1996

Heuristic algorithms for the terminal assignment problem

Teresa Li-Pei Chiu
San Jose State University

HEURISTIC ALGORITHMS

FOR

THE TERMINAL ASSIGNMENT PROBLEM

A Thesis

Presented to

The Faculty of the Department of Mathematics and Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Teresa Li-Pei Chiu

May, 1996

APPROVED FOR THE DEPARTMENT OF
MATHEMATICS AND COMPUTER SCIENCE

Sami Khuri

Dr. Sami Khuri

John Mitchem

Dr. John Mitchem

Frederick Stern

Dr. Frederick Stern

APPROVED FOR THE UNIVERSITY

Serena M. Stanford

ABSTRACT

HEURISTIC ALGORITHMS FOR THE TERMINAL ASSIGNMENT PROBLEM

by Teresa Li-Pei Chiu

In this research, applications of heuristic techniques for obtaining near optimal solutions for the *terminal assignment* (TA) *problem* are investigated.

The terminal assignment problem originates in the network industry, where the task is to assign terminals to concentrators in such a way that each terminal is assigned to one (and only one) concentrator and the aggregate capacity of all terminals assigned to any concentrator does not overload that concentrator. Under these two hard constraints, an assignment with the lowest possible cost is to be sought. The proposed cost is taken to be the distance between a terminal and a concentrator.

The intractability of this problem [6] is a motivation for the pursuit of heuristics that produce approximate, instead of exact, solutions. To our knowledge, other than the paper presented by Abuali et al. [2], no previous work on the TA problem has been published in the literature. Consequently, the test cases adopted for this study are solely based on the workbench as illustrated in [2].

The heuristic techniques that are incorporated in this research include greedy-based algorithms, genetic algorithms (GA) [1], and grouping genetic algorithms (GGA) [3]. The encouraging results may serve as a good indication of the applicability of these techniques on constrained problems such as the TA problem.

ACKNOWLEDGMENTS

To my mentor and my advisor, Dr. Sami Khuri, I give my heartfelt thanks for his technical expertise, his guidance, his encouragement, his endless patience, his dedication and joint passion for the topic, and in seeing this project to completion. I wish to thank Dr. Khuri for his insistence that a project of this magnitude was not beyond my reach. I will continue to aspire to Dr. Khuri's level of dedication to his students.

I owe a debt of thanks to Dr. John Mitchem and Dr. Frederick Stern for the valuable advice and discussions in refining this project.

Earning my Master of Science degree at San Jose State University has afforded me the opportunity to work with many excellent teachers. Along the way, I have also made some wonderful friends. Space will not allow me to thank every one whose influence contributed to this project. I wish to extend my deepest appreciation to all of them.

Finally, I would like to thank my mother and sisters who stood by, criticized and appreciated, and bore with undying patience, my excitements and depressions throughout the entire process.

Contents

Chapter 1 Introduction

The objective of the terminal assignment (TA) problem involves determining minimum cost links to form a network by connecting a given collection of terminals to a given collection of concentrators. The terminal sites and concentrator sites have fixed locations and are known. The capacity requirement of each terminal is known and may vary from one terminal to another. The capacities of all concentrators are known and, in this research, assumed to be the same. The cost of the link from each terminal to each concentrator is also known. The problem is now to identify for each terminal the concentrator to which it should be assigned, under two constraints, in order to minimize the total cost. The two constraints imposed on the TA problem are: (1) each terminal must be connected to one and only one of the concentrators, and (2) the aggregate capacity requirement of the terminals connected to any one concentrator must not exceed the capacity of that concentrator. The TA problem has been proven to be *NP-Complete* [22]. NP-Complete problems are problems that are not currently solvable in polynomial time. However, they are polynomially equivalent in the sense that any NP-Complete problem can be transformed into any other in polynomial time. Thus, if any NP-Complete problem can be solved in polynomial time, they all can [15].

In the literature of operations research, there are a number of combinatorial optimization problems that share the name "assignment problem" but whose underlying ideas are quite different from that of the terminal assignment problem. One such example is a special case of the *weighted matching problem* [25].

