INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

SOLVING CONCENTRATOR LOCATION AND TERMINAL ASSIGNMENT PROBLEMS USING SIMULATED ANNEALING

Gene H.M. Kapantow

A Thesis
In
The Faculty
of
Commerce and Administration

Presented in Partial Fulfilment of the Requirements
for the Degree of Master of Science in Administration at
Concordia University
Montreal, Quebec, Canada

October 1996

ABSTRACT

Solving Concentrator Location and Terminal Assignment Problems
Using Simulated Annealing

Gene H.M. Kapantow

Centralized computer networks are hierarchical communication infrastructures in which a central computer services a large number of terminals or workstations. For a large network, some concentrators are commonly used to increase the cost efficiency. Several terminals are connected to a concentrator via low capacity lines, and each concentrator is connected to the central computer via a high capacity line. A concentrator is generally subject to technological constraints on the amount of traffic it can manage, and each terminal has its capacity requirement. The problem then is to determine the number and location of concentrators and to allocate the terminals to these concentrators at minimum cost. This is known as the concentrator location problem. If the number and location of concentrators are already known beforehand, the problem then reduces to determining the allocation of terminals only. This is known as the terminal assignment problem. These problems are NP-complete. Therefore, finding a polynomial time algorithm to solve them to optimality is highly unlikely. This study aims to develop some efficient algorithms based on simulated annealing to solve these problems. The results are compared to those given by some existing heuristics.

Acknowledgment

First and foremost, I would like to thank the Lord Jesus Christ for providing everything I need in this life. Without his constant love for me, I doubt if I would have been able to pursue my M.Sc. studies.

I have many people to thank for their help and support. First, I would like to express my deepest gratitude to my thesis supervisor, Dr. Jean-Marie Bourjolly, for the support, guidance, concern and valuable inputs he has given me during the course of this thesis. I am also indebted to him for introducing me to the simulated annealing algorithm.

I would also like to thank Dr. Samuel Pierre as my thesis co-supervisor for the valuable inputs he has given me during the course of this thesis. Many thanks also go to Dr. Mohan Gopalakrishnan for the help he has given me during the course of my M.Sc. studies and for the valuable inputs he has given in this thesis.

I would further like to thank my friend. Daniel Tomiuk, for his help and friendship not only during the course of this thesis but also during the whole course of my M.Sc. studies. I am also impressed by his intelligence and discussions which are always interesting and fruitful. My discussions with him have helped me much in doing my thesis. I am fortunate to have him as my friend.

A note of gratitude is also due to the administrative staff of the M.Sc. program. Particularly, I thank Mrs. Heather Thomson, Mrs. Theresa Sarazin-Wadey and the former M.Sc. program's assistant director, Mrs. Karen Fiddler, for their kind help during the course of my M.Sc. studies.

Finally, I would like to express my deepest gratitude to my father, my sisters and my brothers in law for all the support they have provided.

Dedication

~ In fond memory of my Mother ~

"Who had always been there when I needed her"

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Background and Purpose of the Study

The size and complexity of computer networks have been growing very fast in the last decade. In the past, a computer network usually only served a small number of users. Nowadays, it is common to find a computer network that serves hundreds or even thousands of users. It seems that this incredible growth will continue as the customers' needs and the telecommunication technology keep growing. As the size and complexity of computer networks grow, so too does the need for good design strategies.

Designing a modern computer network is a very difficult task (Gavish, 1982). "The overall complexity of the problem had led to the development of solution procedures in which the problem is partitioned into a hierarchy of subproblems that are solved one at a time" (p.356). One of these subproblems is the concentrator location problem, on which this study is focused.

Basically, the concentrator location problem is used to determine the number and location of concentrators and to allocate the terminals to these concentrators at minimum cost in a centralized computer network. When the number of

concentrators and their geographical locations are given, the problem then reduces to allocating the terminals to the existing concentrators only, which is known as the terminal assignment problem (Boorstyn & Frank, 1977; Kershenbaum, 1993). The formal definitions of these problems are given in Chapter 2.[1]

The concentrator location and terminal assignment problems are considered as very difficult problems.[2] In most cases, it is unlikely to find polynomial time algorithms to solve them. Hence, researchers have focused their efforts on developing heuristics that provide an approximate solution in a reasonable amount of time. This study aims to develop some efficient heuristic algorithms to solve these problems. The algorithms developed are based on simulated annealing which was introduced by Kirkpatrick, Gelat and Vecchi (1983). The results are compared to those given by some existing heuristics.

The rest of this chapter will give an introduction to centralized computer networks. In Chapter 2 the formal definitions of the problems being solved are given along with their complexity. In Chapter 3, some existing heuristics are briefly described. In Chapter 4, an overview of simulated annealing is presented. In Chapter 5, the algorithms that are developed in this study are described; Chapter 6 gives the details of their implementations. In Chapter 7, the computational experiments are described in detail. Finally, the result of the study and some future considerations are given in Chapter 8.

---

[1] See equations (2.1)-(2.4) for the concentrator location problems and equations (2.6)-(2.9) for the terminal assignment problems.
[2] See Chapter 2.

1.2 An Introduction to Centralized Computer Networks

Kershenbaum (1993) defined a centralized computer network as "a network where all communication is to and from a single site" (p.179). The simplest model of this network is a star topology, where each terminal is connected directly to the central computer by monopolizing a low capacity line. It is called a star because the central computer plays a crucial role in the network, and the network resembles a star with the central computer in the middle. For large networks, this topology can be improved upon in terms of cost efficiency by using concentrators between clusters of terminals and the central computer. The terminals are connected to a concentrator via low capacity lines, and each concentrator is connected to the central computer via a high capacity line.

As in general computer networks, a centralized computer network is constructed from a set of communication facilities, which includes the transmission media that interconnect locations on the network and a set of devices which are often known as network nodes in the general networks (Kershenbaum, 1993). The transmission media can be twisted pair cable, coaxial cable, optical fiber, terrestrial microwave, satellite microwave, and radio. Each of them has its own physical characteristics in terms of its capability in transmitting data. The different characteristics of these transmission media are beyond the scope of this study and the interested reader is referred to Stallings (1994) for a detail description of each media type. In this study the different characteristics of each transmission media type are simply represented in terms of costs.

Network nodes are devices that are used to construct a network. Throughout this study, there are three types of network nodes that are mainly used, i.e., terminals, central computers, and concentrators. The definitions of these nodes are given by Kershenbaum (1993, p.3), these are:

> *Terminals*: Simple devices, usually serving a single user; sources and destinations of low volume traffic. They usually include a keyboard and CRT and can also include disk drives and a printer. A personal computer, workstation, or a telephone may serve as a terminal.
>
> *Central Computers* (sometimes are referred to as *hosts* or *servers*): A large computer serving many users providing computing capability or access to a database. A Source and/or destination of a major amount of traffic. A large workstation might be a host.
>
> *Concentrators*: Devices which join the traffic on low speed lines into a single stream which can use a higher speed line.

The term *concentrator* in this study is used as a general term for all devices that function similarly with the definition given above, this includes multiplexers and cluster controllers.[3] It should be noted that all these simple definitions are given only to familiarize the reader with the concepts of this study. In real life however, the boundaries between these devices sometimes become blurred. For example, as mentioned above, a workstation can function both as a terminal and a central computer, similarly many of today's terminals can function as concentrators too (Kershenbaum, 1993).

Figure 1.1 shows an example of a centralized computer network. This example is known as a star-star topology, because all terminals are connected directly to concentrators and each concentrator is connected directly to the center. This topology can be refined by interconnecting onto one line concentrators in close

[3] A specific description of each of these devices can be seen in Ramos & Schroeder (1994)

proximity to one another in a hierarchical manner. Similarly, terminals can also be allowed to share a common line.[4] However, this study will focus only on the star-star topology for it is typically used as the starting point when designing more complex networks.

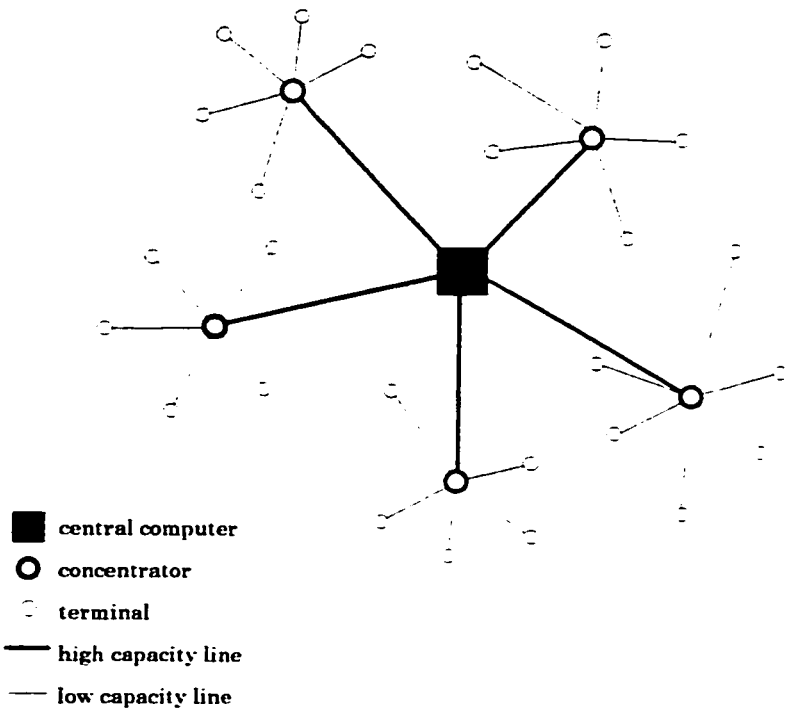


Figure 1.1. An example of a centralized computer network

In a star-star topology, three levels can be distinguished: level 1 is represented by the central computer, level 2 is made of the concentrators, and level 3 is made of

[4] If a line is shared by several terminals, it is called a *multidrop* (Stallings, 1994) or *multipoint* (Doll, 1978) line.

the terminals. A concentrator is generally subject to technological constraints on the amount of traffic it can manage. In addition, each terminal has its capacity requirement also known as the *weight* of the terminal. It denotes the amount of traffic exchanged between the terminal and the central computer. These constraints will determine the number of terminals a concentrator can control.

1.3 Summary

This chapter has presented the background and the purpose of this study along with an introduction to centralized computer networks. It also has briefly introduced the concentrator location and the terminal assignment problems. The next chapter gives the formal definitions of these problems.

CHAPTER 2

Formulation of the Problems

2.1 Problem Formulation

As mentioned in the previous chapter the main problem addressed in this study is the concentrator location problem for star-star topology networks where all terminals are connected directly to concentrators and each concentrator is connected directly to the central computer. This is known as the *star-star concentrator location problem (SSCP)* (Mirzaian & Steiglitz, 1981).

To simplify, one can assume the potential locations of the concentrators are known, and the number and locations of the terminals is given. The problem becomes one of selecting some locations of concentrators and a least cost assignment of terminals to the chosen concentrators that satisfies a set of additional constraints. Formally, the problem is to minimize the total cost Z given by

$$Z = \sum_{i=1}^{n} \sum_{j=0}^{m} c_{ij} x_{ij} + \sum_{j=1}^{m} d_j y_j \qquad (2.1)$$

subject to

$$\sum_{j=0}^{m} x_{ij} = 1, \qquad \forall_i \qquad (2.2)$$

$$\sum_{i=1}^{n} w_i x_{ij} \le k_j y_j, \quad \forall_j \tag{2.3}$$

$$x_{ij}, y_j \in \{0,1\}, \quad \forall_i, \forall_j \tag{2.4}$$

where

x_{ij} $= \begin{cases} 1, & \text{if terminal } i \text{ is connected to a concentrator at site } j \\ 0, & \text{otherwise} \end{cases}$

y_j $= \begin{cases} 1, & \text{if a concentrator is located at site } j \\ 0, & \text{otherwise} \end{cases}$

n = the number of terminals to be assigned

m = the number of potential locations of concentrators

c_{ij} = the cost of connecting terminal i to concentrator j

d_j = the cost of locating a concentrator at site j[1]

w_i = the weight of terminal i

k_j = the capacity of concentrator j

Thus, the first component of the objective function (equation 2.1) is the cost of assigning terminals to concentrators, whereas the second component is the cost of locating concentrators and connecting them to the central computer. Equation (2.2) is needed to make sure that every terminal is assigned to exactly one concentrator. Equation (2.3) ensures that the capacity constraint on each concentrator is not violated. Moreover, this model allows some terminals to be connected directly to the

[1] The value of d_j usually consists of the setup cost for the concentrator at site j and the cost for connecting that concentrator to the central computer

central computer as shown in equations (2.1) and (2.2) where j starts from 0 to m. In this case, the site 0 represents the central computer.

In some cases a specific set J of concentrators has already been chosen, therefore, the problem reduces to the allocation problem, which is known as the terminal assignment problem. Because a set of concentrators is chosen beforehand, the second component of equation (2.1), which is the cost of locating concentrators and connecting them to the central computer, can be removed from the objective function. The problem, then becomes one of minimizing the total cost Z given by

$$Z = \sum_{i=1}^{n} \sum_{j \in J} c_{ij} x_{ij} \tag{2.6}$$

subject to

$$\sum_{j \in J} x_{ij} = 1, \qquad \forall_i \tag{2.7}$$

$$\sum_{i=1}^{n} w_i x_{ij} \leq k_j, \qquad \forall_j \tag{2.8}$$

$$x_{ij} \in \{0,1\} \tag{2.9}$$

This problem usually occurs if one wants to add some terminals to an existing computer network.

2.2 Related Problems

The concentrator location problem, which is shown by equations (2.1) - (2.4), is also known as the *capacitated facility location problem*, where the concentrators represent the facilities and the terminals represent the demand nodes[2]. When the

[2] See Cornuejols. Nemhauser & Wolsey (1990) for the details of the location problems.

capacity constraints, i.e., equation (2.3), are relaxed, the problem becomes the *uncapacitated facility location* problem. The latter problem can further be modified by assuming that the network designers know in advance the number of facilities to be located. If p is the number of facilities, this modification can be written as

$$\sum_{j=1}^{m} y_j = p \qquad (2.5)$$

The problem then becomes the *p-facility location problem*. Moreover, if $d_j=0$ for all j, where d_j is the cost of locating a facility at site j, we then have the *p-median problem*. All of these problems are members of a family of location problems (Krarup & Pruzan, 1990).

Although this study only addresses the concentrator location problems and the terminal assignment problems, most of the other location problems mentioned above can be solved by using the algorithms developed in this study, especially the ones intended for solving the concentrator location problems.

2.3 Problem Complexity

Mirzaian and Steiglitz (1981) showed that almost all of the star-star concentrator location problems (SSCP) are strongly NP-Complete. However, if the capacity of the concentrators is less than two or all the connection costs are the same ($c_{ij}=c$, for all i and j), then these problems are solvable in polynomial time. They showed that except for these two cases, the concentrator location problems are NP-Complete (even for the special cases where the capacity of concentrators is three or unlimited).

Therefore in most cases, it is unlikely to find a polynomial time algorithm to solve these problems.

The terminal assignment problems in general are also considered to be difficult. However, if all terminals have the same weight (w_i=w, for all i), they can be solved in polynomial time; otherwise they are NP-complete (Abuali, Schoenefeld, & Wainwright, 1994). In cases where all terminals have the same weight, the alternating chain algorithm[3] can be used to solve the terminal assignment problems to optimality (Boorstyn & Frank, 1977; Kershenbaum, 1993). However, when the terminals have different weights, Kershenbaum (1993) states that "this complicates the implementations . . . so an optimal solution is no longer guaranteed" (p.215). Therefore he suggests to use the greedy algorithm, possibly with local exchanges to solve these kinds of problems.

2.4 Summary

This chapter has presented the formal definitions of the concentrator location and the terminal assignment problems. The definitions of some related problems such as the *uncapacitated facility location*, *p-facility location*, and *p-median* problems were also presented.

The chapter also showed that the concentrator location and the terminal assignment problems are NP-Complete. Therefore, finding a polynomial time algorithm to solve them to optimality is highly unlikely. The next chapter gives an overview of some well-known heuristic algorithms for solving them.

---

[3] See Chapter 3 for a description of the alternating chain algorithm

CHAPTER 3

An Overview of some Heuristics

In this chapter, some heuristic algorithms for solving the terminal assignment and the concentrator location problems are presented. As previously mentioned, the concentrator location problem is a special case of the facility location problems. Most of the algorithms for solving the facility location problems are also applicable to solve concentrator location problems (Boorstyn & Frank, 1977). Therefore some general facility location algorithms are presented too. The term *concentrator* and *terminal* in the concentrator location problems are equivalent with the terms *facility* and *demand*, respectively, in the general facility location problems. In this chapter these terms are used interchangeably. Some terminal assignment heuristics are presented in the first part and followed by facility (concentrator) location heuristics in the second part.

