in the user having to scroll to view the entire network.
- ROLAND does not give any guidance in terms of deciding what technology is appropriate in particular scenarios. A rule-based decision support system that helps

to determine "appropriate and affordable " technology for an access network service area is being developed by [Ruc96]. Integrating ROLAND with this system would give network designers the ability to specify a particular scenario and have both the network topology and suitable technology suggested.

- Interfacing ROLAND to a Geographic Information System, would aid the designer in placing equipment in the network. Work done by [PS97] in this area forms part of a knowledge based planning tool for access network design and is especially useful in determining the locations for radio transmitters.

- ROLAND's palette of algorithms could be extended to include other heuristics applicable to telecommunications network design or even other domains. The inclusion of more sophisticated algorithms such as the dynamic concentrator location algorithm presented in Appendix C would help to make the tool useful in a wider context, while the software's design facilitates the inclusion of these algorithms.

Appendix A

Concentrator Location Algorithms Pseudocode

This appendix gives pseudocode for Kershenbaum's implementations of the Centre-of-Mass Algorithm, and the Add and Drop Heuristics described i n Chapter 5

A.1 Centre-of-Mass Algorithm

Main Algorithm

```
Algorithm COM
Accepts:
   nt,                      // number of distribution points
   weight[nt],              // demand of each distribution points
   x[nt], y[nt],            // locations of distribution points
   Cassoc[nt],
   nClus, Wlimit, Cfac      // algorithm parameters

Returns:
   Void
begin
   dcl nHeap[nt,heap] , sHeap[heap] , selVal[nt] , cWt[nt]
```

```
/* Initialization */

Cmax = 0 // Find the maximum cost
for_each( t , nt)
   for_each( t2 , nt )
      if ( t2 >= t ) // Consider only lower number neighbours
         cost[t][t2] <- INFINITY
      else
         Cmax <- max( Cmax , cost[t][t2] )
Cthresh = Cmax * Cfac // Set the cost threshold

for_each( t , nt ) // Set up the neighbour heaps
begin
   nHeap[t].init( nt , cost[t] )
   set_val[t] = nHeap[t].top // Cost to nearest neighbour
   CNum[t] = t // Cluster number
   CWt[t] <- weight[t] // Cluster weight
end
sHeap.init( nt , sel_val ) // Heap of nearest costs

nc = nt // Current number of clusters

/* Form clusters */
while( nc > nClus )
c = sHeap.pop /* Node with best nearest neighbour */
if ( c == -1 )    /* Heap is empty; no more merges */
   break
if ( cNum[c] < 0 )  /* Cluster no longer exists */
   sHeap.heap_replace(c , INFINITY ) /* Remove from consideration */
   continue
c2 = nHeap[c].pop // Nearest neighbour
if ( TestMerge( c , c2 ) )  // Test the validity of merging
begin
```

```
      Merge( c , c2 )  // Merge cluster c2 into cluster c
      nc = nc - 1
   end
   sHeap.heap_replace(c , (nHeap[c].top))

   // Set the associated cluster for each terminal
   for_each( t , nt )
   begin
      c = t
      while( cNum[c] != c )
      begin
         c = cNum[c]
         Cassoc[t] = c
      end
   end
end
```

TestMerge Procedure

This procedure test the feasability of merging the two clusters passed to it

```
Procedure TestMerge
Accepts:
   c1,                  // the two components to merge
   c2,
   Wlimit, cNum, cWt
Returns:
   Boolean
begin
   if( ( cNum[c2] >= 0 ) &&                    // clus c2 still exists
       ( cWt[c1] + cWt[c2] < Wlimit ) &&       // weight within limit
       ( cost[c1][c2] < Cthresh )       )      // cost within limit
```

```
      return ( TRUE )
   else
      return ( FALSE )
end
```

Merge Procedure

This procedure merges the two specifed clusters together.

```
Procedure Merge
Accepts:
   c1,
   c2,
   Wlimit, cNum, cWt
Returns:
   Void
begin
   cNum[c2] = c1
   x[c1] = (x[c1]*cWt[c1] + x[c2]*cWt[c2]) / (cWt[c1] + cWt[c2])
   y[c1] = (y[c1]*cWt[c1] + y[c2]*cWt[c2]) / (cWt[c1] + cWt[c2])
   cWt[c1] = cWt[c1] + cWt[c2]
   for_each ( t , nt )
      if ( (t != c1) && (cNum[t] != t) )
      cost[c1][t] <- Eucl( x[c1] , x[t] , y[c1] , y[t] )
   else
      cost[c1][t] <- INFINITY
nHeap[c1].set_heap(nt , cost[c1])
```

Cost is passed by value not by reference. The algorithm therefore destroys half of the cost matrix during processing. To avoid processing both terminal i as a neighbour of terminal j, and terminal j as neighbour of i, cost[i,j] $= \infty$ for $j \geq i$.