The weighted matching problem is a much more involved version of the matching problem and is defined as: Given a graph, $G = (V, E)$, and a weight for each edge, w: $E \rightarrow R^+$, find a matching of G with the maximum possible sum of weights. The weighted matching problem is considerably easier in its bipartite case. When the graph is bipartite, this problem is usually called the *assignment problem*, for it models the assignment of a set of tasks to a crew of workers. In that framework, the presence of an edge (u_i, v_j) indicates that worker i is capable of carrying out task j, and the weight of that edge represents the profit to be derived from assigning worker i to task j.

A slight variation of the above description is the *Hitchcock problem* [26], which is motivated by the following situation: We have m sources of some commodity, each with a supply of a_i units, $i = 1, 2, \ldots, m$, and n terminals, each of which has a demand of b_j units, $j = 1, 2, \ldots, n$. Furthermore, we know the cost c_{ij} of sending the commodity from source i to terminal j. The aim is now to satisfy the demands at minimum cost. When all the a_i and b_j are 1, the Hitchcock problem is called the *assignment problem.*

The objective of this research is to compare different heuristic algorithms applied on the TA problem. For this study's purpose, a greedy-based algorithm has been implemented as well as a genetic algorithm (GA) implemented under different GA packages. Genetic algorithms are powerful search algorithms based on the mechanisms of natural genetics. The potential of such algorithms to yield good solutions even for hard optimization tasks has been demonstrated by various applications [5, 12, 14]. This work illustrates the feasibility of applying GA on the TA problem. A recently developed area in the GA research, *grouping genetic algorithm* (GGA) [13], is also explored. The grouping genetic algorithms, as the name suggests, focus purely on problems with the grouping properties, i.e., performing tasks equivalent of grouping objects under certain restrictions. GGA adopts all the basic notions of GA with an alteration in the representation of strings (candidate solutions). This study reports and compares the results of the greedy algorithm with the results of the GAs performed on several problem instances.

The remaining of this work is organized in the following fashion. Chapter 2 illustrates the formal definition of the terminal assignment problem. We also give a small-sized example and detailed description of the structure and solution. Addressing the different heuristic algorithms adopted in this work, Chapter 3 first introduces the greedy technique, the general format of greedy algorithms and a specific implementation for the TA problem. In Chapter 4, genetic algorithms are presented. We discuss the GA operators and how the GAs differ from other algorithms. The GA

implementation of the TA problem is given in Chapter 5. The two GA packages adopted for this work, GENEsYs and LibGA, are presented in Chapter 6. Chapter 7 gives an introduction to the grouping genetic algorithms and the implementation of the TA problem. The test cases used in this research are described in Chapter 8. In Chapter 9 are the results and analyses of the performance of different heuristic algorithms applied on the test cases. Conclusions are drawn in Chapter 10.

Chapter 2 The Terminal Assignment Problem

The terminal assignment problem may arise in various contexts where one wishes to distribute (assign) individual objects to a collection of groups while trying to keep the cost as low as possible. The two constraints that have to be taken into consideration are (1) each object belongs to one and only one group, and (2) any group must be able to service the requirement (capacity) of all objects assigned to it.

2.1 Formal Definition

In the following, a formal definition of the terminal assignment problem is presented by making use of Stinson's terminology [33] for combinatorial optimization problems. We introduce concepts and notations that we use in subsequent sections of this work.

Problem instance:

Terminals : $l_1, \ l_2, \ \ldots, l_T$
Weights : $w_1, w_2, \ldots, w_T$

Concentrators : $r_1, r_2, \ldots, r_C$

Each of the C concentrators is of capacity W. w_i is the weight, or capacity requirement of terminal l_i. The weights and capacity are positive integers and $w_i < W$ for $i = 1, 2, \ldots, T$.

The T terminals and C concentrators are placed on the Euclidean grid, i.e., l_i has coordinates (l_{i1}, l_{i2}) and r_j is located at (r_{j1}, r_{j2}).