3.1 Heuristics for the Terminal Assignment Problems

3.1.1 Original Greedy Algorithm

The basic idea of this algorithm is to assign every terminal to the nearest concentrator. In the absence of the capacity constraint on concentrators, the assignment can be done without any difficulties. In the presence of capacity constraints, which is the most common problem in real-life, the assignment becomes difficult because of the possibility that some terminals cannot be assigned to the nearest concentrators. In the latter case, this algorithm will assign each terminal to the "best available"[1] concentrator. It means that if the remaining capacity of the nearest concentrator is less than the terminal weight, this algorithm will assign that terminal to the next best available concentrator. The result of this algorithm is a configuration in which every terminal is connected to a concentrator.

The terminal to be assigned first is the one with the smallest connection cost overall, followed by the one with the second smallest cost , the third smallest cost and so on, subject to the capacity constraint of every concentrator. This process continues until all terminals have been assigned or the algorithm fails to find a feasible solution.

[1] By "best", we mean *the least cost*. If the connection costs are determined only by the distances, then by "best", we mean *the nearest*.

3.1.2 Modified Greedy Algorithm[2]

The major advantages of the original greedy algorithm are that it is relatively easy to implement and requires little computing time. However, it has some disadvantages too. The most obvious one is that it tends to strand the terminals that are considered last. It is common that if some regions are not covered properly by concentrators with sufficient capacity, the last terminals considered might be assigned to concentrators that are very far away. Another serious problem is that this algorithm may easily fail to find a feasible solution even if feasible solutions exist. This is because the order of assigning terminals is based solely on the connection cost without considering the weight of terminals.

To deal with these problems, the original greedy algorithm can be modified as described in Kershenbaum (1993). The purpose of this modification is to give preference to the terminals that would suffer the most by not being connected to the nearest concentrators. These terminals will be referred to as critical terminals. Instead of using the connection cost as a criterion in choosing the order of assignments, a tradeoff function that reflects this preference is used. Let us say that c_{i1} is the cost of connecting terminal i to the first best available concentrator and c_{i2} is the cost of connecting terminal i to the second best available concentrator. Then a tradeoff function, t_i, can be constructed as:

$$t_i = c_{i1} - \alpha c_{i2} \qquad (3.1)$$

where α is a parameter between 0 and 1, reflecting the preference that is given to the critical terminals.

---

[2] It is also referred to as "the greedy algorithm with tradeoff α" (Abuali. et al.. 1994).

If the value of α is set to 0, it shows that no preference is given to the critical terminals and the criterion becomes the same as that of the original greedy algorithm, i.e., t_i is equal to the connection cost of the first best available concentrator. The bigger the value of α the more preference is given to the critical terminals. In the case where a terminal, due to its weight, can only be connected to one remaining available concentrator, the tradeoff value of that terminal has to be set to $-\infty$, so that it will be assigned right away.

3.1.3 Alternating Chain Algorithm

Kershenbaum (1993) classified this as a semi-greedy algorithm, which is based on the following facts :

1. All terminals should be assigned to their nearest best concentrators, except if the capacity constraints would be violated.
2. A terminal that has already been assigned to its best concentrator can be moved to another concentrator only if it will create room for another terminal which otherwise would have deviated farther.
3. If an optimal partial solution with p terminals exists, an optimal partial solution with $p+1$ terminals can be found by finding the least expensive way to add the $(p+1)$st terminal.

Based on these facts, one can start by trying to assign each terminal to its best concentrator. If all terminals can be assigned to their best concentrators, the optimal solution is found. If after assigning p terminals the remaining terminals cannot be assigned to their nearest concentrator due to the capacity constraints, it

will start looking for the least expensive way to add the $(p+1)$st terminal. In this state, the partial solution for p terminals is optimum.

In order to find the least expensive way to add the $(p+1)$st terminal, alternating chains[3] of all possibilities of adding new terminals and relocating some terminals already in the solution are constructed. An alternating chain with the least cost is the one to be chosen. Kershenbaum (1993) explained in detail how to implement this algorithm efficiently. If all terminals have the same weight, this algorithm guarantees that the optimal configuration can be found in polynomial time.

3.2 Heuristics for the Facility Location Problems

The algorithms presented in this section can be classified either as construction algorithms or improvement algorithms. The construction algorithms are those that attempt to build a good solution from scratch, whereas the improvement algorithms are those that try to improve an existing (initial) solution which is usually obtained from a construction algorithm.

Most research on the performance of heuristic facility location algorithms reached the conclusion that the exchange (interchange) heuristic is the most robust heuristic algorithm (Densham & Rushton, 1992). Kuehn and Hamburger (1963) were the pioneer of this kind of heuristics. Their heuristic consists of two parts. They named the first part as the *main program* and the second one as the *bump and shift routine.* What they called the *main program* now is known as the *ADD*

[3] Also known as *augmenting paths* (Kershenbaum. 1993).

algorithm, which is one of the most widely used construction algorithms; the *bump and shift routine* is known as the *exchange or interchange algorithm* which is the basis of most improvement algorithms (Cornuejols, et al., 1990).

There are some recent books that discuss some location facility heuristics in detail. Kershenbaum (1993) and Daskin (1995) are among these. All of the algorithms discussed in this section are taken from these two books. Kershenbaum presents three construction heuristic algorithms that were developed specifically to solve concentrator location problems. Those are *COM*, *ADD*, and *DROP* algorithms. Daskin describes some algorithms with more general applications to facility location problems. As construction algorithms he presents *ADD* and *DROP* and as improvement algorithms he includes *neighbourhood search* and *exchange* algorithms.

3.2.1 Center of Mass (COM) Algorithm

The Center of Mass (COM) algorithm is described in detail in Kershenbaum (1993). The basic idea of this algorithm is to identify the natural cluster of traffics. One starts by assuming that each terminal is in a cluster by itself, and then creates a new cluster by combining two clusters that are close to each other subject to some given constraints. The constraints can be a desired weight for a cluster, a distance limit between two clusters to be combined or a desired number of clusters..

Let us assume that for each terminal i, we have its coordinates, (x_i,y_i), and weight, w_i. If terminals i and j are to be combined, then a new cluster formed with

these two terminals is represented by their center of mass, (x_k, y_k), which can be calculated as follows:

$$x_k = \frac{w_i x_i + w_j x_j}{w_i + w_j} \tag{3.2}$$

$$y_k = \frac{w_i y_i + w_j y_j}{w_i + w_j} \tag{3.3}$$

The weight of the cluster k, w_k, is the summation of w_i and w_j. For further consideration, terminal i and j are removed from the calculation and replaced by cluster k. This algorithm stops if no new clusters can be formed that satisfy the given constraints.

3.2.2 ADD Algorithm

This is a greedy algorithm. At the beginning all terminals are connected to the center. This presupposes that the capacity of the center is large enough to accommodate all terminals. Then, every potential concentrator is examined: one computes the savings that can be made if it is added to the configuration. The concentrator that can save the most money is selected first. This algorithm stops when the addition of a new concentrator will not result in any savings.

The savings obtained by adding a new concentrator, say, j, to the current configuration are the difference between the savings obtained by moving a few terminals from concentrators currently in the configuration to it and the cost of the concentrator j itself. The number of terminals that will be moved to the new concentrator depends on the capacity of the concentrator j. The savings namely, s_j, can be written as

$$s_j = \sum_{i \in I(j)} \left(c_i^I - c_{ij} \right) - d_j \qquad (3.4)$$

where

c_{ij} = the cost of connecting terminal i to concentrator j

c_i^I = the cost of connecting terminal i to the concentrator with which it currently associated

d_j = the cost of locating concentrator j

$I(j)$ = the set of terminals for which one can save money by moving them to concentrator j, subject to the capacity of concentrator j.

3.2.3 DROP Algorithm

This is another greedy algorithm, that works in the reverse direction of the ADD algorithm. At the beginning all possible sites of concentrators are considered in use. In the case that there are no capacity constraints on concentrators, the terminals are simply connected to their nearest concentrator. The algorithm then investigates each concentrator to find out which one will bring the most savings if it is dropped from the configuration. The algorithm stops if it no longer finds a concentrator whose removal will save some money.

In the presence of capacity constraints, evaluating each concentrator to be dropped will involve solving a terminal assignment problem, because some terminals may not be connected to their nearest concentrators. Hence at the beginning, this algorithm will solve a terminal assignment with all concentrators present. Then in order to evaluate each potential drop, it will need to solve another

terminal algorithm problem with a specific concentrator removed. This will cause the running time of this algorithm to be extremely large. Therefore, Kershenbaum (1993) points out that for the capacitated case it is better to tackle the problem using the ADD algorithm because the quality of the solutions obtained from the ADD and DROP algorithms are comparable, while the running time of the DROP algorithm is much larger.

3.2.4 Neighbourhood Search Algorithm

Daskin (1995) presents this algorithm for solving general facility location problems, especially the uncapacitated cases. It also assumes that the demand's sites[4] are the only possible places for the facilities[5] to be located. Moreover, there is no cost associated with locating a facility. These assumptions are common for the p-median problems.

It starts with a given initial solution, which may be calculated with one of the construction algorithms. Let us say that the initial solution consists of p clusters of demand nodes, where one facility is located in each cluster.[6] The nodes within a cluster are referred to as the neighbourhood of the facility in that cluster. At first, one has to make sure that each demand node is assigned to its best (least cost) facility. Since the facilities are uncapacitated, it is feasible to do so. Then, the algorithm will check if the facility in each cluster is optimally located. If it is not, the facility has to be relocated in one of the nodes in that cluster to obtain the optimal

[4] The term *demand* is equivalent to the term *terminal* in the concentrator location problem.
[5] The term *facility* is equivalent to the term *concentrator* in the concentrator location problem.
[6] In the concentrator location problem, it means that p concentrators are used.

solution for that specific cluster or neighbourhood. This is known as 1-median problem, which can be solved to optimality (Daskin, 1995).

Whenever there is a change in the location of any facility, all assignments have to be reexamined to make sure that each demand node is assigned to its best facility. If there are some nodes that are moved from one cluster to another, the locations of the facilities in the clusters that are involved in the exchanges have to be recomputed because the neighborhoods have changed. This algorithm stops when there is no change in any neighborhood anymore.

3.2.5 Exchange Algorithm

Daskin (1995) gives an implementation of the exchange algorithm for solving the uncapacitated case of facility location problem. However, it is also applicable for solving the capacitated cases. The main idea of this algorithm is to improve an initial solution obtained using a construction algorithm by exchanging a facility that is not in the solution for a facility that is in the solution. One starts with a given initial solution obtained either from the ADD or DROP or another construction algorithm. This algorithm then searches the best exchange that can be made for each facility in the solution and takes the best of them. In other words, if there are m facilities in the solution, there will be m best possible exchanges, one for each facility. This algorithm chooses the best of them for further consideration. If the chosen exchange can reduce the total cost then it is accepted and all demands are reassigned to their nearest facilities. This algorithm stops if exchanging any of the facilities in the solution with any facilities not in the solution does not produce any further total cost reductions.

The main steps of this algorithm can be summarized as follows:

1. Get an initial solution
2. Find the best replacement site for each facility site in the solution.
3. Find the best pairs overall from step 2.
4. If the exchange obtained in step 3 can reduce the total cost then
   - do the exchange
   - reassign all demands to their nearest facilities
   - go to step 2

   Otherwise, stop the algorithm.

3.2.6 Improvement to the Exchange Algorithm

Daskin (1995) proposed a further improvement to the exchange algorithm. The improvement is aimed at overcoming the dependency toward the initial solution in terms of the number of facilities to be located. It is clear that if the number of facilities given by the initial solution is suboptimal, then the solution that can be obtained by the exchange algorithm will be suboptimal too. To overcome this problem, the combination of the ADD and DROP algorithms are used.

Thus after an exchange is made, the ADD and DROP algorithms are applied separately to see how much savings each of them can produce. If both of them can save money, the one that can save the most is chosen, and then the algorithm continues trying to exchange another pair of facilities again. This algorithm stops when either adding, dropping or exchanging concentrators cannot generate additional savings.

3.3 Summary

This chapter has given an overview of some well-known heuristic algorithms for solving the terminal assignment and the concentrator location problems. The next chapter presents an overview of simulated annealing which is the framework used in the algorithms developed in this study.

CHAPTER 4

An Overview of Simulated Annealing

4.1 Introduction

The iterative improvement algorithm as shown in Figure 4.1 is the most widely used framework in solving combinatorial optimization problems. The obvious advantage of this approach is that it is generally applicable and flexible. Unfortunately, very often the methods based on this approach get stuck in a local but not global optimum. Essentially, this is because they behave as do greedy algorithms, i.e., they only accept new configurations that reduce the total cost. Methods based on this approach are known as simple local search methods. The quality of the solutions obtained by these algorithms depends mostly on the initial solutions. Unfortunately, there are no guidelines available for determining how to choose an initial solution. Moreover, for many problems, the upper bound on computational time is not known (Aarts & Korst, 1989).

```
procedure IterativeImprovement; {for minimizing a function}
begin
    Initialize(InitialConfig)
    CurrentConfig := InitialConfig;
    repeat
        Generate(CandidateConfig ∈ Neighborhood of CurrentConfig);
        ΔCost := Cost(CandidateConfig) - Cost(CurrentConfig);
        if ΔCost < 0 then
            CurrentConfig   := CandidateConfig;
    until  Cost(CandidateConfig) - cost(CurrentConfig) ≥ 0, for all
           CandidateConfig in the neighbourhood of CurrentConfig
end;
```

Figure 4.1. Iterative improvement algorithm in pseudo-PASCAL

However, this approach provides some advantages in terms of applicability and flexibility. The information needed to use this approach consists only of the solution specification, a cost function, and a neighbourhood structure. All of these are not difficult to construct and can be straightforwardly specified for most problems. This is why modified iterative improvement algorithms are still widely used.

Aarts and Korst (1989) present three modification alternatives to improve the iterative improvement method. First, one can apply iterative improvement method with a large number of initial solutions, so that many such local minima can be found, then choose the best of them as the final result. Second, in order to increase the possibility of finding the global minimum, one can enlarge the neighbourhood of a solution, but this inevitably leads to an increase in the complexity of the problem's solution. Third, one can allow, with a certain probability, to accept a new solution that increases the cost in order to avoid getting caught too early in a local minimum.

The simulated annealing algorithm is one of the methods that follows the third alternative.

4.2 The Simulated Annealing Algorithm

4.2.1 Generic Algorithm

The simulated annealing algorithm is based on the strong analogy between its behavior and that of the physical annealing processes of solids. Annealing, in condensed matter physics, denotes a thermal process to get low energy states of a solid in a heat bath. There are two steps involved in this process. First, the temperature of the heat bath is increased up to a maximum at which all particles of the solid arrange themselves randomly in the liquid phase. In a second step the temperature of the heat bath is decreased slowly until the particles solidify. If the temperature is decreased slowly enough, the particles tend to solidify in a structure of minimal energy (Van Laarhoven & Aarts, 1987).

Metropolis, Rosenbluth, Rosenbluth, Teller & Teller (1953) introduced an algorithm to simulate the physical annealing process for a given temperature, T, which is known as the Metropolis Procedure. It works as follows: Given a current state i of the solid with energy E_i, a small perturbation of the current state is made to generate a new state j with energy E_j. If E_j is less than or equal to E_i, the new state j is accepted as the current state. Otherwise, the new state j is accepted with a certain probability, say, $P_{w'}$, as given by

$$P_{w'} = exp\left(-\frac{E_j - E_i}{k_b T}\right) \tag{4.1}$$

where k_b is a physical constant known as Boltzmann Constant. In order to allow the solid to reach thermal equilibrium, this algorithm has to generate a large number of transitions at each temperature.

Kirkpatrick, Gelatt and Vecchi (1983) generalized this algorithm to solve combinatorial optimization problems. They showed that by replacing states in the physical system by configurations and the energy function by a cost function, the Metropolis procedure can be used directly to solve combinatorial optimization problems. In this case, the temperature becomes simply a control parameter in the same unit as the cost function.