A.2 Add Heuristic

The pseudocode for Kershenbaum's implementation of the Add Heuristic for the capacitated concentrator location algorithm follows.

Add procedure

```
Algorithm Add
Accepts:
   nt,                    // number of distribution points
   nc,                    // number of concentrator sites
   weight[nt],            // demand of each distribution points
   Cexpense[nc],          // cost of installing concentrator
   cost[nt][nc],          // cost matrix
   Cassoc[nt],            // structure to record d.p.-conc assignment
   Wlimit,                // concentrator capacity constraint
   th_move                // penalty for moving terminals from one conc. to anot
Returns:
   void
begin
   dcl last_eval[nc]
   // Initially associate all terminals with the center (conc 0)
   Cassoc <- 0
   // Set up a heap of expense associated with each conc.
   // ( < 0 means savings )
   nSel <- 0
   Cexpense[0] <- INFINITY
   for_each (c, nc)
   begin
      Cxpense[c] <- EvalConc( c )
      last_eval[c] <- nSel
   end
   sHeap.init( nc, Cexpense )
```

```
   // Select concentrators

   while ( Top( sHeap ) < 0 )
   begin
      c <- Pop( sHeap )
      if ( last_eval[c] = nSel )
      begin
         AddConc( c, Cassoc )
         nSel <- nSel + 1
      end
      else begin
         Cexpense[c] <- EvalConc( c )
         last_eval[c] <- nSel
         sHeap.heap_replace( c, Cexpense[c] )
      end
   end
```

EvalConc Procedure

```
Procedure EvalConc
Accepts:
    c
Returns:
   integer       // savings achieved by adding concentrator
begin
   dcl delta[nt], permu[nt], Ter[nt]
   expense <- Cost
   slack <- Wlimit
   n <- 0
   for_each ( t, nt )
   begin
      s <- (cost[t][c] - cost[t][Cassoc[t]])
```

```
      if ( s < 0 )
      begin
         n <- n + 1
         delta[n] <- s
         Ter[n] <- t
      end
   end
   if ( n == 0 )  // no terminal benefitted
         return( expense )
   Sort( n, delta, permu )
   for_each ( i, n )
   begin
      p <- permu[i]
      t <- Ter[p]
      if ( delta[p] >= 0 )
         break
      else if ( ( weight[t] = slack ) &&
      ( (Cassoc[t] = 0) || (delta[p]+th_move < 0) ) )
      begin
         expense <- expense + delta[p]
         slack <- slack - weight[t]
      end
   end
   return( expense )
end
```

AddConc

```
Procedure AddConc
Accepts:
   c,                      // the concentrator to be added
   Cassoc[nt]              // records assignment of terminals to concentrators
Returns:
```

```
   void
begin
   dcl Cassoc[nt], delta[nt], permu[nt], Ter[nt]
   slack <- Wlimit
   n <- 0
   for_each ( t, nt )
   begin
      s <- cost[t][c] - cost[t][Cassoc[t]]
      if ( s < 0 )
      begin
         n <- n + 1
         delta[n] <- s
         Ter[n] <- t
      end
   end
   Sort( n, delta, permu ) // sorted values in delta are placed in permu
   for_each ( i, n )
   begin
      p <- permu[i]
      t <- Ter[p]
      if ( delta[p] >= 0 )
         break
      else if ( (weight[t] <= slack) && ( (Cassoc[t] = 0) || (delta[p] + th_move < (
      begin
         Cassoc[t] <- c
         slack <- slack - weight[t]
      end
   end

end
```

A.3 Drop Algorithm

The pseudocode of Kershenbaum's implementation for the uncapacitated case is given.

Main Drop Algorithm

As in the Add algorithm, a heap is set up based on the savings that are calculated for each concentrator; the concentrator which saves the most money is then greedily selected and dropped. The algorithm returns the number of concentrators selected (`nSel`), a list of those concentrators selected (`Conc_list`) and a vector showing which concentrator each terminal is associated with (`Cassoc`).

```
Algorithm Drop
Accepts:
   nt,                          // number of distribution points
   nc,                          // number of concentrator sites
   Ccost[nc],                   // cost of installing concentrator
   cost[nt][nc],                // cost matrix
   *nSel,                       // number of sites selected (passed by reference)
   Conc_list[nc],               // list of concentrators (passed by reference)
   Cassoc[nt]                   // structure recording d.p.-conc assignment
                                // (passed by reference)
Returns:
   void
begin
   dcl sHeap[heap], Csavings[nc], last_eval[nc]
   // Initially all the concentrators are selected
   nSel <- nc
   for_each (c, nc)
      Conc_list[c] <- c
   // Associate each terminal with its nearest conc.
   for_each (t, nt)
   begin
      best_conc <- INFINITY
```

```
      for_each (c, Conc_list)
      begin
         if (cost[t][c] < best_cost)
         begin
           best_conc <- c
           best_cost <- cost[t][c]
         end
      Cassoc[t] <- best_conc
   end
   // Set up a heap of savings associated with each concentrator
   Csavings[0] <- INFINITY // 0 is the center; do not drop it
   for_each (c,nSel)
   begin
      Csavings[c] <- EvalDrop(c)
      last_eval[c] <- nSel
   end
   sHeap.init (nc, Csavings)

   // Select concentrators
   while (top(sHeap) < 0)
   begin
      c <- pop(sHeap)
      if (last_eval[c] == nSel)
         DropConc(c)
      else begin
         Csavings[c] <- EvalDrop(c)
         last_eval[c] <- nSel
         sHeapheap_replace(c, Csavings[c])
      end
end
```