Feasible solution:

Assign each terminal to one of the concentrators such that no concentrator exceeds its capacity. In other words, a feasible solution to the terminal assignment problem is:

A vector $x = x_1\ x_2\ \ldots\ x_T$ where $x_i = j$ means that the ith terminal is assigned to concentrator j such that

$1 \le x_i \le C$ and x_i is an integer, for $i = 1, 2, \ldots, T$

{ all terminals have to be assigned }

$\sum_{i \in R_j} w_i \le W$, for $j = 1, 2, \ldots, C$ { capacity of concentrator is not exceeded }

where $R_j = \{ i \mid x_i = j \}$; i.e., R_j represents the terminals that are assigned to concentrator j.

Objective function:

A function $Z(x) = \sum_{i=1}^{T} cost_{ij}$, where $x = x_1\ x_2\ \ldots\ x_T$ is a solution and $x_i = j$, for $1 \le i \le T$, and $cost_{ij}$ = round $(\sqrt{(l_{i1} - r_{j1})^2 + (l_{i2} - r_{j2})^2})$, i.e., the result of rounding the distance between terminal i and concentrator j. In other words, $Z(x)$ denotes the overall cost of assigning individual terminals to concentrators according to the solution represented by x.

Optimal solution:

A feasible vector x that yields the smallest possible $Z(x)$.

2.2 Example

In a given problem instance of the TA problem, available information includes the numbers and locations of concentrator sites and terminal sites, the individual weight requirements of the terminals, and the uniform capacity of the concentrators. The cost of assigning each terminal to each concentrator is taken to be the Euclidean distance between the two locations.

Table 1 indicates a collection of $T = 10$ terminal sites and $C = 3$ concentrator sites. The weight requirement and the coordinates for each terminal site are specified in Table 1(a). The coordinates for the concentrator sites are listed in Table 1(b). Throughout this research, we assume the capacity of each concentrator to be $W = 12$.

Terminal	Weight	Coordinates
1	5	(54, 28)
2	4	(28, 75)
3	4	(84, 44)
4	2	(67, 17)
5	3	(90, 41)
6	1	(68, 67)
7	3	(24, 79)
8	4	(38, 59)
9	5	(27, 86)
10	4	(07, 76)

(a) Terminal capacity requirements (weight) and terminal coordinates.

Concentrator	Coordinates
1	(19, 76)
2	(50, 30)
3	(23, 79)

(b) Concentrator coordinates, capacity = 12.

Table 1

The coordinates are based on a 100 X 100 Euclidean grid; each of the terminals and concentrators occupies a fixed location on the grid. Different terminals may have different capacity requirements as specified in the above figure.

As introduced in the definition, the cost of connecting a terminal to a concentrator is obtained by rounding the Euclidean distance between the two sites to the nearest integer. Table 2 illustrates the costs of all possible connections from terminals to concentrators. For instance, the cost of assigning terminal 6 to the first concentrator is 49, to the second concentrator is 41, and to the third concentrator is 46.

terminal \ concentrator	1	2	3
1	59	04	59
2	09	50	06
3	72	36	70
4	76	21	76
5	79	41	77
6	49	41	46
7	05	55	01
8	25	31	25
9	12	60	08
10	12	62	16

Table 2. Terminal to concentrator cost (distance) matrix.

Figure 1 illustrates an assignment of the first 9 terminals which cannot be extended to the 10th terminal without introducing infeasibility; that is, none of the 3 concentrators is able to service the capacity requirement of terminal l_{10}. The total cost, computed as the sum of the costs for the 9 selected links, is 223.