If we are given a configuration called *CurrentConfig*, another configuration, say, *CandidateConfig*, can be formed by choosing it randomly from the neighbourhood of *CurrentConfig*. The notion of the neighbourhood in this case corresponds to the small perturbation in the Metropolis procedure. Let us say that $\Delta Cost$ is the difference between the cost of *CandidateConfig* and that of *CurrentConfig*, and *temp* denotes the temperature. By assuming k_b=1, the equation (4.1) can be modified to be:

$$P_w = exp\left(-\frac{\Delta Cost}{temp}\right) \qquad (4.2)$$

Figure 4.2 shows the pseudo-PASCAL of the generic simulated annealing algorithm. L_k and $temp_k$ denote the number of transitions and the temperature value, respectively, generated at the k^{th} iteration. Initially the value of $temp_0$ should be set high enough so that the probability of accepting a configuration worse than the current one during the early stages of the process is high. This enables the algorithm not to get caught too early in a local minimum. As the value of $temp_k$

approaches zero, the probability of accepting worse configurations will approach zero making simulated annealing similar to the iterative improvement algorithm. In the

```
procedure SimulatedAnnealing; {for minimizing a function}
begin
    Initialize (InitialConfig, L0,temp0);
    CurrentConfig := InitialConfig;
    k := 0;
    repeat
        for l := 1 to Lk do
        begin
            Generate(CandidateConfig ∈ Neighborhood of CurrentSolution);
            ΔCost := Cost(CandidateConfig) - Cost(CurrentConfig);
            if ΔCost < 0 then
                CurrentConfig   := CandidateConfig;
            else
                if exp(-ΔCost/tempk) > random(0,1) then
                    CurrentConfig := CandidateConfig;
        end;
        k:=k+1;
        Calculate(Lk);
        Calculate(tempk);
    until StoppingCriterion is true;
end;
```

Figure 4.2. Generic simulated annealing algorithm in pseudo-PASCAL

context of a minimization problem, as the temperature becomes very small only configurations yielding lower costs will be accepted. Therefore, while the simulated annealing algorithm inherits the advantages of the iterative improvement method which are generally applicable and flexible, it also overcomes the disadvantage of the local search algorithm, which gets stuck early on in the vicinity of some local minima.

Van Laarhoven and Aarts (1987) provide theoretical analysis of the simulated annealing algorithm. They show that under certain conditions on the initial temperature, the number of iteration at each temperature, the rule for decreasing the temperature and the stopping criteria, the result obtained with the simulated annealing algorithm converges to the global optimum with a probability of one. One of the following conditions has to be satisfied to guarantee the global optimum:

1. For each value of the control parameter $temp_k$, an infinite number of transitions is generated, i.e., $L_k \rightarrow \infty$, and $lim_{k\rightarrow\infty} temp_k = 0$.
2. For each value $temp_k$ one transition is generated and $temp_k$ approaches zero not faster then $\frac{\Gamma}{\log(k)}$, where Γ is a constant.

Algorithms that employ the first condition are known as *homogeneous algorithms*, and those that employ the second one are known as *inhomogeneous algorithms*. Practically, both of these conditions are almost impossible to be satisfied because they require an amount of computation time that is infinitely large. Therefore, most of the current applications of the simulated annealing algorithm use some sort of approximation. Consequently, there is no guarantee anymore that the configuration representing the global minimum will be found with these applications.

4.2.2 Implementing the Algorithm

The simulated annealing algorithm is not a completely specified algorithm. It is a generic algorithm that needs some decisions to be made before it can be applied to solve a specific problem. These decisions can be divided into two groups, namely,

generic and problem specific decisions. The first group consists of the decisions about: (1) the initial temperature, (2) the number of iterations at each temperature,[1] (3) the rule for decreasing the temperature and (4) the stopping criteria. The values given to these parameters make up the annealing schedule. The second group of decisions which are problem specific consists of: (1). How to formulate the problem, (2) How to calculate the initial solution, (3) How to generate a neighbour and (4) How to evaluate the cost. In this section only the first group will be discussed in more details. Because the latter group is problem specific, it will be discussed separately in Chapter 5.

The chosen annealing schedule will determine the convergence of the algorithm to the optimal solution. However, as mentioned in the previous section, an amount of computation time that is infinitely large is needed both for the homogeneous and inhomogeneous algorithms to ensure finding an optimal solution. Therefore, some sort of approximation that provides a finite-time annealing schedule is preferred in practice. There exists a variety of approximation schedules proposed in the literature. The reviews of most of these schedules can be found in Van Laarhoven & Aarts (1987) and Eglese (1990).

One class of annealing schedules that is widely used is the so-called "conceptually simple annealing schedule." This is called a simple schedule because all decisions in this class are based only on empirical rules rather than theoretical results (Aarts & Korst, 1989).[2] Most of the annealing schedules in this class are based on the one that was proposed by Kirkpatrick (1984).

---

[1] It is also referred to as the length of Markov chains in the literature.
[2] For the more elaborate and theoretically based schedules see Van Laarhoven & Aarts (1987).

The value of the initial temperature, $temp_0$, in most of the simple schedules is set beforehand to be high enough so that almost all cost increasing transitions are accepted. Kirkpatrick (1984) proposed the following rule of thumb to obtain the initial temperature:

> One first finds the "melting temperature" by starting at arbitrary temperature, attempting a few hundred moves, and determining the fraction of the moves which are accepted. If that fraction is less than, say, 80%, the temperature is doubled (p.978).

A more elaborate way to calculate an initial temperature is derived directly from equation (4.2) which was proposed by Johnson et al. as cited in Van Laarhoven and Aarts (1987)[3] as follows:

$$temp_0 = \frac{\overline{\Delta Cost}^{(+)}}{ln(P_{u'}^{-1})} \qquad (4.3)$$

The notation $\overline{\Delta Cost}^{(+)}$ denotes the average increase in cost of all cost increasing transitions generated. This can be approximated by generating a number of random transitions, recording all $\Delta Cost$s that are greater than zero, and then computing the average of the $\Delta Cost$s. $P_{u'}$ denotes the probability of accepting a worse configuration which has to be determined beforehand. Osborne and Gillett (1991), for example, set $P_{u'}$=0.3.

A simple way to determine the number of iterations at each temperature, say, L_k, is by setting it to be the same at every iteration ($L_k = L$, for each k). The value of L usually depends polynomially on the size of the problem being solved (Van Laarhoven & Aarts, 1987). The more elaborate approach is that the value of L_k is not fixed but consists of accepting a sufficient number of transitions[4] subject to a

---

[3] p. 60

[4] Osborne and Gillett(1991) considered the cost increasing transitions only in their algorithms.

constant upperbound (Kirkpatrick, et al., 1983). The upperbound is usually chosen polynomially relative to the problem size.

The rule for decreasing the temperature that is widely used is the following:

$$temp_{k-1} = \alpha * (temp_k) \qquad (4.4)$$

where α is a constant smaller than but close to 1. In practice, the typical value of α is in between 0.80 and 0.99 (Eglese, 1990).

A simple choice for the stopping criterion is by fixing the final value of $temp_k$, say, ε, or by limiting the number of temperature steps. Chardaire and Lutton (1993), for example, set the number of temperature steps to twenty five. Another proposed method is that the algorithm stops after the configuration does not change for a certain number of consecutive temperature changes (Kirkpatrick, et al., 1983).

In practice, however, most of the proposed schedules still have very long running-times. If one tries to shorten the running-time, one usually ends up with a poor result (Eglese, 1990). Therefore, some researchers tried to make other modifications toward the algorithm. The simplest modification that can be made is to store the best solution found so far. This modification is easy to implement and will most likely produce better results than those obtained by the original algorithm for the same running-times as shown in some research reviewed by Eglese.[5]

4.2.3 Some Applications of the Simulated Annealing Algorithm

Aarts and Korst (1989) list a number of applications of simulated annealing in the various areas of combinatorial optimization problems. Traveling Salesman Problems and VLSI Design are the two main areas where a large number of

[5] See Eglese (1990) for other possible modifications

simulated annealing applications were developed. Other areas include Graph Partitioning, Quadratic Assignment, Linear Arrangement, Graph Colouring, Scheduling, Matching, Facility Layout, Image Processing, Code Design, Biology and Physics.

Some other applications are collected by Vidal (1993). One of them is the application of simulated annealing to concentrator location problems, which was developed by Chardaire and Lutton.[6] This is the only application of simulated annealing in this area that we are aware of. The application was developed to solve the star-star concentrator location problem, which is similar to our study. It runs under the UNIX operating system.

The annealing schedule used is a variant of the simple one. The initial temperature was set high enough so that about half of the cost increasing transition attempts were accepted. The stopping criterion was based on k (the number of temperature steps), which was set to be around 25. Based on this value, they found that the value of α was 0.91. The number of transitions to be executed at each k was slightly increased for every temperature decrease. The increasing factor was between 1 and 1.05.

In the study, three kinds of transformations were applied to generate the neighbours of a solution. These were additions, deletions and displacements of concentrators. These transformations are similar to the ones used in our study, except in the way that they were implemented. In the study being reviewed, the authors only considered the concentrators that were adjacent to the one being removed when reassigning its terminals to the remaining concentrators. The same

---

[6] pp. 175-199

principle was used to select terminals to be moved to a newly added concentrator. They did this in order to reduce the program's running-time. However, we intuitively believe that this strategy hindered their final results. The effects of this strategy can be very serious if the capacity constraints are tight, since most likely the initial configuration contains some terminals that are connected to concentrators that are far away from them. Moreover, if the capacity constraints are tight, even considering all concentrators might not be sufficient. In this case, the terminal assignment procedure has to be applied.

The authors compared their application with an implementation of the Lagrangian relaxation developed by Cournuéjols, et al.,[7] which they considered to be the best relaxation method developed thus far. They found that if the number of terminals is more than two hundreds, their method almost always generated better solutions with less CPU time.

4.3 Summary

This chapter has presented the concepts of the simulated annealing algorithm. Recall that the simulated annealing algorithm is a generic algorithm which can be used to solve a large number of combinatorial optimization problems. In order to solve a specific problem, some features of this algorithm must be specified beforehand. The next chapter discusses how this features are specified in this study to develop specific algorithms for solving the terminal assignment and concentrator location problems.

[7] As cited in Chardaire & Lutton (1993).

CHAPTER 5

Design of Algorithms for the Problems of this Study

As mentioned in the previous chapter[1] there are some features that must be specified before the simulated annealing algorithm can be used in solving a specific problem. These are the annealing schedule, the problem formulation, the initial solution, the neighbour generation method and the configuration cost evaluation method. The problem formulation has been described in Chapter 2. The cost evaluation is carried out by calculating all connection costs of a configuration as shown in equation (2.6) for the terminal assignment problem and equation (2.1) for the concentrator location problem. In this chapter we address the issues related to obtaining an initial solution, generating a neighbour and choosing an annealing schedule for each algorithm that we developed.

[1] See section 4.2.2

5.1 Obtaining an Initial Solution and Generating a Neighbour

This section discusses the methods that are developed in this study for obtaining an initial solution and generating a neighbour. The discussion starts within the terminal assignment context and moves to the concentrator location problem.

5.1.1 Terminal Assignment

5.1.1.1 Obtaining an Initial Solution

An initial solution for the terminal assignment problem is calculated based on the modified greedy algorithm as described in Chapter 3[2]. This algorithm is chosen because while it overcomes some weaknesses of the original greedy algorithm, it is still relatively easy to implement and requires little computing time. The main steps of this modified algorithm are as follows:

step 1: Find the best available concentrator for each terminal that has not yet been assigned. If there is a terminal that cannot be fitted to any concentrator, terminate the process and report the failure; otherwise proceed to the next step.

step 2: Find the second best available concentrator for each terminal, if it exists.

step 3: Calculate the tradeoff value for each terminal:

- If a terminal does not have a second best available concentrator, set the tradeoff value to $-\infty$, otherwise calculate the tradeoff value according to equation (3.1).

[2] See section 3.1.2

step 4: Compare all results obtained in step 3, and choose the smallest overall.

step 5: Do the assignment for the terminal and concentrator obtained in step 4.

step 6: If all terminals have already been assigned (success) or the remaining terminals cannot be fitted into the remaining capacity of any concentrators (failure) then terminate the process, otherwise go back to step 1

It should be noted that although this modification can improve the quality of the solution given by the original greedy algorithm,[3] it cannot guarantee that a feasible solution can be found for every problem that has feasible solutions. Kershenbaum (1993) listed three cases where a feasible solution is not found:

1. The total capacity of concentrators is less than the total weight of terminals.
2. The total capacity of concentrators is equal to or more than the total weight of terminals, but there is no way to assign all the terminals. For instance, there are four terminals, each with weight 2 to be assigned to three concentrators each having a capacity of 3. Even though the total weight of the terminals is less than the total capacity of the concentrators, no feasible solutions exist.
3. The total capacity of concentrators is equal to or more than the total weight of terminals and feasible solutions exist, but the algorithm fails to find one.

The third case is the one that we want to avoid. However, except for the first case, it is hard to detect early on in the execution of the algorithm if the problem being solved has feasible solutions or not.

[3] See section 3.1.2

If the algorithm fails to find a feasible solution, and we suspect that this failure belongs to the third case, a dummy concentrator that can accommodate all unassigned terminals can be introduced to the solution, and this can be used as an initial solution for the simulated annealing algorithm. The connection costs between this dummy concentrator with all terminals have to be set very high to discourage establishing these connections. Thus, we can expect that all terminals that are connected to this dummy concentrator will move to the real concentrators as the simulated annealing algorithm proceeds.

5.1.1.2 Neighbour Generation

In general, a neighbour of a configuration is generated by making a small change to the current configuration. Therefore, at first, the meaning of *a small change* should be defined. In this study a small change is defined as either moving a terminal from one concentrator to another or swapping two terminals from two different concentrators. Having clarified the meaning of a small change, the next step is to determine how to do it. Based on the definition of a small change, there are two questions that should be answered before it can be implemented. First, the question of what action is to be carried out; whether simply moving a terminal or swapping two terminals. Second, the question of which terminals to be moved or swapped. Two methods are tried here.

The first method, named *CRS (completely random selection)*, consists of randomly choosing both the action to be performed and the terminals involved. One starts by choosing two concentrators randomly, and then a terminal is randomly chosen from each of them. However, we must allow simple moves to happen, and

also address questions such as the possible emptiness of both concentrators. In order to allow both simple moves and swaps and to prevent the occurrence of the problems we have just mentioned, some modifications are made.

The first modification is that the first concentrator to be chosen has to have at least one terminal connected to it. This however, is not a necessity for the second concentrator. This restriction is needed to ensure that at least one terminal is chosen so that there is at least one action that can be accomplished. Having chosen two concentrators, a terminal is then chosen randomly from the first concentrator. After that, a random number between 1 and $(p+q)$, inclusively, is generated, where p is the number of terminals being connected to the second concentrator and q is the number of terminals than can be added to it. The value of q is computed based on the remaining capacity on the second concentrator, called R_2, and the weight of the chosen terminal from the first concentrator, called w_1. The formula to compute q can be written as follows:

$$q = \text{trunc}(R_2/w_1) \qquad (5.1)$$

where trunc is the function to truncate a real value to the largest integer value that is less than or equal to it.

The value of the random number generated will determine what action is to be done. If it is greater than p, simply move the terminal that is chosen from the first concentrator to the second concentrator. If it is between 1 and p, inclusively, the action to be done is swapping and the value of the random number represents the terminal number on the second concentrator that will be swapped with the chosen terminal from the first concentrator. However, before the swapping is made, we have to check whether it is possible or not.

Let us say that the remaining capacity on the first and the second concentrators are R_1 and R_2, respectively. The chosen terminal from the first concentrator is t_1 and its weight is w_1, and the chosen terminal from the second concentrator to be swapped with t_1 is t_2 and its weight is w_2. If t_1 is moved out, the remaining capacity on the first concentrator becomes R_1+w_1. This means that the weight of t_2 has to be at most R_1+w_1. Moreover, in order to make enough room for t_1 on the second concentrator, R_2+w_2 has to be bigger that w_1. Therefore, w_2 has to be bigger than or equal to $(w_1- R_2)$ but less than or equal to (R_1+w_1). This can be written as:

$$(w_1- R_2) \le w_2 \le (R_1+w_1) \qquad (5.2)$$

If inequality (5.2) cannot be satisfied by the chosen terminals, the algorithm will start again from the beginning by choosing another pair of concentrators. The main steps of this approach are as follows:

step 1 : Choose randomly a concentrator that at least has one terminal being connected to it.

step 2 : Choose another concentrator randomly.

step 3 : Choose a terminal randomly from the first concentrator.

step 4 : Generate a random number between 1 and $(p+q)$, inclusively.

step 5 : If the random number obtained in step 4 is bigger than p, simply move the terminal obtained in step 3 to the second concentrator.

Otherwise, if inequality (5.2) can be satisfied then swap the terminal, which is identified by the random number generated from the second concentrator with the one obtained in step 3; otherwise go to step 1.

The second method, named *BRS* (*best random selection*), is a modification of the first one. This time, the action to be done is not chosen randomly but depends on the "savings" that can be made. If there are no actions that can save money, the one with the minimum cost increasing effect is chosen. One starts by randomly choosing two concentrators. Similarly to the *CRS* approach, the first concentrator must have at least one terminal connected to it. Then, all possibilities of moving a terminal from the first concentrator to the second one and swapping two terminals, one from each concentrator, are examined.

Moving a terminal from the first concentrator to the second is possible only if there is enough room available on the second concentrator to accept the terminal being moved. Swapping terminals is possible only if both concentrators have terminals connected to them and inequality (5.2) is satisfied.