EvalDrop

Evaldrop determines the savings that are obtained by dropping a given concentrator. This works out to be the cost of the concentrator (which we save) minus the cost of moving all the terminals originally associated with this concentrator to new concentrators.

```
Procedure EvalDrop
Accepts:
   conc     // concentrator to evaluate
Returns:
   integer  // savings achieved by dropping concentrator
   // We save the cost of the conc by dropping it
   expense = -Ccost
   // If we drop the conc, we must pay to move the terminals associated
   // with it to the nearest conc.
   for_each (t, nt)
   begin
      if Cassoc[t] = conc )
      begin
         new_cost <- INFINITY
         for_each (i, nSel)
         begin
            c <- Conc_list[i]
            if ( (c!=conc) && (cost[t][c] < new_cost) )
               new_cost <- cost[t][c]
         end
         expense <- expense + new_cost - cost[t][conc]
      end
   end
   return (expense)
end
```

DropConc

DropConc is responsible for the actual dropping of the concentrator from the network:

```
Procedure Dropconc
Accepts:
   conc,
   *nSel,
   Conc_list[nt],
   Cassoc[nt]
Returns:
   void
begin
   // Rehome all terms associated with this concentrator
   for_each (t, nt)
   begin
      if (Cassoc[t] == conc)
      begin
         new_cost <- INFINITY
         for_each (i, nSel)
         begin
            c <- conc_list[i]
            if ( (c!=conc) && (cost[t][c] < new_cost) )
            begin
               new_cost <- cost[t][c]
               new_conc <- c
            end
         end
         Cassoc[t] <- new_conc
      end
   end
   flag <- FALSE
   // Remove concentrator from list of concentrators
   for_each (i, nSel-1)
```

```
    begin
        if (Conc_list[i] == conc)
        begin
            flag <- TRUE
            if (flag)
                Conc_list[i] <- Conc_list[i+1]
    end
    nSel <- nSel - 1
end
```

Appendix B

Line Layout Algorithms Pseudocode

B.1 Kruskal

The Main Algorithm

```
Algorithm Constrained_Kruskal
Accepts:
   n,                 // number of nodes
   costs[n][n],       // cost of various links in the graph
   weights[n],        // weights of each node in the graph
   Wmax               // limit on size of subtrees
Returns:
   solution[linked_list]  // the edges to be included in the solution

begin
   Initialize array of m edges: links
   link_heap.init( m, LinkStruc) // this creates a heap of edges
   Initialize structure tracking components: comp_struc
   numAccepted = 0    // no nodes (and edges)
   solution = NULL    // have been added to the solution
```

```
   while ((numAccepted < n) && (link_heap isn't empty))
   begin
      best_edge = link_heap.pop
      // next we find the components that the two nodes on either side of
      // best_edge belong to
      comp1 = FindComponent (comp_struc, best_edge's start node)
      comp2 = FindComponent (comp_struc, best_edge's end node)
      if (comp1 != comp2) && TestMerge(comp1, comp2, comp_struc, Wmax)
      // if the nodes on either side of best_edge belong to different
      // components and TestMerge returns true
      begin
         Merge(comp1, comp2, comp_struc)    // merge the two components
         append best_edge to solution
         numAccepted++
      end
   end
   return solution
end
```


Merge and TestComponent Algorithms

The procedure for merging two components together has to consider the case where one of the end-points is the local exchange. This is due to the fact that the multipoint lines that the algorithm is forming have to be kept separate. This can be achieved in the algorithm by associating a boolean variable with each component which is set to *true* when merging two components. In this way, the algorithm can avoid merging two multipoint lines that are already connected to the centre.