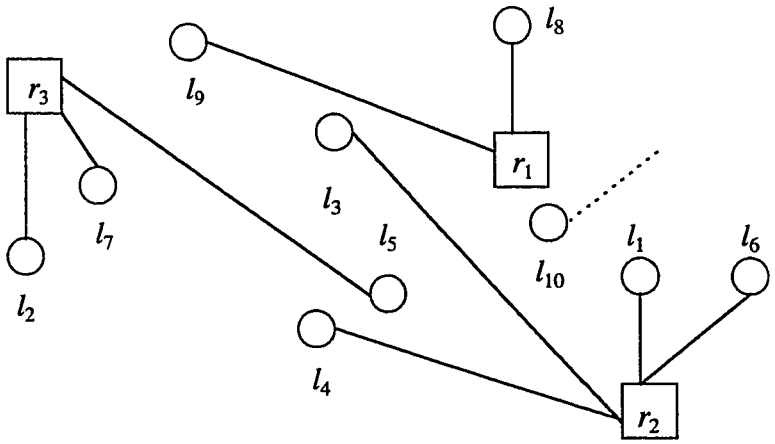

Figure 1. Terminal assignments to concentrators
Total cost = 223 with terminal 10 stranded.

In Figure 2, a feasible assignment of terminal sites to concentrator sites is shown. By exhaustive case analysis, it is the optimal terminal assignment for this specific problem instance. The total cost is 231, and is obtained by summing up the costs of the 10 links: the cost from terminal l_1 to concentrator r_2 is 4, from l_2 to r_3 is 6, from l_3 to r_1 is 72, from l_4 to r_2 is 21, from l_5 to r_2 is 41, from l_6 to r_2 is 41, from l_7 to r_3 is 1, from l_8 to r_1 is 25, from l_9 to r_3 is 8, and from l_{10} to r_1 is 12. Moreover, after the connections have been established, concentrator r_1 holds weight 12, r_2 holds weight 12, and r_3 holds weight 11. Therefore, all 10 terminals can be feasibly assigned without overloading any concentrator.

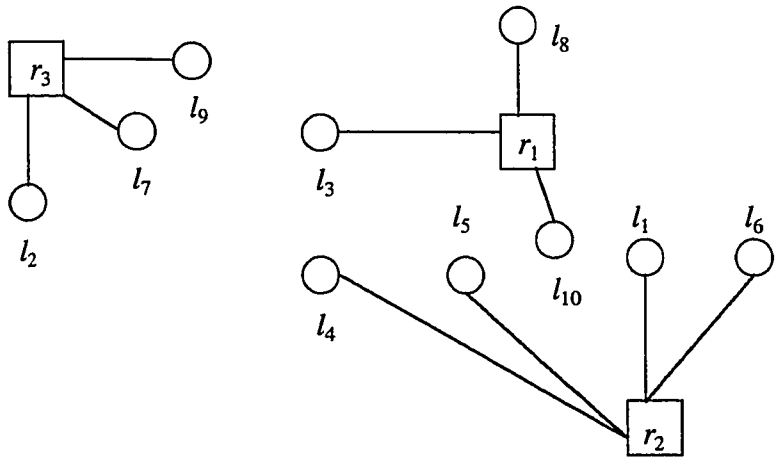

Figure 2. Terminal assignments to concentrators
Total cost = 231 with all terminals assigned.

It is conceivable that two different sites may occupy the same coordinates on the grid since the locations are completely randomly generated. We do not rule out this possibility in the sense that, in real practices, the grid is of a much larger scale. It is not uncommon to permit two sites to be placed in the same location, e.g., an office, with a negligible cost if they are to be connected. Furthermore, situations may also arise where a number of concentrators and terminals form a straight line on the grid such that, in order to connect two sites, the connection is bound to run into at least one unrelated concentrator or terminal. We note that, in general, solving this problem does not require major effort, and it is thus permissible in this study without any modification to the established definition.

The problem addressed in this research assumes that the terminals have unequal weight requirements. If one alternately assumes that all capacity requirements of the terminals are equal, then the problem is no longer NP-complete. The alternating chain

algorithms produce a provably optimal solution for the terminal assignment problem in the situation where the weight requirements of the terminals are all equal [22].

In the next chapter, our first heuristic strategy based on the greedy technique is introduced. We discuss the concepts behind and the strength of greedy-based algorithms, and describe an implementation of greedy heuristic for the terminal assignment problem.