The main steps of this second approach are as follows:

step 1 : Choose randomly a concentrator that has at least one terminal connected to it.

step 2 : Choose another concentrator randomly.

step 3 : If there is room available on the second concentrator obtained in step 2, evaluate which terminals from the first concentrator can fit into it and select the one offering the most saving.

step 4 : If both the first and the second concentrators have terminals connected to them, calculate which pair of terminals, one from each concentrator, that satisfy inequality (5.2), can save the most if they are swapped.

step 5 : Compare the savings obtained in Step 3 and 4, and choose the best one.

step 6 : Do the action according to the result obtained in step 5.

5.1.2. Concentrator Location

5.1.2.1 Obtaining an Initial Solution

The initial solution for the concentrator location problem in this research is calculated by using the ADD Algorithm as described in Chapter 3. The main steps of this algorithm can be summarized as follows:

step 1 : Assign all terminals to the center and set it as the current configuration.

step 2 : For every concentrator not in the current configuration, examine the savings obtainable if it is added to the configuration according to equation (3.4) :

- compute the saving that can be obtained if a given terminal is moved to the concentrator being considered. Repeat for each terminal.
  - If this saving is positive consider moving the terminal in question to the concentrator being examined; otherwise just ignore it.
  - Sum up the savings obtainable for each concentrator subject to its capacity. If the total weight of terminals exceeds the capacity of the concentrator then take the first terminals that can save the most money without violating the capacity constraint of the concentrator being examined.
- Subtract the cost of locating concentrator j from the savings obtained in the previous step to obtain the actual savings.

step 3 : If the actual savings exist

- Add the concentrator that can save the most money to the current solution.

- Move the terminals for which money can be saved to the new concentrator.
- Go back to step 2.

Otherwise, terminate the process.

The results given by the ADD algorithm are not optimum most of the time, because once a concentrator is chosen, it cannot be dropped from the configuration at a latter stage. It follows that if a bad decision is made at some point, it must be carried to the end. To get a better result, some modifications should be made such as allowing a previously chosen concentrator to be dropped or to be replaced by a concentrator not present in the current solution. One way to do this is by using simulated annealing (see Chapter 4).

5.1.2.2 Neighbour Generation

In keeping with the above observation, given a configuration, a neighbour solution can be obtained in three ways, i.e., (1) dropping a concentrator, (2) adding a new concentrator, or (3) swapping a concentrator currently used with a concentrator that is not being used. As a matter of fact, swapping concentrators can be seen as a combination of dropping and adding processes. In this research, two methods for dropping, and two methods for adding concentrators, are investigated.

5.1.2.2.1 Drop Procedure

The two methods for dropping a concentrator are *RC* (the concentrator that is dropped is randomly chosen) and *LS* (drop the least significant concentrator, i.e., the

one that has the least contribution in lowering the configuration cost). The contribution of each concentrator to lowering the cost of the current configuration, if dropped, is calculated by using the following formula:

$$Q_j = \sum_{i \in I(j)} \left(c_i^{\bullet} - c_{ij} \right) - d_j \qquad (5.3)$$

where

Q_j = the contribution of concentrator j

$c_i^{\bullet}$ = cost of connecting terminal i to its best available concentrator other than concentrator j

c_{ij} = cost of connecting terminal i to concentrator j

d_j = cost of locating concentrator j

$I(j)$ = all terminals currently connected to concentrator j

Thus, Q_j is the amount of money that will be lost if concentrator j is dropped from the configuration and each terminal that was previously connected to it is moved to its best available concentrator. Equation (5.3) is similar to equation (3.4) used by the ADD Algorithm to evaluate which concentrator is to be added to the configuration. The smaller the value of Q_j, the less significant the contribution of concentrator j. If Q_j is negative, it means that dropping concentrator j will save some money. Therefore, the concentrator with the smallest value of Q_j is the one we consider to be dropped.

After a concentrator is dropped, all terminals that previously were connected to it have to be reassigned to the remaining concentrators in the configuration based on their connection costs. In other words, this procedure will reassign these terminals to their best available concentrators.

The main steps of the Drop Procedure can be summarized as follows:

step 1 : Drop a concentrator from the configuration, either by using *RC* method or *LS* method.

step 2 : Reassign each terminal that was connected to the concentrator dropped in step 1 to its best available concentrator in the configuration.

5.1.2.2.2. Add Procedure

The concentrator to be added to the configuration is chosen from all concentrators not currently in the configuration. Two methods of choosing the concentrator are investigated in this research; these are, *RC* (the concentrator to be added is randomly chosen) and *COM* (the concentrator to be added is the closest to the center of mass).

The center of mass (X_{COM}, Y_{COM}) is calculated as follows:

$$X_{COM} = \frac{\left(\sum_{i \in J(j)} x_i\right) + x_0}{n_{J(j)} + 1} \tag{5.4}$$

$$Y_{COM} = \frac{\left(\sum_{i \in J(j)} y_i\right) + y_0}{n_{J(j)} + 1} \tag{5.5}$$

where (x_i, y_i) are the coordinates of the terminals currently connected to the concentrator j, (x_0,y_0) denotes the central computer's coordinate and $n_{I(j)}$ is the number of terminals connected to concentrator j. The concentrator to be chosen is the one that is the closest to (X_{COM}, Y_{COM}). This *COM* method is applicable only if the input data are the locations or coordinates of the terminals and concentrators. If the input data is a cost matrix, only the first method (*RC*) can be used.

After a concentrator is added to the configuration, we have to select the terminals to be moved into it. Those terminals have to be the ones that can save the most by moving to this new concentrator. Therefore, each terminal has to be investigated to see how much money it can save. The savings, s_i, can be calculated using the following formula:

$$s_i = c_i^I - c_{ij} \qquad (5.6)$$

where

c_i^I = the cost of connecting terminal i to the concentrator with which it is currently associated.

c_{ij} = the cost of connecting terminal i to the new concentrator j.

The positive s_i indicates that it is better to move terminal i to the new concentrator. If the total weight of the terminals with positive s_i exceeds the capacity of the new concentrator, only the first terminals that can save the most without violating the capacity constraint are moved.

The main steps of this Add Procedure are as follows:

step 1 : Add a concentrator into the configuration either by using *RC* or *COM*.

step 2 : For each terminal, calculate the obtainable savings if it is moved to the new concentrators according to equation (5.6).

step 3 : Move the terminals that can save money:

- If the total weight of the terminals that can save money exceeds the capacity of the new concentrator then move the first terminals that can save the most.

5.1.2.2.3 Swap Procedure

As mentioned before, the Swap Procedure can be seen as a combination of dropping and adding concentrators. At first two concentrators that are to be swapped are chosen, one from active concentrators (those currently in the configuration) and one from nonactive concentrators (those not currently in the configuration). Then the chosen active concentrator is dropped from the configuration according to the Drop Procedure. The next step is adding the chosen nonactive concentrator to the configuration according to the Add Procedure. Because two methods for dropping and two methods for adding concentrators are investigated, four combinations for swapping are possible. Those are:

1. *RC-RC*: Drop randomly and add randomly.
2. *LS-RC*: Drop the least significant and add randomly.
3. *RC-COM*: Drop randomly and add the closest to the center of mass.
4. *LS-COM*: Drop the least significant and add the closest to the center of mass.

5.1.2.3 Selection of a Candidate Configuration

Having described the three procedures for generating neighbour solutions, we then move to the next step which is to decide which one of them is to be used. Before deciding which procedure is to be used, one has to decide whether the number of concentrators in the configuration will be kept constant or not. Moreover, one has to choose beforehand what methods are to be used in choosing concentrators to be added to (*RC* or *LS*) or dropped from (*RC* or *COM*) the configuration.

Keeping the number of concentrators constant can reduce the running-time of the program, but on the other hand deciding how many concentrators to be placed is a difficult problem. However, the number of concentrators given by the ADD Algorithm in the initial solution may be used as a starting point. Thus, one can select some numbers close this, and choose the one that produces the lowest configuration cost. If one decides to do this, the Swap Procedure is the only one that will be applied to get the candidate configuration. Moreover, if the given number of concentrators is less than the one given by the initial solution, the program will drop the least significant concentrators at the starting point. On the other hand if it is more that the number given by the initial solution, the program will add concentrators randomly.

If one decides to let the computer find the "best" number of concentrators, the three procedures (Add, Drop, and Swap) are applied. It will produce three different candidates with different number of concentrators. The one with the best configuration cost is chosen as the candidate configuration for further consideration in the program.

5.1.2.4 An Improvement to the Algorithms

The methods developed in this study for reassigning terminals after a concentrator is dropped from or added to the configuration will work well if the capacity of every concentrator is very large. In this case, each concentrator will rarely reject any terminal to be connected to it. Therefore, each terminal can be assigned to its best concentrator most of the time. When the capacity constraints are tight, these methods may fail to assign some terminals to their best concentrators. The tighter are the capacity constraints, the worse the reassigning methods may perform.

Therefore, in the cases where the capacity constraints are tight enough, a modification to these reassigning methods is needed. One possible modification is by applying the terminal assignment procedure to the candidate configuration. Thus, every time there is a change in the configuration, the terminal assignment procedure should be applied to get a better assignment before going to the next step in the simulated annealing procedure. However there is a tradeoff here.

Applying the terminal assignment procedure after every dropping, adding or swapping concentrators will increase the program's running-time significantly, due to the fact that the terminal assignment itself is another simulated annealing procedure. To deal with this problem, the terminal assignment procedure can be applied only after a certain number of iterations in the simulated annealing procedure. For example, in an implementation, the terminal assignment procedure is applied only if the control parameter (temperature) changes. Moreover, in the cases where the program's running-time is very crucial, it can be applied only at the end of the program. This is to make sure that we have the 'best' terminal

assignments for the chosen concentrators. These modifications reduce the running-time, but at the same time the quality of the solution may decrease.

5.2 Choosing an Annealing Schedule

As previously mentioned that there are four parameters that must be specified to make up an annealing schedule. Those are the initial temperature, the number of iterations at each temperature, the rule for decreasing the temperature and the stopping criterion. This section discusses how these parameters are obtained in this study.

5.2.1 Initial Temperature

The initial temperature is chosen so that during the iterations in this temperature, the probability for accepting a worse configuration is approximately equal to a constant called P_w. In this study, P_w is computed according to equation (4.3). To calculate the average increase in cost, the algorithms developed in this study generate a number of random transitions (neighbouring configurations) at the beginning, and they record the $\Delta Costs$ of all cost increasing transitions[4] and finally compute the average of them.

Many authors argue that the value of P_w has to be high enough. Kirkpatrick (1984), for example, suggests to use 0.80. However, the results from preliminary testing show that for the algorithms developed in this study, setting P_w to 0.80 yielded only in increasing of the programs' running-times without improving the

[4] The transitions that produce $\Delta Costs$ greater than zero

quality of solutions. Therefore, P_w for all algorithms developed in this study is set to 0.10.

5.2.2 Number of Iterations at Each Temperature

The number of iterations at each temperature k, called L_k, is not fixed but depends on a sufficient number of cost increasing transitions accepted, say, L_{min}, subject to a constant upperbound, say, L_{max}. A constant upper bound is needed because as the temperature approaches zero, the probability for accepting a worse configuration will approach zero too. Therefore, L_k will approach infinity.

Intuitively, the value of L_{min} should be much smaller than that of L_{max}. Therefore, in the algorithms for solving the terminal assignment problem L_{min} is set as a small portion of L_{max}., i.e., 2% for *BRS* and 1% for *CRS*. A different strategy is used by the algorithms for solving the concentrator location problems. In this case, the value of L_{min} is computed as a portion, which is 5% of the number of the cost increasing transitions generated when the initial temperature is being computed.

L_{max} for all algorithms is computed as a multiplication of the number of the dominant factor in generating a neighbour. It can be written as follows:

$$L_{max} = q * \eta \tag{5.7}$$

where q is a multiplier and η is the dominant factor (η= the number of terminals for the terminals assignment problem and η=the number of potential sites of concentrators for the concentrator location problem).

In the terminal assignment problem, since an attempt is always made to choose two terminals in order to generate a neighboring configuration, the number of terminals becomes the dominant factor. It is different with the concentrator

location problem where two concentrators are always chosen to generate a neighbouring configuration. Therefore, in this case, the number of concentrator becomes the dominant factor.

The value of the multiplier q in equation (5.7) will determine the quality of solutions. It is clear that the larger it is, the higher the probability for getting a better solution. However, at the same time the program's running-time may increase significantly. Therefore, choosing the right value of q is a difficult task. In this study, this value is determined through experimentation. The details of this are described in Chapter 7.

5.2.3 Temperature Decrement and the stopping Criterion

The temperature is decreased according to Equation (3.2), which is $temp_{k-1} = \alpha * (temp_k)$. For the terminals assignment problem, α is set 0.992 for the *BRS* and 0.90 for the CRS method. For the concentrator location problem, α is set to 0.85 for all methods. Finally, an algorithm stops if after t consecutive temperature steps the configuration cost does not change. For the terminal assignment problem, t is set to 3, whereas for the concentrator location problem it is set to 4. All values of α and t are obtained from preliminary experimentation.

5.3 Summary

This chapter has described how the algorithms were developed in this study to solve the terminal assignment and concentrator location problems. Three main topics were addressed, which are (1) how an initial configuration is computed, (2) how a

neighbouring configuration is generated, and (3) how an annealing schedule is chosen. The next chapter describes how these algorithms are implemented with an emphasis on the data structures and the program structures.

CHAPTER 6

Implementation of the Algorithms

The algorithms to solve the concentrator location and the terminal assignment problems that were explained in the previous chapter, are programmed in DELPHI 1.0[1], which is a PASCAL-based compiler. These programs run under the MS-Windows 3.1[2] environment.

Our programs feature several user-friendly windows to interact with users. Through these windows, users can enter the information needed to run these programs, such as the data file name, the simulated annealing parameters and the method for generating neighbours. Figure 6.1, for example, shows the opening window of the terminal assignment program, which is identical to that of the concentrator location program.

This chapter gives the implementation details of these programs with an emphasis on the data structures and the program structures. It starts with the terminal assignment program followed by the concentrator location program.

---

[1]DELPHI is a trademark of Borland International. Inc.
[2]MS and Windows are trademarks of Microsoft Corporation.

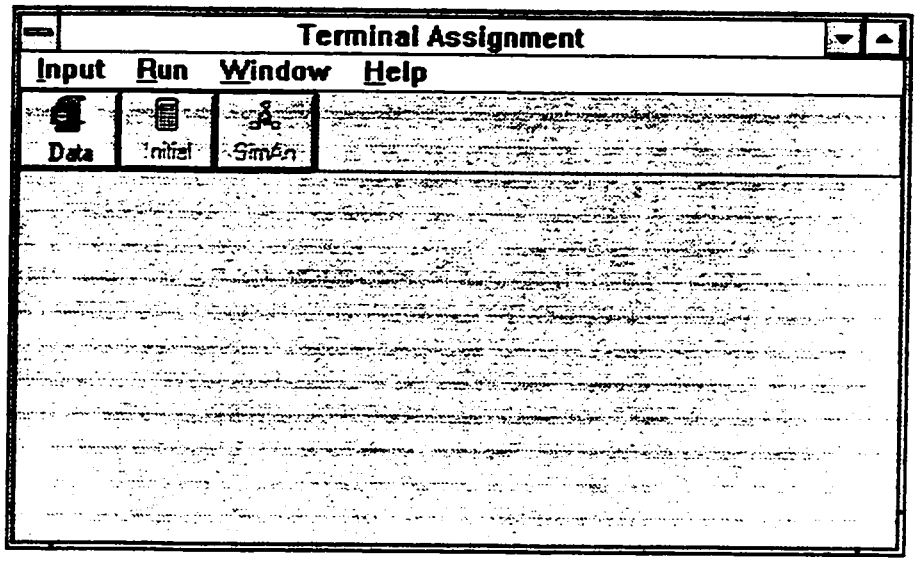


Figure 6.1. The opening window of the terminal assignment program.

6.1 Terminal Assignment

6.1.1 Input and Output

The input data for this terminal assignment program is either a connection cost matrix or coordinates of the terminals and the concentrators. In the latter case, the program assumes that the connection costs are given by the distances. Therefore, it will calculate the distances between all pairs of terminals and concentrators and keep them in a cost matrix. The input data's dialog window for this terminal assignment program is shown in Figure 6.2.