For the sake of illustration, it is assumed that the component data structure `comp_struc` has the fields *weight*, *connected_centre* and a pointer: *next*. The *next* pointer is used to keep track of the nodes in a component.

```
Procedure Merge
Accepts:
   c1,
   c2,              // the two components to merge
   comp_struc.         // the structure of the components
Returns:
   void
begin
   if (c1 == CENTRE)
      swap(c1, c2)
   if (c2 == CENTRE)
      comp_struc[c1].connected_centre = TRUE
   else begin
      comp_struc[c1].weight = comp_struc[c1].weight + comp_struc[c2].weight
      comp_struc[c2].next = c1
   end
   comp_struc[c1].connected_centre = comp_struc[c1].connected_centre ||
      comp_struc[c2].connected_centre
end

Procedure TestComp
Accepts:
   c1,
   c2,                 // the two components to merge
   comp_struc,         // the structure of the components
   Wmax.               // limit on size of structures
Returns:
   boolean
begin
   if (comp_struc[c1].weight + comp_struc[c2].weight <= Wmax) &&
      !(comp_struc[c1].connected_centre] && comp_struc[c2].connected_centre)
      return TRUE
   else return FALSE
```

```
end
```

B.2 Esau-Williams

Main Algorithm

```
Algorithm Esau-Williams
Accepts:
   nn,              // number of nodes
   cost[nn][nn],    // cost of various links in the graph
   weights[nn],     // weights of each node in the graph
   Wmax,            // limit on size of subtrees
   comp_struc,
   nbour_heap[nn],  // heaps of neighbours of each node
   toff_heap[c],    // heaps of tradeoff values ( c[i][j] - c[i][0] )
   tradeoff[c],     // tradeoff values
   ends[nn][2],     // end-points of links in the solution
   pred[nn]         // predecessor of each node in CMST
Returns:
   void
begin
   // we start by initializing the component structure and neighbour heaps
   for each node, n
   begin
      comp_struc[n].weight = weights[node]
      comp_struc[n].next = node
      comp_struc[n].cost_to_centre = cost[node][centre]
      comp_struc[n].connected_centre = FALSE
      nbour_heap[n].set_heap(n, cost[node][])
   end
   comp_struc[centre].connected_centre = TRUE
   comp_struc[centre].weight = 0
```

```
// tradeoff heap is then initialized
tradeoff[centre] = INFINITY
for each node, n
begin
  if ( n != centre )
  begin
     j = nbour_heap[n].top()
     tradeoff[n] = cost[n][j] - comp_struc[n].cost_to_centre
  end
end
toff_heap.set_heap(n, tradeoff)

// we now select the links we include in our solution

numLinks = 0
while (numLinks < n - 1)
begin
  n1 = toff_heap.top()
  n2 = nbour_heap[n1].top()
  c1 = FindComponent(n1)
  c2 = FindComponent(n2)

  if (tradeoff[n1] != (cost[n1][n2] - comp_struc[c1].cost_to_centre))
  begin
     tradeoff[n1] = cost[n1][n2] - comp_struc[c1].cost_to_centre
     toff_heap.heap_replace(n1, tradeoff[n1])
     continue     // return to beginning of while loop
  end
  if (TestComponent(c1, c2, Wmax, comp_struc)
  begin
     Merge (c1, c2, comp_struc)
     ends[numLinks] = (n1,n2)
     numLinks++
```

```
      end
      nbour_heap[n1].heap_replace(n2, INFINITY)
      j = nbour_heap[n1].top
      tradeoff[n1] = cost[n1][j] - comp_struc[c1].cost_to_centre
      toff_heap.heap_replace(n1, tradeoff[n1])
   end
end
```

Merge and TestComponent Algorithms

Esau-Williams uses the same TestComponent procedure as that used by Kruskal's greedy algorithm. The Merge procedure is extended to keep track of the cost of connecting the resulting component to the local exchange.

```
Procedure Merge
Accepts:
   c1,
   c2,              // the two components to merge
   comp_struc,      // the structure of the components
Returns:
   void
begin
   if (c1 == CENTRE)
      swap(c1, c2)
   if (c2 == CENTRE)
   begin
      comp_struc[c1].connected_centre = TRUE
      comp_struc[c1].cost_to_centre = 0
   end
   else
   begin
      comp_struc[c1].weight = comp_struc[c1].weight + comp_struc[c2].weight
      comp_struc[c2].next = c1
```

```
      comp_struc[c1].connected_centre = comp_struc[c1].connected_centre ||
         comp_struc[c2].connected_centre
      comp_struc[c1].cost_to_centre = min (comp_struc[c1].cost_to_centre,
         comp_struc[c2].cost_to_centre)
   end
end
```