Chapter 3 Greedy Algorithms

Algorithms for combinatorial optimization problems typically go through a sequence of steps, with a set of choices (guesses) at each step according to certain rules. Unfortunately, it is seldom possible to always make the correct choice throughout the entire sequence. In general, optimality of the sequence of guesses cannot be checked until most or all of the guesses have been made. In this chapter we consider the algorithms, called greedy algorithms, that make decisions solely on the basis of the information available at the time of the guess.

3.1 How Do Greedy Algorithms Work?

For each decision point, a *greedy algorithm* always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution. Since these choices are made on the basis of short-term calculations rather than on the basis of a concern about future choices, algorithms based on this approach are called greedy algorithms. The outline of a greedy algorithm can be sketched as follows:

```
Algorithm Greedy          {C is the set of all the candidates}
    S ← Φ                 {S is the set in which we construct the solution}
    while not solution (S) and C ≠ Φ do
        x ← an element of C maximizing (minimizing) select (x)
        C ← C - {x}
        if feasible (S ∪ {x})        {procedure "feasible" checks to
            S ← S ∪ {x}               see whether S ∪ {x} is feasible }
    end while
    if solution (S) return S   { if a solution is successfully constructed }
    else return "there is no solution"
end Greedy
```

Procedure "*select*" attempts to include an element in the solution under construction. Depending on the results given by *select*, the greedy algorithm includes the element yielding the largest (smallest) value.

The idea behind this strategy is that, as long as we have not exhausted all the possibilities, we continue to look for the current best choice. However, a slightly different strategy may sometimes be useful. Occasions may arise where the feasibility of a solution depends on the feasibility of every single element included in that solution; i.e., as soon as one element has been determined to be invalid, there is no need to check other possibilities and the algorithm can return an error flag immediately. Our implementation of the TA problem is an example of such cases; during the construction of a solution, if a terminal cannot be feasibly assigned to any concentrator, we are certain that the resulting solution can never be feasible—at least one concentrator is overloaded. Therefore, the algorithm aborts this attempt and starts to build the next candidate solution. The details are given in the next chapter.

The greedy method is quite powerful and works well for a wide range of problems, but it does not always produce optimal solutions. We consider the *knapsack problem* as an example. The knapsack problem gets its name from the problem of a backpacker who has a knapsack of fixed capacity M; a choice of n items to include, each of which is associated with a weight w_i, and a profit p_i which can be realized only if the item is included in the knapsack. The backpacker is then to decide which items to include in order to maximize the profit. There are two variants of this problem, depending on whether fractions of items can be included or not. If the answer is yes, the problem is called the rational knapsack problem. Otherwise, we call the problem the 0-1 knapsack problem, because the choice to be made is binary (0 or 1). A greedy algorithm can be devised to solve the rational knapsack problem, i.e., it will produce an optimal solution. However, no greedy algorithm can solve the 0-1 knapsack problem [33]: the greedy solution, the solution yielded by the greedy algorithm, is not necessarily an optimal solution. In general, the greedy solution is feasible but not guaranteed to be optimal.

The design of a greedy algorithm is often straightforward. Although the complexity will depend on the data structures used, we expect greedy algorithms to be fast since we only focus on one particular solution—there is no element of look-ahead or backtracking.

The greedy algorithm first appeared in the combinatorial optimization literature in an article by Edmonds [11] in 1971. More material on greedy algorithms can be found in Lawler [24] and Papadimitriou and Steiglitz [26].