If the number of terminals is n and the number of concentrators is m, an *$(n \times m)$ matrix* is needed to represent the cost matrix. Using a regular matrix in PASCAL under the MS-Windows 3.1 environment has the effect of limiting the matrix size to 64 KBytes. Actually, this limitation originates from the memory

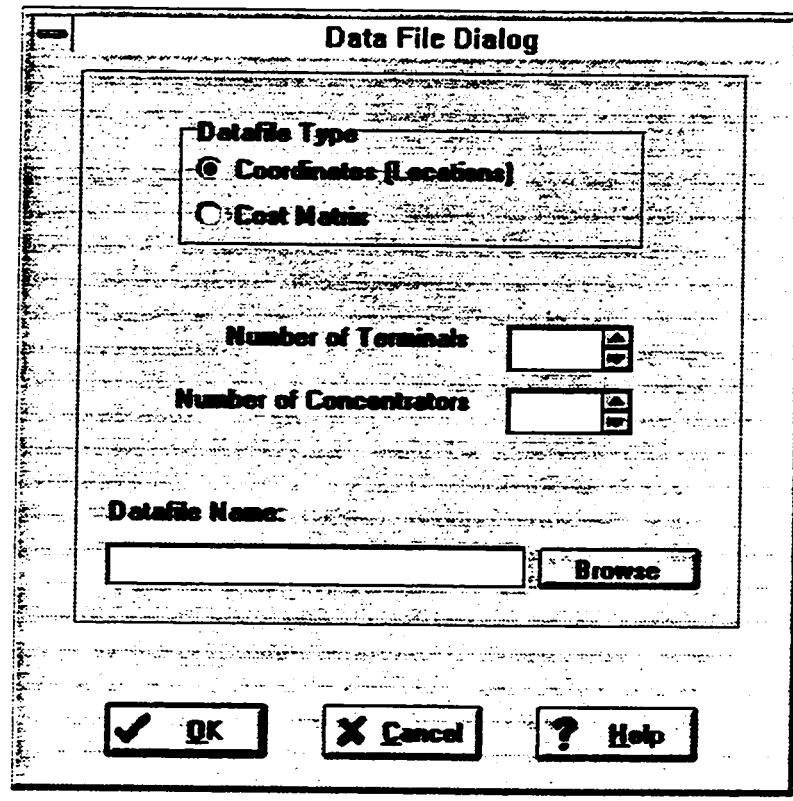


Figure 6.2. Input data dialog window of the terminal assignment program

management of MS-DOS[3] which is unfortunately kept by MS-Windows 3.1. In order to overcome this 64K limitation, all of the matrices used in this program are modified by using pointers. The following is an example of how this can be done :

Regular matrix in Pascal :

CostMatrix = Array [1..n,1..m] of LongInt;

Modified matrix :

MIntegerRowType = array[1..m] of LongInt;

CostMatrix = array[1..n] of ^MIntegerRowType;

[3] MS-DOS is a trademark of Microsoft Corporation.

Accessing data in both matrices is very similar. If in a regular matrix a cell can be accessed by *CostMatrix[row,column]*, in a modified matrix it can be accessed by *CostMatrix[row]^[column]*.

In case the input data are the coordinates of terminals and concentrators, another data structure is needed to store them before being transformed to a cost matrix. If a coordinate is represented by *point(x,y)*, it can be translated to the PASCAL code as:

```
point = record
    x       : longint;
    y       : longint;
end;
```

All coordinates of terminals and concentrators are recorded in an array as follows:

Coordinates = Array[1..n+m] of point;

Moreover, the capacity of each concentrator is stored in a *[1 × m]* array of integer, and the weight of each terminal is stored in a *[1 × n]* array of integer.

The result of this terminal assignment is stored in a $(n \times m)$ Boolean matrix. If the value of cell *(i,j)* of this matrix is *true* that means terminal i is connected to concentrator j. The actual implementation of this matrix uses the same structure as that of the cost matrix, except that the data type is Boolean. The outputs are presented both in text and graphics. An example of a graphical output of the terminal assignment program is given in Figure 6.3.

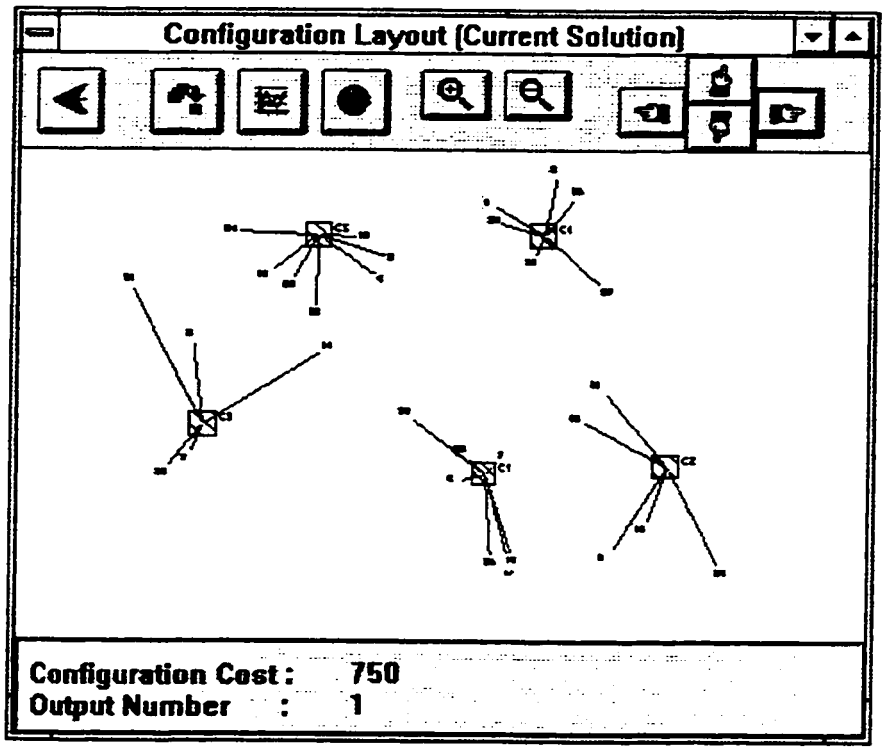


Figure 6.3. An example of a graphical output of the terminal assignment program

6.1.2 Initial Solution

To implement the modified greedy algorithm[4] in computing an initial solution as explained in the previous chapter, two main data structures are used, namely, a sorted linked-list to maintain the order of available concentrators for each terminal, and a *(1 × m)* array to keep track of the remaining capacity of each concentrator.

The tradeoff values[5] are calculated based on the first and second best available concentrators for each terminal. Therefore, in practice, it is better to create a sorted

[4] See section 3.1.2 for a description of this algorithm
[5] See equation (3.1) for the formula

list of concentrators that shows the order of the best available concentrators for each terminal. This data structure is illustrated in Figure 6.4.

The concentrators in each T_i list are sorted according to their connection cost with terminal i in increasing order. The concentrator with the smallest connection cost with terminal i is C_{i1} and the second smallest is C_{i2}. In general, if the

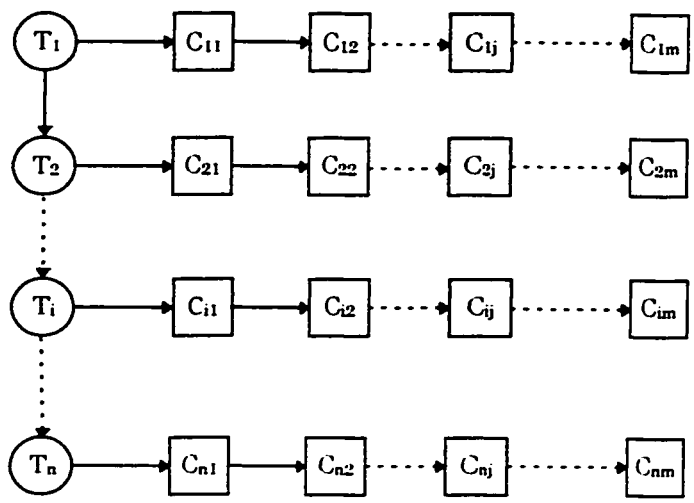


Figure 6.4. The data structure used in calculating the initial solution for the terminal assignment program

connection cost of terminal i and concentrator j is denoted by $Cost(T_i,C_{ij})$, the following expression is true for every T_i:

$$Cost(T_i,C_{i1}) \leq Cost(T_i,C_{i2}) \leq \ldots\ldots \leq Cost(T_i,C_{ij}) \leq \ldots\ldots \leq Cost(T_i,C_{im})$$

Therefore, the tradeoff value of each terminal i can be computed by taking the first two available concentrators in each T_i list starting at C_{i1} and use them as the

input for equation (3.1). Because the capacity constraints are not taken into consideration when forming the list, there may be some concentrators in a T_i list that cannot accommodate terminal i, due to its weight. In other words, the first two elements of the list T_i are not always the two best available concentrators. To speed up the search for available concentrators, whenever the procedure finds a concentrator in a T_i list that cannot accommodate terminal i, it removes that concentrator immediately from the T_i list, so that the list becomes shorter for the next search.

If a T_i list only has one concentrator left that can accommodate terminal i, the procedure simply sets its tradeoff value to minus infinity. Moreover, if a T_i list does not have any available concentrators left, the procedure will stop and report a failure. If no failure occurs, the terminal i that has the minimum tradeoff value will be assigned to its best available concentrator. It implies that a terminal with only one feasible concentrator left has the priority to be assigned first, because its tradeoff value is $-\infty$.[6] This prevents the algorithm to easily fail finding a feasible solution.

After terminal i is assigned to concentrator C_{ij}, the T_i list is removed from the overall list. Therefore, as the algorithm proceeds the size of the linked-list will keep going down and at one point the linked-list will become empty, which indicates that all terminals have already been assigned or otherwise a failure is reported. Then the procedure will stop.

[6] In the application. this value is set to the largest minus integer (-MaxLongInt=-2147483648).

6.1.3 Neighbour Generation

As explained in Chapter 5, there are two methods used in generating neighbours, which are *CRS (Completely Random Selection)* and *BRS (Best Random Selection)*. Figure 6.5 shows the dialog window for selecting the method to be used in generating a neighbouring solution.[7] Regardless of the chosen method, the first step

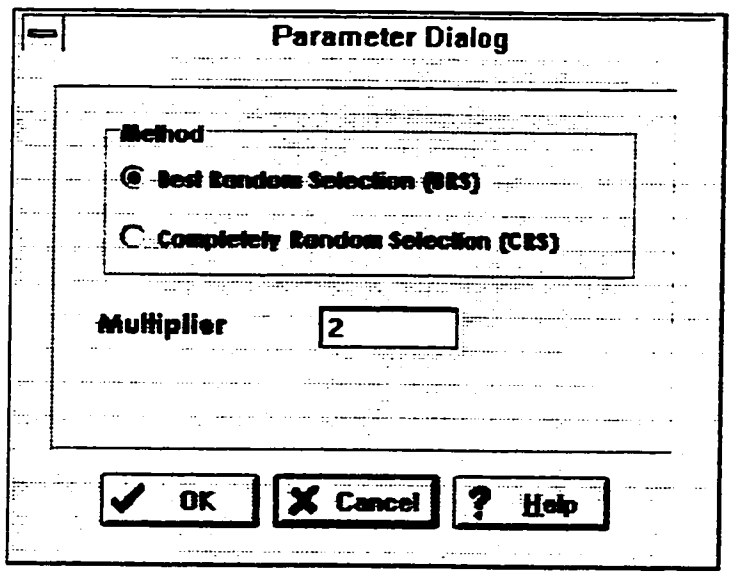


Figure 6.5. Method selection window of the terminal assignment program

in the neighbour generation process is to choose a pair of concentrators. Since the number of concentrators is given, no special data structure is needed. The selection of concentrators can be done simply by generating two different random numbers that are ceiled by the number of concentrators in the configuration. The values of random number generated represent the concentrators to be chosen.

[7] The value of the multiplier has to be defined in this window too.

The next step is to choose what action is to be done toward the chosen concentrators. As described in the previous section, the action can be either a move of a terminal from the first concentrator to the second one or a swap between two terminals, one from each chosen concentrator. Regardless of the chosen action, one has to make sure that it is possible to be accomplished. It means that inequality (5.2) has to be satisfied. To implement this, three data structures are involved, Those are two *[1×m]* arrays of integer, one for storing the capacity of each concentrator and the other one for storing the remaining capacity in each concentrator, and one *[1×n]* array of integer for storing the weight of each terminal.

6.1.4 Program Structure

Figure 6.6 shows the main program for solving the terminal assignment problem written in pseudo-PASCAL. This program is an instance of the simulated annealing algorithm (Figure 4.2). The final result of this program is stored in a Boolean matrix called *BestConfig*. The program starts with an initialization process of some parameters and configurations as listed from line 3 to 11. The initial configuration is obtained by using the modified greedy algorithm (line 3).

Due to the nature of simulated annealing that it accepts a worse configuration with a certain probability, the current configuration is not always the best configuration.[8] Therefore, two different configurations are needed to store each of them. Those are *CurrentConfig*, which is used to store the current configuration and *BestConfig*, which is used to store the best configuration found so far. At the

[8] See Chapter 4

beginning, both *CurrentConfig* and *BestConfig* are set equal to the initial configuration (lines 4 & 5).

```
1            Procedure TeminalAssignment (BestConfig);
2            begin
3               ModifiedGreedyAlgorithm (InitConfig);
4               SetEqual (CurrentConfig.InitConfig);
5               SetEqual (BestConfig.CurrentConfig);
6               Initialize(LMax.LMin.MaxCostUnchange.InitTemp.alpha);
7               NewCost := ConfigurationCost(CurrentConfig);
8               BestCost := NewCost;
9               CurrentCost := NewCost;
10              temp := InitTemp;
11              CostUnchange := 0;
12              repeat
13                 L:=0;
14                 UpCostChange := 0;
15                 repeat {inner loop starts}
16                    Inc(L);
17                    SelectTwoConcentrators (conc1.conc2);
18                    case TerminalSelectMethod of
19                       1  :  RandomSelection (conc1.conc2.term1.term2.savings);
20                       2  :  BestSelection (conc1.conc2.term1.term2.savings);
21                    end;
22                    if savings > 0 then
23                    begin
24                       UpdateConfiguration (CurrentConfig.conc1.conc2.term1.term2);
25                       NewCost := ConfigurationCost(CurrentConfig);
26                       if NewCost < BestCost then
27                       begin
28                          SetEqual (BestConfig.CurrentConfig);
29                          BestCost := NewCost;
30                       end;
31                    end
32                    else
33                    if exp(savings/temp) > random(0.1) then
34                    begin
35                       UpdateConfiguration (CurrentConfig.conc1.conc2. term1.term2);
36                       NewCost := ConfigurationCost(CurrentConfig);
37                       Inc(UpCostChange);
38                    end;
39                 until (UpCostChange=LMin) or (L=LMax); {inner loop ends}
40                 if CurrentCost = NewCost then
41                    Inc(CostUnchange)
42                    else
43                    begin
44                       CostUnchange := 0;
45                       CurrentCost := NewCost;
46                    end;
47                 temp := temp*α;
48              until CostUnchange = MaxCostUnchange;
49           end;
```

Figure 6.6. Pseudo-PASCAL of the main program for solving the terminal assignment problem

In line 6, some parameters are initialized. Those are *LMax*, *LMin*, *MaxCostUnchange*, *InitTemp* and *alpha*. *LMax* and *LMin* will determine when the iterations at a certain temperature stop as shown in line 39. *LMax* is the upperbound of the number of iterations whereas *LMin* is the maximum number of cost increasing transitions that can be accepted at a certain temperature. *MaxCostUnchange* denotes the maximum number of consecutive temperature steps allowed for the configuration cost to be unchanged. This value determines when the program will stop as shown at line 48. *InitTemp* is the initial temperature and *alpha* is the decreasing factor for the temperature.[9]

Three variables representing configuration costs are used, namely, *NewCost*, *CurrentCost* and *BestCost*. *NewCost* is used to store the cost of the newly accepted configuration at the current temperature. *CurrentCost* is used to store the cost of the configuration accepted in the previous temperature. *BestCost* is used to store the cost of the best configuration found so far. Each time the temperature changes, the value of *CurrentCost* is compared to that of *NewCost* (line 40). If their values are equal, it means that the configuration cost does not change. If the configuration cost does not change for a consecutive number of temperature steps, i.e., *MaxCostUnchange*, the program stops (lines 40-48).

After initializing several configurations and parameters, the program continues with the neighbour generation process. Inside the inner loop which starts at line 15, the main processes of the neighbour generation are accomplished. First two concentrators are chosen randomly (line 17), then the method of how to generate a neighbour is chosen; either randomly (line 19) which represents the *CRS* method,

[9] See section 5.2 for the detail description of how to determine these parameters.

or based on the best savings that can be obtained (line 20) which represents the *BRS* method. After a neighbouring configuration is chosen, the obtainable savings are evaluated. If there are savings (line 22), the chosen neighbouring configuration is set to be the *CurrentConfig* (line 24), and then if the cost of *CurrentConfig* becomes better than that of the *BestConfig*, the *CurrentConfig* is set to be the *BestConfig* (lines 26-30). If there are no obtainable savings, the neighbouring configuration is accepted with a certain probability as the *CurrentConfig* (line 33-38).