B.3 Unified Algorithm

The algorithm implementation looks very similar to that of *Esau-Williams* presented previously. A bare-bones outline of the algorithm looks as follows:

```
Algorithm UnifiedAlgorithm
for each node, n
   V[n] = VFunction(n, comp_struc)
CompStrucInit(comp_struc)
NeighbourStructInit(neigh_struc)
for each node, n
begin
   bestJ = FindBestNeighbour (n, comp_struc)
   tradeoff[n] = cost[n][bestJ] - V[n]
end
toff_heap.set_heap (num_nodes, toff_heap)

// we now select the links for inclusion in the solution
num_links = 0
while (num_links < num_nodes - 1)
begin
   n1 = toff_heap.top()
   n2 = FindBestNeighbour (n1, neigh_struc)
   c1 = FindComponent(n1, comp_struc)
   c2 = FindComponent(n2, comp_struc)
   if (tradeoff[n1] != (cost[n1][n2] - V[c1])
```

```
    begin
        tradeoff[n1] = cost[n1][n2] - V[c1]
        toff_heap.replace(n1, tradeoff[n1])
        continue
    end
    if (TestComp(c1, c2, Wt_lim, neigh_struc))
    begin
        Merge(c1, c2, centre, neigh_struc)
        add link(n1, n2) to solution
    end
    j = FindNextNeighbour(n1, neigh_struc)
    ResetV(V, c1, c2)
    tradeoff[n1] = cost[n1][j] - V[c1]
    toff_heap.replace(n1, tradeoff[n1])
end
```

Appendix C

Dynamic Concentrator Location Problem

C.1 Introduction

This appendix gives a description of a model for Local Access Telecommunication Network (LATN) Design and Expansion Problems, as presented by [Shu91]. The problem is expressed as a mixed-integer linear programming (MILP) problem, where the installation of facilities is described by 0-1 variables.

C.2 Customer Demand

The LATN may be viewed as a tree. The root of this tree is the switching centre, and each non-root node is a customer node with an associated demand (this demand represents the number of circuits needed from the customer node to the root switching node).

The majority of telecommunications access networks have only one level of traffic compression. In other words, a customer node is connected either directly to the switching centre, or it is connected to a concentrator (which compresses the input traffic into a higher frequency signal) which is connected to the switching centre by means of a point-to-point (home-run) cable.

A demand or customer node could be a "collection point" of individual customer premises connected via a distribution network. It could also be a phone-shop servicing a particular zone.

C.2.1 Equipment Type

A variety of equipment exists for the compression of traffic:

- concentrators
- multiplexors
- remote switches
- fibre optic terminals ("fibre to the zone")

These electronic devices are conceptually equivalent from a modelling point of view and in this description of the access network, they shall be referred to as concentrators or facilities.

C.3 Modelling variables

- I - the index set of facility locations
- J - the index set of demand locations in the network
- T = the number of periods in the planning horizon
- P_i = the set of facility types which can be placed in location i, $i = 1, \ldots, n$. In this model, the "type" of facility refers to the "size" of the concentrator in question.
- D_j^t = the demand generated by node j in period t
- Q_p = the capacity of facility type p
- M_p = the maximum number of facilities type p allowed in each location;
- n_{ip}^0 = the number of facilities type p exisiting in location i in the beginning of the planning horizon

- S_p^t = the salvage value recovered when facility type p is retired in period t.

Due to the tree-like structure of the access network, for any pair of nodes (i,j) s.t. i $\neq$ j in the network, there exists a unique path P[i,j] linking them. A breadth-first search can be used to label the customer nodes of the tree. The root of the tree (i.e. the switching centre) is given the label 0. The predecessor of customer node i in the tree is called p_i. We label the arc (p_l, l) (where p_l is node l's predecessor in the tree) as arc l.

The capacity b_i ($i = \pm 1, \pm 2, \cdots, \pm p$, p = n + m) represents the existing cable capacity which can be used to send traffic between the two end-points of the arc.

Let x_l represent the total capacity of the cable on arc l.

C.3.1 Decision Variables

- X_{ijp}^t = fraction of customer j's demand in time t served by facility p at location i
- Y_{ip}^t = a binary variable: $Y_{ip}^t = 1$ if facility type p is placed in location i in period t; otherwise $Y_{ip}^t = 0$.

C.4 Costs

The model should simultaneously consider both *link expansions* and *facility location* (i.e. the installation of new concentrators) in a multi-period framework.

C.4.1 Costs of Installing Facilities

Cost Factors

The following cost *factors* are captured in the model:

- F_{ip}^t = the setup cost of having a facility type p open in location i in period t
- U_p^t = the variable cost per unit of demand served by facility type p in period t. The cost is independent of the location of the facility.

- C_{ij}^t = the transportation cost of serving a unit of demand at node j by a facility in location i in period t. It is independent of the *type* of facility. $C_{ij}^t = \infty$ if the demand at node j cannot be served by the facility in location i.

Setup Cost

- infrastructure investment (cost of equipment)
- cost of land, building
- fixed cost to install a dedicated fibre cable to send compressed traffic to the switching centre
- maintenance - repairs to equipment, servicing

This cost may be expressed as:

$$\sum_{t=1}^{T}\sum_{i=1}^{n}\sum_{p=1}^{m} Y_{ip}^t F_{ip}^t$$

The setup cost is essentially a *fixed* cost. The remaining two costs are variable and reflect the operation expenses appropriate to the load (volume-dependent operational costs) and the unit cost for sending the compressed signals to the switching centre.