3.2 Implementation of the TA Problem

A greedy-based heuristic algorithm for the TA problem is to assign terminals to the nearest available concentrators in a greedy fashion. Availability refers to the ability of the concentrator to service terminal capacity requirements. Namely, for each terminal, the algorithm looks for the concentrator that is closest to the terminal and checks if there is enough capacity to satisfy the requirement of the particular terminal. If there is, then the terminal is assigned to this concentrator; if the concentrator cannot handle the terminal, the algorithm looks for the next closest concentrator and performs the same evaluation. This process is repeated until an available concentrator is found and the algorithm continues to assign the remaining terminals, if there are any. Otherwise, no concentrator holds the required capacity and the attempt is declared a failure and the solution is infeasible. The assignments are carried out in a random fashion; i.e., the terminal to be connected next is selected at random. Specifically, a greedy algorithm [22] is

```
while additional assignments of terminals to concentrators are possible
{       for a randomly chosen terminal, say l_i
        {       determine cost_ij, the distance from l_i to r_j
                where r_j is the closest feasible concentrator for terminal l_i
                assign terminal l_i to concentrator r_j
        }
}
```

This greedy algorithm is likely to force the terminals that are considered last to be connected to concentrators that are very far away. For instance, Table 2 in Section 2.2 indicates that terminal l_5 is closest to concentrator r_2 with cost 41. However, as illustrated in Figure 1, l_5 is forced to be assigned to r_3 with cost 77. Again, terminal l_9 is closest to concentrator r_3 with cost 8, but is assigned to r_1 with cost 12. Therefore, we incorporate a random sequence of terminal connections each time to help explore the range of feasible assignments.

The greedy algorithm can fail to produce a feasible solution when: (1) the total concentrator capacity is less than the total terminal capacity requirement, (2) there is not a feasible solution to the problem instance, or (3) the algorithm misses the feasible solution(s). We consider Figure 1 in the previous chapter. This particular infeasible solution is obtained using the greedy algorithm outlined above; the exact solution may be reproduced by trying to assign the terminals in their numerical order as listed in Table 1(a) in Section 2.2. Table 1 indicates that the total concentrator capacity is sufficient for the total terminal capacity requirement—the concentrators may service a capacity requirement of 36 while the terminals altogether require 35. Moreover, Figure 2 illustrates that this problem instance does have at least one feasible solution. However, depending on the random order of terminals to be assigned, the algorithm may occasionally miss the feasible solutions, as in this case.

3.3 Variations

Abuali et al. [1] adopt a variation of the given greedy heuristic. A modification of the algorithm computes a "tradeoff" that can lead to assigning a terminal l_i to a given concentrator even in the presence of another closer unassigned terminal l_j. The modification may be advantageous if the alternative would be to attach l_j to the given concentrator and to subsequently attach l_i to a second concentrator that is much further away [22].

In this study, we abandon the idea of computing a tradeoff in order to illustrate the rudimentary principle of the greedy algorithms—the decisions are based purely upon previous decisions without any knowledge of future decisions.

At the beginning of this research, a second greedy heuristic was also implemented. The basic idea of this algorithm is as follows. All the connections from terminals to concentrators are first sorted according to their costs in a non-decreasing order. The algorithm then performs a traversal on the list of connections. If the connection involves a terminal that has not yet been assigned to any concentrator, it is immediately assigned to the one indicated by the connection if still available. If, however, the terminal has already been taken care of, or if the concentrator does not hold enough capacity to service the terminal, the algorithms ignores the connection and moves down to the next in the list. The process halts either when all terminals have been assigned, or when a terminal cannot be serviced by any concentrator.

Unfortunately, this seemingly plausible algorithm has shown to produce infeasible solutions for a majority of the test cases we adopt in this research. Therefore, we have excluded this heuristic from the rest of the study. This particular heuristic certainly agrees with the idea of the greedy strategy; however, the setup of this algorithm has made it so inflexible that there is no room for improvement of the solutions—there can only be one candidate solution, and once the algorithm determines some terminals cannot be assigned, there are no tools for the algorithm to remedy this problem. This is where the random order comes in handy as pointed out for the algorithm specified in the previous section; if an order does not work out, the algorithm can always proceed by employing another order. However, we note that the random order does not apply in this case for, if we use a random instead of a sorted list, we lose the essence of the greedy strategy to make the best choice at any given time.