6.2 Concentrator Location

6.2.1 Input and Output

The input data of the concentrator location program is similar to that of the terminal assignment program. It can be a cost (distance) matrix or coordinates of the terminals and potential concentrators. However, in the concentrator location problem, the center, called C_0 must be considered as a potential concentrator. Thus, the number of potential concentrators becomes $m+1$ and consequently the size of all data structures that represent the potential concentrators also becomes $m+1$. For example, the size of the cost matrix becomes $(n \times (m+1))$, where n is the number of terminals. It should be noted that it is possible to include the center in the terminal assignment problem by treating it as a regular concentrator.

As that in the terminal assignment, the concentrator location program also needs to know the capacity of each concentrator, which is stored in a $(1 \times (m+1))$ array, and the weight of each terminal, which is stored in a $(1 \times n)$ array. Moreover, this program requires the cost for locating each concentrator (d_j), which is stored in

(1 × (m+1)) array. If the input data are the coordinates of the terminals and the potential concentrators, the unit costs of both high and low capacity lines are also needed. Figure 6.7 shows the input data's dialog window of the concentrator location program.

Since it is likely that not all of the potential concentrators will be in the final configuration, there should be a way to distinguish which concentrators are in use (*active)* and which of them are not (*nonactive*). Therefore, in this program there are two single arrays that record this information, one for the active concentrators and

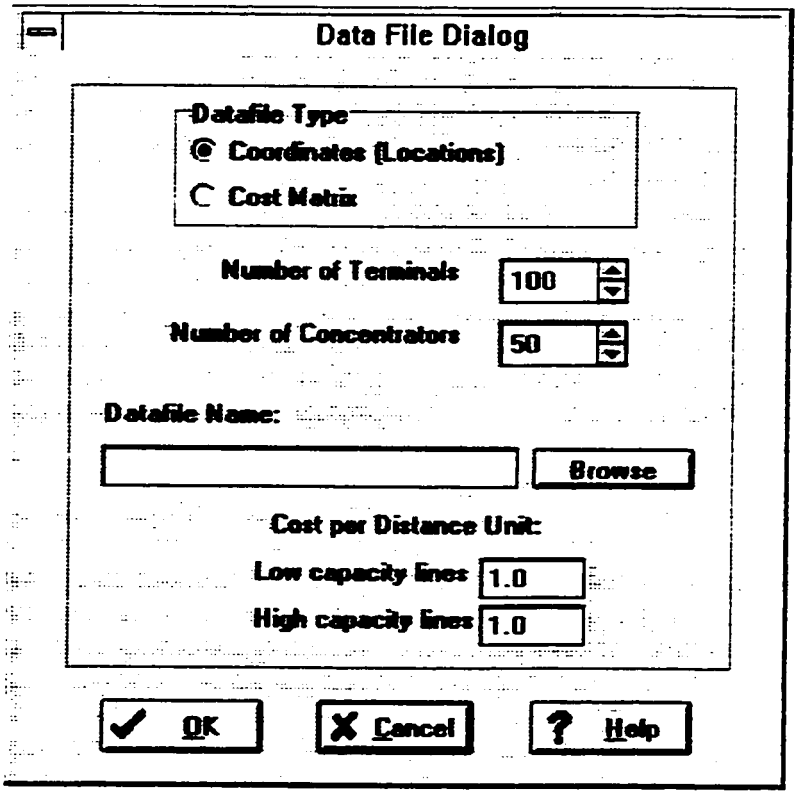


Figure 6.7. Input data's dialog window of the concentrator location program

the other one for the nonactive concentrators. Two arrays are used, instead of one, in order to make the swapping process workable, which will be explained later in this chapter.

The configurations or the solutions are stored in the *(n × (m+1))* Boolean matrices, where *true* means that the connection is established and *false* means that it is not. As those in the terminal assignment program, the outputs of this concentrator location program are presented both in text and graphics. Figure 6.8 shows an example of a graphical output of this program.

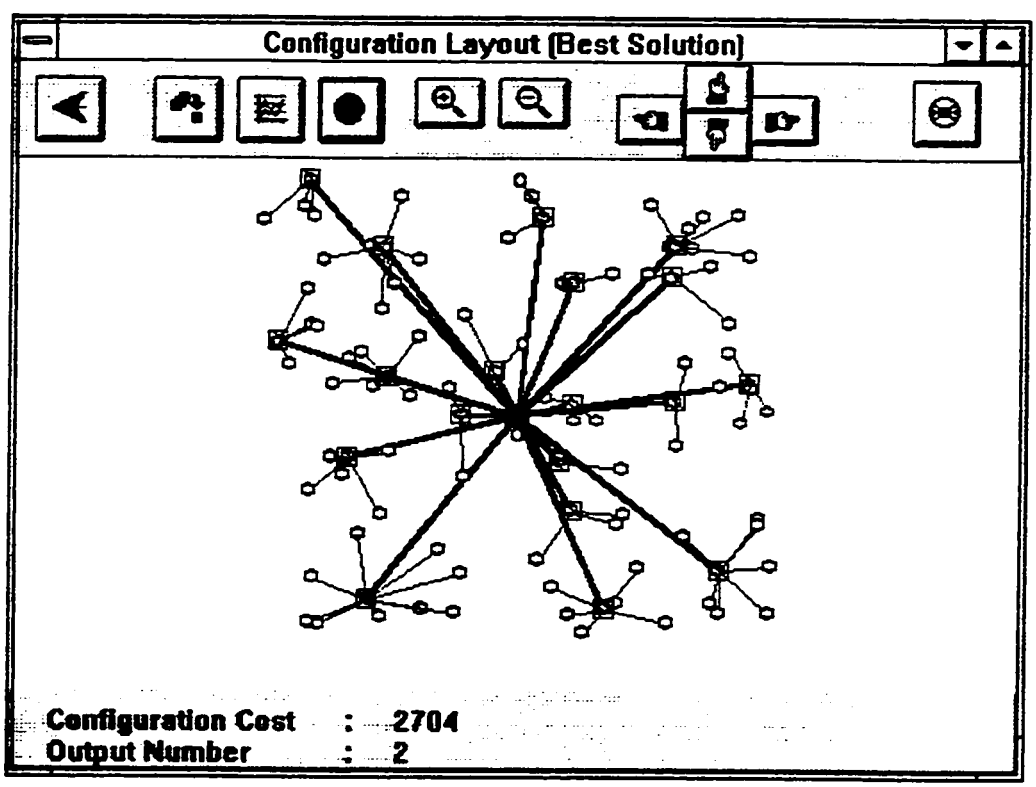


Figure 6.8. An example of a graphical output of the concentrator location program

6.2.2 Initial Solution

In implementing the Add algorithm a sorted linked list, as shown in Figure 6.9, is used. For each potential concentrator j, a sorted list of terminals is constructed. The terminals in each C_j list are sorted in decreasing order based on the amount of money that can be saved if they are moved from the concentrator to which they are currently associated to concentrator j. Thus, $T1_j$ is the best candidate terminal to be moved to concentrator j, and Ti_j is the i^{th} best candidate terminal to be moved to concentrator j. The saving that can be obtained if terminal i is moved to

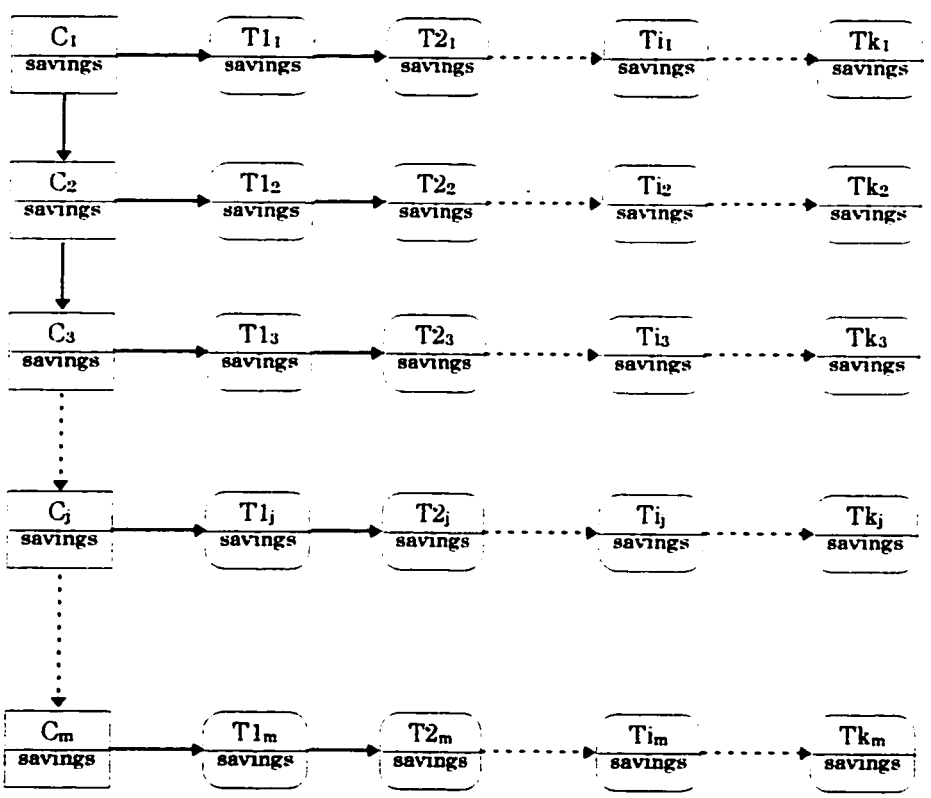


Figure 6.9. Data structure used in calculating the initial solution for the concentrator location program

concentrator j is denoted as *savings* which is written underneath Ti_j. From now on, it will be referred to as $Ti_j.savings$. Only the terminals for which money can be saved are included in the list. Hence, for C_1 there are k_1 terminals that can save money if they are moved to it, and for C_j there are k_j terminals.

The total savings that can be made by each concentrator j are calculated by adding up the $Ti_j.savings$ in that C_j list subject to the capacity of concentrator j. Based on the savings they can make, the C_j lists are sorted in decreasing order, so that

$$C_1.savings \geq C_2.savings \geq \ldots\ldots \geq C_j.savings \geq \ldots \geq C_m.savings$$

where $C_j.savings$ denotes the obtainable savings if concentrator j is added to the configuration. Therefore, the concentrator that can save the most money will always be on top of the list.

By using this data structure the updating time can be reduced. If we take a close look at the Add algorithm, it is obvious that as it proceeds, the terminals only move to the concentrators that are closer to them. It implies that $Ti_j.savings$ never increases. As a result, the values of $C_j.savings$ will never increase either.

For example, let us say that C_1 is the concentrator that is on top of the list and T_1 and T_2 are the terminals with which money can be saved if they are moved to C_1. After adding C_1 to the configuration, the C_1 list should be deleted from the overall list, and the remaining list should be updated. To update the remaining list, first we have to find out which concentrator lists contain T_1 and T_2 and then recalculate the new savings that can be made if these terminals are moved from their new home, C_1, to the concentrator being evaluated. Having moved T_1 and T_2 to the closer concentrator, C_1, the savings that we can obtain by moving them again to other

concentrators should decrease. In some cases, there may be no savings anymore. If no savings can be made, these terminals are deleted from the list, otherwise they should be relocated to keep the list sorted. Having deleted or relocated these terminals, the new total savings of the concentrator being evaluated ($C_j.savings$) should be recalculated. These new savings will not be more than the savings that it could make previously. Therefore, we do not need to compare this concentrator with all others, but only with those that saved less previously. Thus, by keeping the list sorted, the time needed to update it can be reduced, because only a portion of the linked-list needs to be reevaluated.

The result of this Add Algorithm is stored in a $(1 \times n)$ array of integer. The indices represent the terminal numbers, and the contents of the array are the concentrator numbers in which the terminals are assigned to. However, this data structure is not suitable to be used in the simulated annealing algorithm. Therefore, this result is transformed to a $(n \times (m+1))$ Boolean matrix as was used in the terminal assignment program.

6.2.3 Neighbour Generation

As explained in Chapter 5, there are three different procedures in generating a neighbouring solution. i.e., add, drop, and swap procedures. Add procedure itself can be done in two ways, namely, *RC* (randomly chosen) and *LS* (the least significant). Drop procedure can also be done in two ways, namely, *RC* (randomly chosen) and *COM* (the closest to the center of mass). Because a swap can be done by applying drop and add procedures sequentially, it does not have its own method.

The dialog window for choosing the procedure (method) to be used in generating neighbouring solutions is shown in Figure 6.10.

In order to choose a concentrator randomly (*RC* method) two lists of concentrators are used, one for the *active* (in use) concentrators and one for the *nonactive* (not in use) concentrators. These lists are implemented as *(1 × (m+1))* arrays. The first list, namely, the *active* list, is used in the process of choosing a concentrator to be dropped from the configuration (drop procedure), whereas the second one, namely, the *nonactive* list, is used in the process of choosing a concentrator to be added to the configuration (add procedure). Having the *active* list allows the calculation of the configuration cost becomes faster since the program does not need to check every potential concentrator but can go directly to the concentrators pointed to by the *active* list.

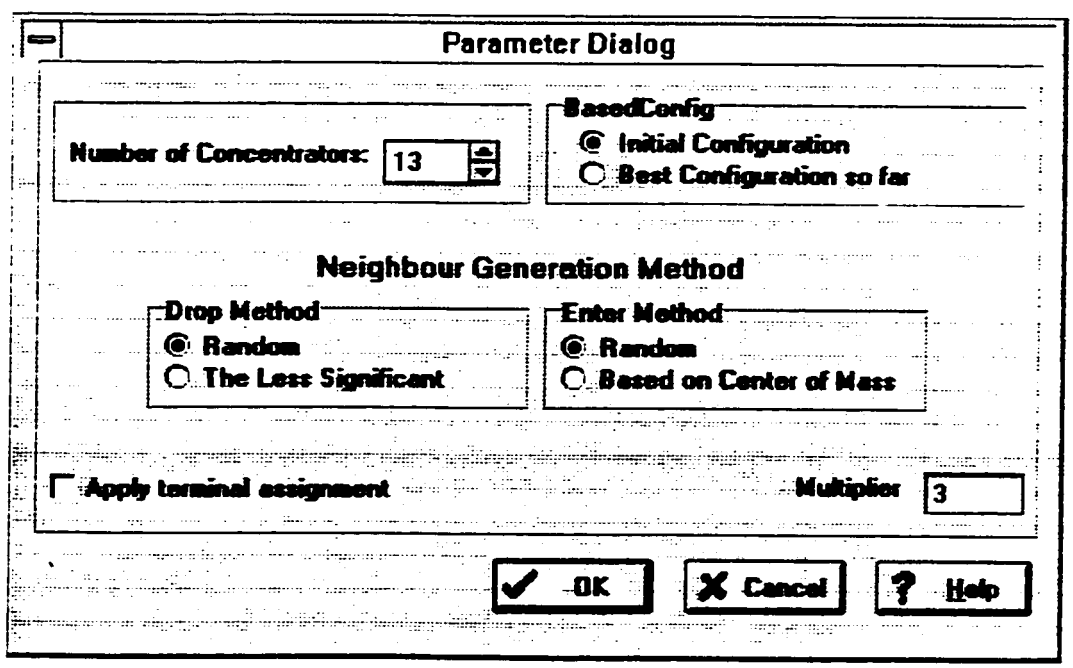


Figure 6.10. Method selection window for the concentrator location problem

To accelerate the process of reassigning a terminal to another concentrator, a sorted list of concentrators, called *lookup*, is constructed for each terminal. The best concentrator for each terminal is located on top of every list. Thus, in order to reassign a terminal, one just needs to go to that terminal's *lookup* list to find its new best available concentrator.

However, one has to make sure that the chosen new home is being used in the current configuration. One way to accomplish this is by looking at the concentrators present in the *active* list. However, it will take time if we have to search the entire *active* list every time a terminal needs to be reassigned. Therefore, a new data structure, named *inuse* list is used. This is a Boolean array of concentrators. If a concentrator is in use, its value is set to be true; otherwise, its value is set to be false.

6.2.4 Program Structure

Figure 6.11 shows the main structure of the concentrator location program, which is very similar to that of the terminal assignment program. It starts by calculating the initial configuration (*InitConfig*). Then, all other configurations, parameters and lists are initialized.