Operational Costs

These are volume dependent and constitute the cost of serving customer demand by established facilities. It is captured by

$$\sum_{t=1}^{T}\sum_{j=1}^{m}\sum_{i=1}^{n}\sum_{p=1}^{|P_i|} U_p^t X_{ijp}^t D_j^t$$

Transportation Costs

These are the costs of delivering demands to facilites and are captured by:

$$\sum_{t=1}^{T}\sum_{i=1}^{n}\sum_{p=1}^{|P_i|}\sum_{j=1}^{m} C_{ij}^t X_{ijp}^t D_j^t$$

C.4.2 Costs of Cable Expansion

If the capacity demand along a particular cable exceeds the existing capacity b_l, a fixed cable expansion cost F_l, as well as a marginal variable cable expansion cost c_l, occurs.

The fixed cost represents the expenses for digging trenches and laying down pipes. The variable costs include the cost of the cables and the maintenance cost.

Let x_i be the cable requirement on arc l, $l = \pm 1, \pm 2, \cdots, \pm p$. We set:

$$x_l = x'_l + x''_l$$

where $0 \leq x'_l \leq b_l$, and x''_l is the amount we are expanding the capacity of the cable beyond the existing capacity for i, $i = \pm 1, \pm 2, \cdots, \pm p$.

C.5 Formulation of Problem

This formulation does not take into account the cable expansion. It incorporates the costs of installing and operating concentrators.

$$Z_{IP} = min\{\sum_{t=1}^{T}\sum_{i=1}^{n}\sum_{p=1}^{|P_i|} Y_{ip}^t F_{ip}^t + \sum_{t=1}^{T}\sum_{j=1}^{m}\sum_{i=1}^{n}\sum_{p=1}^{|P_i|} U_p^t X_{ijp}^t D_j^t + \sum_{t=1}^{T}\sum_{i=1}^{n}\sum_{p=1}^{|P_i|}\sum_{j=1}^{m} C_{ij}^t X_{ijp}^t D_j^t\} \quad (5)$$

subject to the following constraints:

$$\sum_{i=1}^{n}\sum_{p=1}^{|P_i|} X_{ijp}^t = 1 \quad \forall j, t \quad (6)$$

$$\sum_{j=1}^{m} D_j^t X_{ijp}^t \leq n_{ip}^t Q_p \quad \forall i, t, p \quad (7)$$

$$Y_{ip}^t = 0, 1 \quad \forall i, t, p \quad (8)$$

$$X_{ijp}^t \geq 0 \quad \forall i, j, t, p \quad (9)$$

- Constraints (6) states that the demand at each customer node must be satisfied
- Constraints (7) states that the total demand served by each facility type should not exceed the available capacity of that particular facility.

- Constraints (9) captures the notion that a customer node can be served by more than one facility.

C.6 Tradeoffs

Transportation Costs vs Facility Costs : Installing a greater number of smaller facilities decreases the transportation costs, but increases the total facility costs.

Economies of Scale vs Discounted Costs : By making use of smaller facilites and spreading their installation over the planning horizon, the *total discounted costs* may be lowered. The *economies of scale*, however, dictates that larger facilities have a smaller unit cost.

Bibliography

[Aik85] C.H. Aikens. Facility location models for distribution planning. *European Journal of Operational Research*, 22:263–279, 1985.

[Bak93] G. Baker. High-bit-rate digital subscriber lines. *Electronics & Communication*, 5(5):279–283, 1993.

[Bau77] W.J. Baumol, editor. *Economic Theory and Operations Analysis*. Prentice Hall International, London, 4 edition, 1977.

[BCM+82] T.P. Byrne, R. Coburn, H.C. Mazzoni, G.W. Aughenbaugh, and J.L. Duffany. Positioning the subscriber loop network for digital services. *IEEE Trans. on Communications*, COM-30(9):2006–2011, 1982.

[BF77] R.R. Boorstyn and H. Frank. Large-scale network topological optimization. *IEEE Trans. on Communications*, COM-25(1):29–47, January 1977.

[Bie93] D. Bienstock. A lot-sizing problem on trees, related to network design. *Mathematics of Operations Research*, 18(2):402–422, 1993.

[BMSW91a] A. Balakrishnan, T.L. Magnanti, A. Shulman, and R. T. Wong. Models for planning capacity expansion in local access telecommunications networks. *Annals of Operations Research*, 33:239–284, 1991.

[BMSW91b] A. Balakrishnan, T.L. Magnanti, A. Shulman, and R.T. Wong. Models for planning capacity expansion in local access telecommunications networks. *Annals of Operations Research*, 33:239–284, 1991.

[BMW89] A. Balakrishnan, T.L. Magnanti, and R.T. Wong. A dual ascent procedure for large-scale uncapacitated network design. *Operations Research*, 37:716–740, 1989.

[Boy82] G. R. Boyer. Administration of new feeder technology. *IEEE Transactions on Communications*, 30(9):2029–2033, 1982.