In the following chapter we present our second heuristic technique—*genetic algorithms*. Since the emergence in the 1970s, genetic algorithms (GA) have been successfully applied to a number of combinatorial optimization problems as a heuristic technique for obtaining near optimal solutions. GAs are theoretically and empirically proven to provide robust search in complex spaces [16]. We will introduce fundamental genetic operators and how they make genetic algorithms different from other algorithms.

Chapter 4 Genetic Algorithms

Genetic algorithms (GA) have shown to be powerful general purpose adaptive search procedures in a variety of applications presented in literature. Genetic algorithms have been developed by John Holland [21], his colleagues, and his students at the University of Michigan. The goals of their research have been twofold: (1) to abstract and rigorously explain the adaptive processes of natural systems, and (2) to design artificial systems software that retains the important mechanisms of natural systems. This approach has led to important discoveries in both natural and artificial systems science.

Genetic algorithms are search algorithms based on the mechanics of natural selection and natural genetics. They combine survival of the fittest among string structures with a structured yet randomized information exchange to form a search algorithm with some of the innovative flair of human search. The algorithm proceeds in steps called generations. In every generation, a new set of strings is created using bits and pieces of the fittest of the old; an occasional new part is tried for good measure. Randomized genetic algorithms are different from random walks. Instead, they efficiently exploit the search space to speculate on new search points with expected improvement of performance. One fundamental premise of genetic algorithms is that they can solve complicated problems by simulating evolution.

4.1 How Do Genetic Algorithms Differ from Traditional Methods?

Genetic algorithms are different from traditional optimization and search procedures in several ways:

- GAs work with encodings for some parameter set rather than with the parameters themselves.
- GAs search from a population of points, not a single point.
- GAs use objective function information rather than some other auxiliary knowledge, e.g., derivatives.
- GAs use probabilistic, instead of deterministic, transition rules and application of the genetic operators causes information from the previous generation to be carried over to the next.

Genetic algorithms manipulate decision or control variable representations at the string level to exploit similarities among high-performance strings while other methods usually deal with functions and their control variables directly. Because GAs operate at the encoding level, they are difficult to fool even when the function may be difficult for traditional schemes. GAs work from a population while many other methods work from a single point. In this way, GAs find safety in numbers. By maintaining a population of well-adapted sample points, the probability of reaching a false peak is reduced. GAs achieve much of their breadth by ignoring information except that concerning payoff. Other methods rely heavily on such information, and in problems where the necessary information is not available or difficult to obtain, these other

techniques break down. GAs remain general by exploiting information available in any search problem. Genetic algorithms process similarities in the underlying encoding together with information ranking the structures according to their survival capability in the current environment. Last but not least, the transition rules of genetic algorithms are stochastic; many other methods have deterministic transition rules. A distinction exists, however, between the randomized operators of GAs and other methods that are simple random walks. Even though it may seem unusual to use chance to achieve directed results (the best points), GAs use random choice to guide a highly exploitative search.

Goldberg [16] and Holland [21] are good references on genetic algorithms for interested readers who would like to know more about GAs. They provide detailed explanations of the GA strategy and actual implementations of selected GA operators.

4.2 Simple Genetic Algorithms

A genetic algorithm is an iterative procedure which borrows the ideas of natural selection and 'survival of the fittest' from natural evolution. By simulating natural evolution, GAs can easily solve complex problems. Furthermore, by emulating biological selection and reproduction techniques, GAs can effectively search the problem domains in a general, representation-independent manner.

The genetic algorithm maintains a population or pool of candidate solutions for a given objective function. The candidate solutions represent an encoding of the problem into a form that is analogous to the chromosomes of biological systems. Each

chromosome is made up of a string of genes. The chromosome is typically represented in the GA as a string of bits. However, integers and floating point numbers have been used. The chromosome also has associated with it a fitness value, which is obtained by evaluating the chromosome with the objective function. It is the fitness of a chromosome which determines its ability to survive and produce offspring for the next generation. We consider the example of TA problem described in Section 2.2. Figure 3 indicates the chromosome representing the optimal solution.