Besides the initial configuration, there are two other configurations used in this program which are the current configuration (*CurrentConfig*) and the best configuration (*BestConfig*). These two configurations are used in the same way as those in the terminal assignment program. The parameters that are initialized at line 6 are also function similarly with those of the terminal assignment program.

```
1   Procedure ConcentratorLocation(BestConfig):
2   begin
3       AddAlgoritm(InitConfig);
4       SetEqual(CurrentConfig.InitConfig):
5       SetEqual(BestConfig.InitConfig):
6       Initialize(LMax,LMin,MaxCostUnchange,InitTemp,alpha):
7       Initialize(ActiveList,NonActiveList,CapUsedList,InUsedList):
8       Generate(LookUpList);
9       NewCost := ConfigCost(CurrentConfig):
10      CurrentCost := NewCost;
11      BestCost := NewCost;
12      temp := InitTemp;
13      repeat
14          L := 0;
15          UpCostChange := 0;
16          repeat
17              Inc(L);
18              Case Method of
19                  1: ChooseRandomly(): {a concentrator to be dropped}
20                     ChooseRandomly(): {a concentrator to be added}
21                  2: ChooseLeastSignificant(): {a concentrator to be dropped}
22                     ChooseRandomly(): {a concentrator to be added}
23                  3  ChooseRandomly(): {a concentrator to be dropped}
24                     CenterOfMass(): {a concentrator to be added}
25                  4  ChooseLeastSignificant ():{a concentrator to be dropped}
26                     CenterOfMass(): {a concentrator to be added}
27              end: {End of Case}
28              GenerateNeighbour():
29              TerminalAssignment():
30              CalculateSavings():
31              if savings > 0 then
32              begin
33                  UpdateConfig(CurrentConfig):
34                  NewCost := ConfigCost(CurrentConfig):
35                  if NewCost <  BestConfig then
36                  begin
37                      BestCost :=NewCost:
38                      SetEqual(BestConfig.CurrentConfig):
39                  end;
40              end
41              else
42              if exp(savings/temp) > random(0.1) then
43              begin
44                  UpdateConfig(CurrentConfig):
45                  NewCost := ConfigCost(CurrentConfig):
46                  Inc(UpCostChange):
47              end;
48          until (UpCostChange = Lmin) or (L = Lmax):
49          if CurrentCost = NewCost then
50              Inc(CostUnchange)
51              else
52              begin
53                  CostUnchange := 0;
54                  CurrentCost := NewCost:
55              end;
56      temp := temp*α
57  until CostUnchange = MaxCostUnchange:
58  end; {End of Concentrator Location}
```

Figure 6.11. Pseudo-PASCAL of the main program for solving the concentrator location problem

After completing the initialization process, the program continues with the neighbour generation process. The four methods for choosing concentrators to be added and dropped are listed from line 19 to line 26. The method to be used has to be chosen beforehand by users. Depending on what method is chosen, this program will run one out of these four methods at a time, and then a neighbouring solution is generated (line 28). The quality of a neighbouring solution can be improved by applying a better terminal assignment, so that at line 29 the terminal assignment procedure is called.

After a neighbouring solution is generated, the savings that can be made by this new configuration are calculated (line 30). If there are savings, the current configuration is updated by accepting the new configuration as the current configuration. Moreover, if the cost of this newly accepted configuration is less than the cost of the best configuration found so far, then it is set to be the best configuration (line 38). If there are no savings, the new configuration can be accepted as the current configuration with a certain probability (lines 42 - 47).

It should be noted that in this pseudo code, the terminal assignment procedure is called within the inner loop. This is desirable, because each time a concentrator is added, dropped or swapped with another one, all terminals have to be reassigned, since the assignments are specific to a set of concentrators. However, it will take a long time to run the whole program, because the terminal assignment program itself is another simulated annealing application. Therefore, in this research some modifications of this program are tried. First, the terminal assignment procedure is moved outside of the inner loop, i.e., after line 48. However, if the running-time is still extremely large, the terminal assignment procedure can be called only twice:

once at the beginning and once the end of the program. Moreover, for the uncapacitated cases where the terminal assignment procedure is not needed, it is removed completely from the program.[10]

Similarly to the terminal assignment program, this program stops when the cost of the current configuration does not change for a consecutive number of temperature steps (line 57). The counter for this is shown at line 50.

6.3 Summary

This chapter described the implementation details of the algorithms developed in this study with an emphasis on the data structures and the program structures. The chapter also showed some examples of the input/output window interface where the programs interact with users.

The next chapter presents the experiments conducted for the algorithms developed in this study. It includes the comparison between the results of the algorithms developed in this study with those of some well-known heuristic algorithms.

[10] See section 5.1.2.4

CHAPTER 7

Computational Experiments

In this chapter, some computational experiments for the algorithms developed in this study are presented. All experiments were carried out on a PC with an Intel Pentium 75 MHz processor. This chapter starts with the discussion of the how the experiments were setup and conducted for the algorithms developed for solving the terminal assignment problems and will move to the experiments for the algorithms developed for solving the concentrator location problems.

Recall that n is the number of terminals, m is the number of potential concentrators, k_j is the capacity of concentrator j and w_i is the weight of terminal i. Throughout this chapter, the size of a network will be referred in terms of (n,m).

7.1 Terminal Assignment

The main purpose of this experiment is to compare the results obtained from the algorithms developed in this study to those given by the modified greedy algorithm. Five different sizes of networks were considered in this experiment. In terms of (n,m), those were (100,20), (200,40), (300,60), (400,80) and (500,100). We assumed

that the capacity of all concentrators were uniform with k_j=k=12. The weight of each terminal was randomly generated between 1 and 3.

For each problem size, five different collections of terminals and concentrators were randomly generated on a 200 × 200 unit rectangular grid. The two methods developed in this study, *CRS* and *BRS*, were applied to these data. The multiplier used for the *BRS* method was 2 and for the *CRS* method was 66.[1] These values were chosen from preliminary experiments so that the running times of both methods were not very different. The *BRS* method needs fewer iterations than does the *CRS* method, this is because it always searches for the best neighbour that can be generated from the chosen concentrators. On the other hand, the *CRS* method simply chooses a neighbour randomly. This means that it needs a large number of iterations in order to obtain a good result.

The initial solution for each method was computed using the modified greedy algorithm with the tradeoff value of 0. By using this tradeoff value, the modified greedy algorithm became similar to the original greedy algorithm. Moreover, eleven modified greedy algorithms with different tradeoff values, ranging from 0.1 to 1.0,[2] were also applied to the data. The best of them for each problem size was then compared to the results given by the simulated annealing (SA) algorithm.

Table 7.1 shows the improvements gained by the methods developed in this study over the initial solution that were computed using the greedy algorithm along with their running times. The percentage improvements gained by the *CRS* and *BRS* methods are given in the shaded columns. The running times of the greedy

[1] Recall that the multiplier will determine the upperboud of the number of iterations at each temperature.
[2] Incremented by 0.1.

algorithm are not presented in this table because they were very fast, even for the problems of size (500,100), they took only less than 3 seconds.

Table 7.1. The Improvements Gained by the *BRS* and *CRS* Methods over the Greedy Algorithm

Problem size	Rep	Greedy Algorithm	SA (BRS method)			SA (CRS Method)		
		Cost	Cost	Imp (%)	Time (sec)	Cost	Imp (%)	Time (sec)
(100,20)	1	2653	2526	4.79	19	2535	4.45	24
	2	2687	2435	9.38	25	2442	9.12	23
	3	3870	3577	7.57	19	3598	7.03	24
	4	3539	3257	7.97	19	3300	6.75	28
	5	2749	2490	9.42	23	2491	9.39	27
	Avg.			7.83	21		7.35	25
(200,40)	1	4648	4131	11.12	112	4150	10.71	116
	2	3974	3578	9.96	135	3587	9.74	128
	3	5362	4980	7.12	120	4969	7.33	115
	4	4573	4200	8.16	120	4198	8.20	114
	5	4082	3536	13.38	142	3556	12.89	116
	Avg.			9.95	126		9.77	118
(300,60)	1	6156	5394	12.38	313	5431	11.76	312
	2	4779	4457	6.74	304	4496	5.92	241
	3	4447	4129	7.15	325	4158	6.50	295
	4	6701	5853	12.66	353	5906	11.86	308
	5	4628	4303	7.02	354	4370	5.57	282
	Avg.			9.19	330		8.33	288
(400,80)	1	7268	6425	11.60	623	6486	10.76	658
	2	5968	5314	10.96	678	5431	9.00	624
	3	5095	4814	5.52	727	4894	3.96	590
	4	7348	6313	14.09	777	6359	13.46	715
	5	6074	5420	10.77	687	5386	11.33	615
	Avg.			10.59	698		9.70	640
(500,100)	1	7871	6869	12.73	1115	6998	11.09	1152
	2	6446	5975	7.31	1019	6011	6.75	1077
	3	6421	5799	9.69	1095	5853	8.85	1148
	4	7323	6605	9.80	1093	6653	9.15	1164
	5	6002	5613	6.48	1115	5671	5.51	968
	Avg.			9.20	1087		8.27	1102

For all the problem sizes considered in this study, the *BRS* method performed slightly better than did the *CRS* one, as shown from the average improvement for each problem size, where the *BRS* method always gave a better result. If we take the overall average, regardless of the problem size, 9.35% (standard deviation = 2.58%) of improvement over the greedy algorithm's result was given by the *BRS* method and only 8.65% (standard deviation = 2.65%) improvement was given by the *CRS* method. The largest improvement gained in this study was also given by the *BRS* method, which was 10.59% (for the problems of size (400,80)). Therefore for further comparison, we will consider only the results from the *BRS* method.

In Table 7.2, the results obtained from the *BRS* method are compared to those obtained from the best modified greedy algorithm that were examined in this study. As mentioned above, 11 different tradeoff values for the modified greedy algorithm were applied to the data generated in this study. The best of them for each data size was presented in the table along with the tradeoff parameter (α) that was used to produce it. From this table, we can see that the *BRS* method performed better than the best modified greedy algorithm did, and this for every problem size examined in this study. The smallest improvement was for the problems of size (100,20) which was 4.02%, whereas the largest improvement obtained was for the problems of size (400,80) which was 6.99%. The overall average improvement, regardless of the problem size, was 5.51% with a standard deviation of 3.01%.

Moreover, we believe that the running times of the *BRS* and *CRS* methods in solving the data sets generated in this study are still acceptable (see Table 7.1). For

the largest data set tested, i.e. (500,100), the average running time was 1087 seconds (18.12 minutes).

Table 7.2. Comparison between the "Best" Modified Greedy Algorithm and the SA Algorithm

Problem Size	Rep	Modified Greedy Algorithm		SA (BRS Method)	
		Cost	tradeoff α	Cost	Imp (%)
(100,20)	1	2606	0.2	2526	3.07
	2	2447	0.9	2435	0.49
	3	3870	0.0	3577	7.57
	4	3532	0.2	3257	7.79
	5	2520	1.0	2490	1.19
	Avg.				4.02
(200,40)	1	4550	0.6	4131	9.21
	2	3793	0.6	3578	5.67
	3	5235	1.0	4980	4.87
	4	4328	1.0	4200	2.96
	5	3715	0.8	3536	4.82
	Avg.				5.51
(300,60)	1	5701	1.0	5394	5.39
	2	4598	0.9	4457	3.07
	3	4325	0.4	4129	4.53
	4	6370	0.8	5853	8.12
	5	4428	0.9	4303	2.82
	Avg.				4.79
(400,80)	1	7268	0.0	6425	11.60
	2	5656	0.8	5314	6.05
	3	4879	0.9	4814	1.33
	4	7135	0.4	6313	11.52
	5	5672	1.0	5420	4.44
	Avg.				6.99
(500,100)	1	7599	0.9	6869	9.61
	2	6315	0.4	5975	5.38
	3	6149	0.8	5799	5.69
	4	7113	0.7	6605	7.14
	5	5802	1.0	5613	3.26
	Avg.				6.22

7.2 Concentrator Location

Two steps of experiments were conducted in this part. First, all algorithms (methods) developed in this study were compared to each other in order to obtain the best of them. Second, the best algorithm was compared to the ADD algorithm, which was the one used to compute the initial solutions. In addition, we compared our algorithm to the best algorithm available in the program called SITATION, which is written by Daskin (1995) for solving the uncapacitated facility location problems.

The data used in this study were randomly generated on a 200 × 200 unit rectangular grid. We assumed that the concentrators to be located were all identical in terms of the capacity and the installation costs. Therefore, the cost for locating a concentrator at site j was determined only by its distance to the central computer. The capacity of each concentrator was assumed to be 12, and the weight of each terminal was randomly generated between 1 and 2. The cost of a high capacity line was assumed to be twice the cost of a low capacity line. Moreover, the first half of the terminal sites were considered as the potential sites of the concentrators to be located.

Recall that based on the procedure of dropping and adding a concentrator, four different methods are possible for generating a neighbour. Those are *RC-RC* (drop randomly and add randomly), *LS-RC* (drop the least significant and add randomly), *RC-COM* (drop randomly and add the closest to the center of mass) and *LS-COM* (drop the least significant and add the closest to the center of mass). To find out which one was the best, two different sizes of networks were considered, i.e., (100,50) and (200,100). For each of them, five different data sets were randomly

generated. Then the four methods were applied to these data. The number of concentrators was kept constant with the number obtained by the initial solution. The multiplier used was 6 and the terminal assignment procedure was not applied.

The results given by each method are presented in Table 7.3. The percentage improvement obtained by each method over the results of the ADD algorithm are presented in the shaded columns. The improvements given by *RC-RC* method surpassed those of the other methods, both for the problems of sizes (100,50) and

Table 7.3. The Improvements Gained by Different Methods of the Simulated Annealing Algorithm over the ADD algorithm

Problem size	Rep	ADD	RC-RC		LS-RC		RC-COM.		LS-COM	
		cost	cost	Imp (%)	cost	Imp (%)	cost	Imp (%)	cost	Imp (%)
(100,50)	1	3923	3834	2.27	3923	0.00	3864	1.50	3923	0.00
	2	3856	3828	0.73	3856	0.00	3794	1.61	3856	0.00
	3	3917	3776	3.60	3875	1.07	3807	2.81	3917	0.00
	4	4052	3949	2.54	4052	0.00	3998	1.33	4052	0.00
	5	3886	3807	2.03	3886	0.00	3800	2.21	3886	0.00
	Avg.			2.23		0.21		1.89		0.00
(200,100)	1	6966	6702	3.79	6915	0.73	6705	3.75	6961	0.07
	2	6529	6427	1.56	6529	0.00	6421	1.65	6529	0.00
	3	6588	6588	0.00	6588	0.00	6417	2.60	6588	0.00
	4	6757	6446	4.60	6636	1.79	6570	2.77	6725	0.47
	5	6763	6465	4.41	6622	2.08	6599	2.42	6763	0.00
	Avg.			2.87		0.62		2.64		0.11

(200,100). Only the improvements obtained by the *RC-COM* method that were close to those obtained by the *RC-RC* method. Therefore, for the additional experiments, we considered only the *RC-RC* method.

As previously mentioned, the value of the multiplier will determine the quality of the solutions.[3] The bigger it is, the higher the probability for getting a better solution. However at the same time the program's running times could increase significantly. Therefore, after choosing the best method, we then tried to find an acceptable multiplier in terms of the quality of the solution and the acceptable running-times.

Table 7.4 shows the improvements over the results of the ADD algorithm given by two different multipliers, which are 3 and 6, along with their running times. Contrarily to the previous experiments, the terminal assignment procedure was applied this time. However, in order to save time, the terminal assignment procedure was applied only twice; once at the beginning and once at the end of the program.

Table 7.4. Comparison between the Improvements Gained by Two Multipliers

Problem size	Rep	multiplier=3		multiplier=6	
		Imp. (%)	time (sec)	Imp. (%)	time (sec)
(100,50)	1	3.01	61	3.05	112
	2	1.53	59	1.53	142
	3	3.93	55	4.19	85
	4	2.02	57	3.16	102
	5	2.11	53	2.11	96
	Avg.	2.52	57.00	2.80	107.40
(200,100)	1	5.34	479	5.27	814
	2	1.86	530	2.04	843
	3	2.84	655	1.35	822
	4	5.02	471	5.18	912
	5	5.93	524	6.48	812
	Avg.	4.06	531.80	4.05	840.60

[3] See section 5.2.2.

For the problems of size (100,50) the multiplier of 6 gave a better improvement (2.80%) than that of the multiplier of 3 (2.52%). However, for the problems of size (200,100) the results were almost identical. The overall average improvements given by these two multipliers are also very close. Regardless of the problem size, the mean percentage improvement gained by the multiplier of 6 was 3.43% (standard deviation = 1.76%) whereas the gain by the multiplier of 3 was 3.29% (standard deviation = 1.66%). Moreover, in both problem sizes, the running times of the multiplier of 6 were much longer than those of the multiplier of 3. Based on these facts, the multiplier of 3 was chosen for the further experiments.

In Table 7.5, our implementation of the simulated annealing with the multiplier of 3 is compared to the ADD algorithm. In this comparison, our algorithm allowed the number of concentrators in the configuration to change at every iteration. In addition, the terminal assignment procedure was applied at the beginning and at the end of the program.

It seems that for the small size problems, the improvement was not much better. For example, for problems of size (100,50), the average improvement was only 2.83%. However, for the larger problems tested in this experiment, the simulated annealing (SA) algorithm performed better. For the problems of size (400,200), for example, the SA algorithm improved the results given by the ADD algorithm by about 5.12%. For this size of problem, it took 5116 seconds (1 hour 42 minutes) on average to get the result.