[Cam95] G. Cameron. Modular visualization environments: past, present and future. *Computer Graphics*, May 1995.

[Car95] E.B. Carne, editor. *Telecomunications Primer: Signals, Building Blocks and Networks.* Prentice Hall, 1995.

[CSwn] G. Cho and D. X. Shaw. A depth-first dynamic programming algorithm for the tree knapsack problem. *Electronic Article*, Unknown. School of Industrial Engineering, Purdue University.

[DH69] J.F. Desler and S.L. Hakimi. A graph-theoretic approach to a class of integer programming problems. *Operations Research*, 17:1017–1033, 1969.

[Dij59] E.W. Dijkstra. A note on two problems in connection with graphs. *Numerische MAthematik*, 1:269 - 271, 1959.

[ER66] M.A. Efroymson and T.L. Ray. A branch-bound algorithm for plant location. *Operations Research*, 14:361–369, 1966.

[Erl78] D. Erlenkotter. A dual-based procedure for uncapacitated facility location. *Operations Research*, 26:992–1009, 1978.

[EW66] L.R. Esau and K.C. Williams. On teleprocessing system design, a method for approximating the optimal network. *IBM System Journal*, 5:142–147, 1966.

[FC72] H. Frank and W. Chou. Topological optimization of computer networks. *Proceedings of the IEEE*, 60(11):1385–1397, November 1972.

[Flo75] J.E. Flood, editor. *Telecommunication networks.* Peter Peregrinus Ltd., 1975.

[Gav85] B. Gavish. Augmented lagrangian based algorithms for centralized network design. *Transactions on Communications*, 33:1247 - 1257, 1985.

[Gav87] B. Gavish. Models for configuring distributed computing systems. *IEEE Trans on Computers*, 36:773–793, 1987.

[GJ79] Garey and Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Fremen, 1979.

[GS79] M. Guignard and K. Spielberg. A direct dual method for the mixed plant location problem with some side constraints. *Mathematical Programming*, 17:198–228, 1979.

[Har95] R. Harper. Public payphones: Evaluating our phone shop experience. In *Second Annual Pan-Asian Rural Communications Summit*. Alcatel Altech Telecoms, September 1995.

[Haw91] G.T. Hawley. Historical perspectives on the U.S. telephone loop. *IEEE Communications*, 29(3):24–28, 1991.

[HJS88] M.P. Helme, C. Jack, and A. Shulman. Planning for new services in the local loop. In *Proceedings Int. Telecommunications Conf.*, number 12, pages 5.2B.1.1 – 5.2B.1.12, 1988.

[Ill96] M. Illenberger. Access network technologies available to provide basic telephone service. In *Teletraffic '96*, pages 53 – 60, September 1996.

[Int83] International Telecommunication Union, Geneva. *General Network Planning*, 1983.

[JK95] Richard Johnsonbaugh and Martin Kalin. *Object-Oriented Programming in C++*. Prentice-Hall, 1995.

[KC74] A. Kershenbaum and W. Chou. A unified algorithm for designing multidrop teleprocessing networks. *IEEE Trans. on Communications*, 22:1762–1772, 1974.

[Ker93] A. Kershenbaum, editor. *Telecomunications Network Design Algorithms*. McGraw-Hill, 1993.

[KH63] A.A. Kuehn and M.J. Hamburger. A heuristic program for locating warehouses. *Management Science*, 9:643–666, 1963.

[KSK95] M. Koksalan, H. Sural, and O. Kirca. A location-distribution application for a beer company. *European Journal of Operational Research*, 80:16–24, 1995.

[KT81] R.M. Karp and R.E. Tarjan. Linear expected time algorithms for connectivity problems. *Journal of Algorithms*, 1:374–393, 1981.

[KvS72] A. Kershenbaum and R. van Slyke. Computing minimal spanning trees efficiently. *Proc Assoc for Computing Machinery Annu. Conf.*, pages 5188–527, August 1972.

[Lam94] Leslie Lamport. *LaTeX User's Guide and Reference Manual.* Addison-Wesley, 1994.

[Law76] Eugene Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart and Winston, 1976.

[LH89] E. Lakervi and E.J. Holmes. *Electricity Distribution Network Design.* Peter Peregrinus Ltd., London, 1989.

[LK89] C. Lo and A. Kershenbaum. A two-phase algorithm and performance bounds for the star-star concentrator location problem. *IEEE Trans. on Communications*, 37(11):1151–1163, November 1989.

[LL87] S. Lee and H. Luss. Multifacility-type capacity expansion planning: Algorithms and complexity. *Operations Research*, 35(2):249–253, 1987.

[Lus82] H. Luss. Operations research and capacity expansion problems: A survey. *Operations Research*, 50(5):907–947, 1982.

[LZC87] H.P.L. Luna, N. Ziviani, and R.H.B. Cabral. The telephonic switching centre network problem: Formalization and computational experience. *Discrete Applied Mathematics*, 18:199–210, 1987.