Table 7.5. Comparison between the Simulated Annealing and the ADD Algorithm

Problem size	Rep	ADD		Simulated Annealing			Imp (%)
		Cost	#Conc	Cost	#Conc	Time sec)	
(100,50)	1	3923	13	3795	13	49	3.26
	2	3856	14	3793	14	75	1.63
	3	3917	15	3770	14	61	3.75
	4	4052	15	3914	15	56	3.41
	5	3886	13	3804	13	74	2.11
	Avg.					63	2.83
(200,100)	1	6966	27	6565	27	487	5.76
	2	6529	27	6384	28	496	2.22
	3	6588	26	6354	26	441	3.55
	4	6757	27	6332	26	528	6.29
	5	6763	27	6346	26	550	6.17
	Avg.					500	4.80
(300.150)	1	9521	40	9212	38	2076	3.25
	2	10044	41	9467	40	1569	5.74
	3	9338	40	9170	40	1367	1.80
	4	9530	37	9354	37	1912	1.85
	5	9325	39	8912	38	2319	4.43
	Avg.					1849	3.41
(400.200)	1	12150	53	11455	53	5180	5.72
	2	12077	53	11577	54	3999	4.14
	3	12241	50	11734	51	4682	4.14
	4	12100	50	11261	50	3920	6.93
	5	12247	50	11674	51	7798	4.68
	Avg.					5116	5.12

Finally, we compare our program with the program written by Daskin (1995), called SITATION. As mentioned before, this program solves a number of uncapacitated facility location problems. One of them is the uncapacitated fixed charge location problem (UFCLP). This problem is very similar to the concentrator location, where the concentrators act as the facilities and the terminals act as the demand nodes. The only main difference is that the center does not exist in the UFCLP. However, by setting the connection cost between concentrators and the

central computer to 0, our program can mimic this problem. In addition, we also have to set the capacity of each concentrator to be large enough to handle the uncapacitated nature of the UFCLP. Moreover, all demand nodes should be considered as the potential location of the facilities.

Ten different data sets of size (150,150) were randomly generated in this study in order to make this comparison. The size of (150,150) was chosen because that is the largest size that can be managed by the SITATION program. The distances between nodes were randomly generated between 10 and 150. The demand of each node was randomly generated between 10 and 25, and the fixed cost for installing a facility was randomly generated between 500 and 1000.

The SITATION program provides several algorithms for solving a problem. All of them were applied to the data under study in order to get the best one. The two exchange-based algorithms gave the best results. These two algorithms differ only on how they calculate the initial solution. The first one was based on the ADD algorithm, whereas the second one used the DROP algorithm to obtain an initial solution. We then compared these results to the SA algorithm. The summary of this comparison is presented in Table 7.6. Two different multipliers for the simulated annealing, which were 2 and 6, were tested and their result are presented in this table along with their running times. All results represent the improvements gained over the results obtained by the ADD algorithm.

Table 7.6. Comparison of the Simulated Annealing Algorithm with the Exchange Algorithm (SITATION Program)

Rep	ADD	Exchange (ADD-based)		Exchange (DROP-based)		SA (Multiplier=6)			SA (Multiplier=2)		
	Cost	Cost	Imp (%)	Cost	Imp (%)	Cost	Imp (%)	Time (sec)	Cost	Imp (%)	Time (sec)
1	42105	40911	2.84	40822	3.05	40856	2.97	878	40891	2.88	374
2	43970	42372	3.63	42026	4.42	42103	4.25	962	42172	4.09	346
3	42965	41630	3.11	41812	2.68	41537	3.32	1102	41552	3.29	279
4	44795	43243	3.46	43475	2.95	43102	3.78	821	43217	3.52	372
5	43450	41523	4.43	41864	3.65	41222	5.13	975	41417	4.68	333
6	43402	42066	3.08	41606	4.14	41480	4.43	698	42060	3.09	480
7	43024	41748	2.97	41784	2.88	41472	3.61	625	41583	3.35	204
8	42674	42433	0.56	42331	0.80	42412	0.61	4063	42037	1.49	247
9	42657	42240	0.98	42354	0.71	42053	1.42	988	42305	0.83	215
10	42967	41437	3.56	41588	3.21	40945	4.71	827	40945	4.71	271
Avg.			2.86		2.65		3.42	1194		3.19	312

From Table 7.6, we can see that both methods of our simulated annealing algorithm performed better than the exchange algorithms did. On average, the simulated annealing algorithm with the multiplier of 6 could improve the result given by the ADD algorithm by about 3.42% (standard deviation = 1.44%). The average running-time was 1194 seconds (19.9 minutes). The simulated annealing algorithm yielded good results even for the multiplier = 2, in which case it could improve the ADD algorithm's results by about 3.19% (standard deviation = 1.25%). The average running-time was only 312 seconds (5.2 minutes).

7.3 Summary

This chapter has presented some computational experiments for the algorithms developed in this study. The results of the experiments showed the algorithms

developed in this study produced better results than did some existing heuristics for the data sets that we tested. The next chapter presents the conclusion of this study along with some future considerations.

CHAPTER 8

Conclusion

8.1 Research Results

The concentrator location and terminal assignment problems are considered as very difficult problems. Therefore, in most cases, it is highly unlikely to find polynomial time algorithms to solve them. In this study, we have successfully developed some heuristics based on simulated annealing to provide "good" if not optimal solutions to these problems. These have been shown to be better than the results obtained from some existing heuristics for the data sets that we tested.

The algorithm that we developed for solving the terminal assignment problems consists of two different methods for generating a neighbour, namely *CRS* (completely random selection) and *BRS* (best random selection) methods.[1] Recall that the *CRS* method chooses a neighbour randomly, whereas the *BRS* method searches for the best neighbour that can be generated from two randomly chosen concentrators. We have shown that for the problems generated in this study, the *BRS* method performed slightly better than did the *CRS* method. Therefore, the

[1] See section 5.1.1.2

BRS method was employed in the simulated annealing algorithm that was used for the experiments conducted in this study.

The smallest mean percentage improvement gained by the simulated annealing algorithm over the original greedy algorithm was 7.83% for the problems of size (100,20)[2] and the largest mean percentage improvement was 10.59% for the problems of size (400,80). When compared to the greedy algorithm modified to incorporate some tradeoffs, the lowest improvement was 4.02% for the problems of size (100,20) and the largest one was 6.99% for the problems of size (400,80).

Four methods for generating a neighbour have been tried in our simulated annealing algorithm for solving the concentrator location problem. Those are the *RC-RC* (both the concentrators to be dropped and to be added are randomly chosen), *LS-RC* (the concentrator to be dropped is the least significant concentrator, i.e., the one that offers the least contribution in lowering the configuration cost, and the concentrator to be added is randomly chosen), *RS-COM* (the concentrator to be dropped is randomly chosen, and the concentrator to be added is the closest the center of mass) and *LS-COM* (the concentrator to be dropped is the least significant one, and the one to be added in the closest to the center of mass) methods.[3] For the data generated in this study, the *RC-RC* method proved to be the best choice. Therefore, it was selected to be the method employed in the simulated annealing algorithms that were compared to some well-known heuristic algorithms.

The largest mean percentage improvement gained by our simulated annealing algorithm over the ADD algorithm was 5.12 % for the problems of size (400,200). This was the largest problem size tested in this study. Our algorithm was then

---

[2] 100 terminals and 20 (potential) concentrators.
[3] See section 5.1.2.2

compared to the best available algorithms present in the SITATION program.[4] The results showed that for the data generated in this study, our simulated annealing algorithm method produced better results than did the SITATION program.[5] The mean percentage improvement of our algorithm over the ADD algorithm was 3.42%, whereas the best algorithm of the SITATION program produced only 2.86% improvement.

8.2 Future Considerations

Although we have shown that the simulated annealing approach generated better results than did some existing heuristics, we believe that our implementations of the simulating annealing algorithm can still be improved in the following areas:

1. Annealing Schedule: Most of the decisions in the annealing schedules used in our algorithms, such as the upperbound on the number of iterations at each temperature, the maximum number of cost increasing transitions allowed at each temperature, the probability of accepting a cost increasing transition and the stopping criteria were determined through experimentation. Although improvements were obtained, we believe that more extensive schedule experimentation could yield even better results.
2. Data Structures: One possibility to reduce the programs' running times is by using more efficient data structures. Therefore, searching for better data structures is an important endeavour. One possible improvement that comes to mind is to use a linked-list instead of a Boolean matrix to represent a

[4] It was written by Daskin (1995) to solve a number of uncapacitated facility location problems.
[5] Ten different data sets of the size of (150,150) were used in this comparison.

configuration. This modification will accelerate the calculation of configuration costs. When it comes to calculating the cost of a configuration implemented in a Boolean matrix, one has to check every terminal for each concentrator in the configuration to see whether it is connected to the concentrator being considered or not. It is not necessary to do so if a linked-list is used, because each concentrator in the configuration only contains the terminals that are connected to it. Therefore, one can restrict oneself to those terminals in order to calculate the configuration cost. We believe that for large size problems, this modification can improve significantly the programs' running times. In addition, a linked-list occupies less computer's memory than does a Boolean matrix, so that it may be possible to solve larger problems.

3. Neighbour Generation: Another possible way to improve the algorithms in terms of the quality of the solutions and running times is by using a better method for generating a neighbour. We believe that by doing some more elaborate studies, a better method can be obtained. Moreover, the neighbourhood structure may also be modified.

4. A dummy concentrator for the terminal assignment program: We previously mentioned[6] that the algorithm used to compute an initial solution for the terminal assignment program, i.e., the modified greedy algorithm, cannot guarantee that a feasible solution can be found for every problem that has feasible solutions. We proposed to add a dummy concentrator with enough capacity to accommodate all unassigned terminals whenever the program fails to find a feasible solution. In the current application, users have to add this

[6] See section 5.1.1.1

dummy concentrator manually by themselves to the input data file. In future applications this should be an automated process of the program. Thus, whenever the program fails to find a feasible solution but the total capacity of the concentrators is larger than the total weight of the terminals, it should be able to create a dummy concentrator to accommodate all unassigned terminals.

References

Aarts, E., & Korst, J. (1989). Simulated annealing and Boltzmann machines: A stochastic approach to combinatorial optimization and neural computing. Chichester: John Wiley & Sons.

Abuali, F.N., Schoenefeld, D.A., & Wainwright, R.L. (1994). Terminal assignment in a communications network using genetic algorithms. Proceedings of 22nd ACM Computer Science Conference (pp. 74-81). Phoenix, Arizona.

Ball, M., & Magazine, M. (1981). The design and analysis of heuristics. Networks, 11, 215-219.

Boorstyn, R.R., & Frank, H. (1977). Large-scale network topological optimization. IEEE Transactions on Communications, COM-25, 29 -47.

Chardaire, P., & Lutton, J.L. (1993). Using simulated annealing to solve concentrator location problems in telecommunication networks. In R.V.V. Vidal (Ed.), Applied simulated annealing (pp. 175-199). Berlin, Germany: Springer-Verlag.

Cornuejols, G., Nemhauser, G.L., & Wolsey, L.A. (1990). The uncapacitated facility location problem. In P.B. Mirchandani & R.L. Francis (Eds.), Discrete location theory (pp. 119-171). New York: John Wiley & Sons, Inc.

Current, J.R., ReVelle, C.S., & Cohon, J.L. (1986). The hierarchical network design problem. European Journal of Operational Research, 27, 57-66.

Daskin, M.S. (1995). Network & discrete location: Models, algorithms and applications. New York: John Wiley & Sons, Inc.

Densham, P.J., & Rushton, G. (1992). A more efficient heuristic for solving large p-median problems. Papers in Regional Science: The Journal of the RSAI, 71, 307-329.

Doll, R.D. (1978). Data communication. New York: John Wiley & Sons Inc.

Eglese, R.W. (1990). Simulated annealing: A tool for operational research. European Journal of Operation Research, 46, 271-281.

Garey, M.R., & Johnson, D.S. (1979). Computer and intractability: A guide to the theory of NP-completeness. San Francisco: W.H. Freeman and Company.

Gavish, B. (1982). Topological design of centralized computer networks: Formulations and algorithms. Networks, 12, 355-377.

Gavish, B. (1985). Augmented langrangean based algorithms for centralized network design. IEEE Transactions on Communications, COM-33, 1247-1257.

Gavish, B. (1991). Topological design of telecommunication networks: Local access design methods. Annals of Operations Research, 33, 17-71.

Greene, J.W., & Supowit, K.J. (1986). Simulated annealing without rejected moves. IEEE Transactions on Computer-Aided Design, CAD-5, 221-228.

Jellet, P.M. (1990). Simulated annealing for a constrained allocation problem. Mathematics and Computers in Simulation, 32, 149-154.

Kershenbaum, A. (1993). Telecommunications network design algorithms. New York: McGraw-Hill, Inc.

Kershenbaum, A., & Boorstyn, R.R. (1983). Centralized teleprocessing network design. Networks, 13, 279-293.

Kirkpatrick, S. (1984). Optimization by simulated annealing: Quantitative studies. Journal of Statistical Physics, 34, 975-986.

Kirkpatrick, S., Gelatt, Jr., C.D., & Vecchi, M.P. (1983). Optimization by simulated annealing. Science, 220, 671-680.

Krarup, J., & Pruzan, M. (1990). Ingredients of location analysis. In P.B. Mirchandani & R.L. Francis (Eds.), Discrete location theory (pp. 1-54). New York: John Wiley & Sons, Inc.

Kuen, A.A., & Hamburger, M.J. (1963). A heuristic program for locating warehouses. Management Science, 9, 643-666.

Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller, A., & Teller, E. (1953). Equation of state calculations by fast computing machines. Journal of Chemical Physics, 21, 1087-1092.

Mirchandani, P.B., & Francis, R.L. (Eds.). (1990). Discrete location theory. New York: John Wiley & Sons, Inc.

Mirzaian, A. (1985). Langrangian relaxation for the star-star concentrator location problem: Approximation algorithm and bounds. Networks, 15, 1-20.

Mirzaian, A., & Steiglitz, K. (1981). A note on the complexity of the star-star concentrator problem. IEEE Transactions on Communications, COM-29, 1549-1552.

Narasimhan, S., & Pirkul, H. (1992). Hierarchical concentration location problem. Computer Communication, 15(3), 185-191.

Osborne, L.J., & Gillet, B.E. (1991). A comparison of two simulated annealing algorithms applied to the directed steiner problem on networks. ORSA Journal on Computing, 3(3), 213-225.

Pirkul, H., & Nagarajan, V. (1992). Locating concentrators in centralized computer networks. Annals of Operations Research, 36, 247-262.

Ramos, E., & Schroeder, A. (1994). Contemporary data communications: A practical approach. New York: Macmillan Publishing Company.

Rose, C. (1992). Low mean internodal distance network topologies and simulated annealing. IEEE Transactions on Communication, 40, 1319-1326.

Sechen, C. (1988). VLSI placement and global routing using simulated annealing. Boston: Kluwer Academic Publishers.

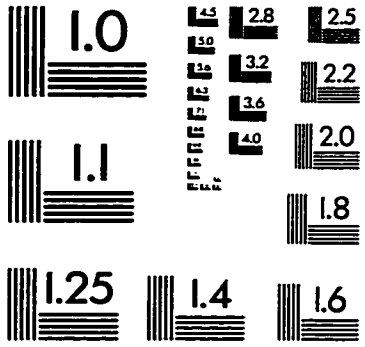
Stalings, W. (1994). Data and computer communications (4th ed.). New York: MacMillan Publishing Company.

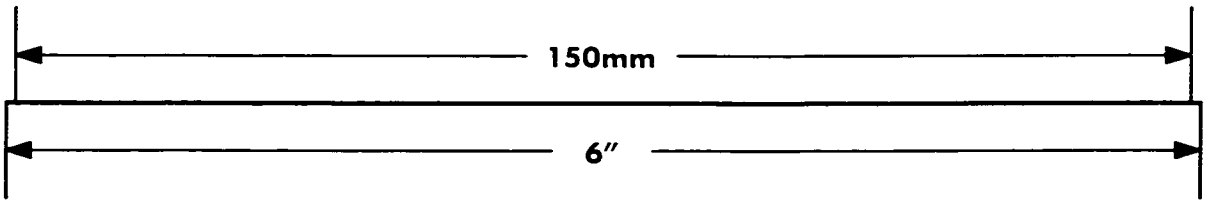
Van Laarhoven, P.J.M., & Aarts, E.H.L. (1987). Simulated annealing: Theory and applications. Dordrecht, The Netherlands: Kluwer Academic Publishers.

Vidal, R.V.V. (Ed.). (1993). Applied simulated annealing. Berlin, Germany: Springer-Verlag.

Wong, D.F., Leong, H.W., & Liu, C.L. (1988). Simulated annealing for VLSI design. Boston: Kluwer Academic Publishers.

TEST TARGET (QA-3)





APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989