[Min89] M. Minoux. Network synthesis and optimum network design problems: Models, solution methods and applications. *NETWORK*, 19:313–360, 1989.

[MS77] P.V. McGregor and D. Shen. Network design: An algorithm for the access facility location problem. *IEEE Trans. on Communications*, COM-25(1):61–73, January 1977.

[NKT89] G.L. Nemhauser, A.H.G. Rinnooy Kan, and M.J. Todd, editors. *Handbooks in Operations and Management Science: Optmization*, volume 1. North-Holland, 1989.

[NW88] George L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, Inc., 1988.

[Pri57] R.C. Prim. Shortest connection networks and some generalizations. *BSTJ*, 36:1389–1401, 1957.

[PRS76] D.T. Phillips, A. Ravindran, and J.T. Solberg. *Operations Research:Principles and Practice*. John Wiley and Sons, inc., 1976.

[PS97] P. Premjeeth and K. Sandrasegaran. A gis aid in site selection for wireless rural network planning. In *Teletraffic '97*, pages 66 – 73, September 1997.

[Riv93] L. Kwabena Riverson. *Telecommunications Development: The Case of Africa*. University Press of America, 1993.

[Rog85] Rogue Wave Software, Corvallis USA. *zApp Programmer's Guide*, 1985.

[Row91] T.R. Rowbotham. Local loop developments in the U.K. *IEEE Communications*, 29(3):50–59, 1991.

[Roy86] T. J. Van Roy. A cross decomposition algorithm for capacitated facility location. *Operations Research*, 34:145–163, 1986.

[Ruc96] H. Ruck. Interactive rule-based decision support system for comparing access network technology alternatives. In *Teletraffic '96*, pages 46 - 52, September 1996.

[Sá69] G. Sá. Branch-and-bound and approximate solutions to the capacitated plant-location problem. *Operations Research*, 17:1005–1016, 1969.

[SC94] D. X. Shaw and G. Cho. The critical-item, upper bounds, and a branch-and-bound algorithm for the tree knapsack problem. *Electronic Article*, 1994. School of Industrial Engineering, Purdue University, http://palette.ecn.purdue.edu/ shawdx2/.

[SC95] D. X. Shaw and G. Cho. Limited column generation for local access telecommunication network design - formulations, algorithms, and implementation. *Electronic Article*, 1995. School of Industrial Engineering, Purdue University, http://palette.ecn.purdue.edu/ shawdx2/.

[Sha93] D. X. Shaw. Reformulation, column generation and lagrangian relaxation for several telecommunication network design problems. *Electronic Article*, 1993. School of Industrial Engineering, Purdue University.

[Sha94] D. X. Shaw. Reformulation, column generation and lagrangian relaxation for local access network design problems. *Electronic Article*, 1994. School of Industrial Engineering, Purdue University.

[Shu91] A. Shulman. An algorithm for solving dynamic capacitated plant location problems with discrete expansion sizes. *Operations Research*, 39(3):423–436, May–June 1991.

[Sim61] H.A. Simon. Modeling human mental processes. In *Western Joint Computer Conference*, 1961.

[Som92] Ian Sommerville. *Software Engineering.* Addison-Wesley, 1992.

[Spi69] K. Spielberg. Algorithms for the simple plant location problem with some side conditions. *Operations Research*, 17:85–111, 1969.

[Sta95] S. Stavrou. Telecommunications and the reconstruction and development programme. In *FRD Programme Report Series*, number 20, February 1995.

[SV90] A. Shulman and R. Vachani. An algorithm for capacity expansion of local access networks. In *IEEE INFOCOM '90*, pages 221–229, San Franciso, 1990.

[SZ82] G.M. Schneider and M.N. Zastrow. An algorithm for the design of multilevel concentrator networks. *Computer Networks*, 6:1–11, 1982.

[TRY86] R. Thomas, L. Rogers, and J. Yates. *Advanced Programmers Guide to UNIX.* McGraw-Hill, 1986.

[Whi95] B. Whitby. Rural traffic per line and calling patterns. *Unknown*, 1995.

[WLH91] D.L. Waring, J.W. Lechleider, and T.R. Hsing. Digital subscriber line technology facilitates a graceful transition from copper to fiber. *IEEE Communications Magazine*, 29(3):96–104, 1991.

[WWB78] D.T. Wang, L.S. Woo, and L.R. Bahl. Optimisation of teleprocessing networks with concentrators and multiconnected terminals. *IEEE Transactions on Computers*, C-27(7):594–604, July 1978.

[Yag73] B. Yaged. Minimum cost routing for dynamic network models. *Networks*, 3:193–224, 1973.

[Zad73] N. Zadeh. On building minimum cost communication networks. *Networks*, 3:315–331, 1973.

[Zad74] N. Zadeh. On building minimum cost communication networks over time. *Networks*, 4:19–34, 1974.