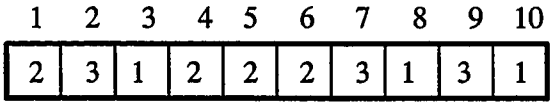

Figure 3. Chromosome representing the optimal solution to the example in Section 2.2.

The value carried by position i of the chromosome specifies the concentrator that terminal i is to be assigned to; i.e., terminal 1 is assigned to concentrator 2, terminal 2 to concentrator 3, terminal 3 to concentrator 1, and so on.

Once an initial pool has been generated and all of its members have been evaluated, the genetic algorithm begins its emulation of the life cycle. At each step in the iteration, chromosomes are probabilistically selected from the population for reproduction. Offspring are generated through a process called crossover. Crossover may be accompanied by another process called mutation, which randomly alter the values (with small probability) in the chromosomes. The offspring are then placed back in the pool, perhaps replacing other members of the pool. This process can be modeled using either a 'generational' [16, 21] or a 'steady-state' [34] genetic

algorithm. The generational GA saves offspring in a temporary location until the end of a generation has been reached. At that time the offspring replaces the entire current population. On the other hand, the steady-state GA immediately places offspring back into the current population.

This informal description leads to the rough outline of a genetic algorithm given below [19, 23]:

Algorithm *GA* **is**
    $t \leftarrow 0$
    initialize $P(t)$
    evaluate $P(t)$
    while not *terminate* $P(t)$ **do**
        $t \leftarrow t + 1$
        $P(t) \leftarrow$ *select* $P(t - 1)$
        crossover $P(t)$
        mutate $P(t)$
        evaluate $P(t)$
    end while
end *GA*

Genetic algorithms rely on genetic operators for selection, crossover, mutation, and replacement. The selection operators use the fitness values to select a portion of the population to be parents for the next generation. Parents are combined using the crossover and mutation operators to produce offspring. This process combines the fittest chromosomes and passes superior genes to the next generation, thus providing new points in the search space. The replacement operators ensure that the 'least fit' or weakest chromosomes of the population are displaced by more fit chromosomes.

While the fundamental concepts of genetic algorithms are fairly simple and straightforward, there are numerous implementation variations and options to incorporate into a genetic algorithm [16]. For instance, there are various ways to parameterize a model and encode it into a finite length chromosome. There are a number of different selection techniques for determining chromosomes for crossover. There are numerous possible crossover operators that have been developed in recent years depending on the problem type and chromosome encoding scheme. There are also several techniques for introducing some random changes to a chromosome, i.e., mutation.

4.3 GA Operators

The mechanics of a simple genetic algorithm are surprisingly straightforward, involving nothing more complex than copying strings and swapping partial strings. Simplicity of operation and power of effect are two of the main attractions of the genetic algorithm approach. The genetic operations that shall be introduced in the following sections take an initial population and generate successive populations that (in general) improve over time. A simple genetic algorithm that yields good results in many practical problems is composed of three operators:

- reproduction
- crossover
- mutation

4.3.1 Reproduction

Reproduction is a process in which individual strings are copied according to their objective (fitness) function values. Intuitively, we can think of the function as some measure of profit (cost) that we desire to maximize (minimize). Copying strings according to their fitness values means that the strings with a better value have a greater chance of contributing one or more offspring in the next generation. This is accomplished with an artificial version of natural selection, a Darwinian survival of the fittest among strings.

The reproduction operator may be implemented in algorithmic form in a number of ways. Perhaps the easiest is to create a biased roulette wheel where each current string in the population has a roulette wheel slot sized in proportion to its fitness value. Table 3 lists a sample population of four strings and their respective fitness values. Summing the fitness over all four strings, we obtain a total of 1168. The percentage of population total fitness is also shown in the table. Figure 4 depicts the distribution of the population on a weighted roulette wheel.

No.	String	Fitness	Percentage
1	01101	231	19.8
2	11000	422	36.1
3	01000	154	13.2
4	10011	361	30.9
Total		1168	100.0

Table 3. Sample problem strings and fitness values.

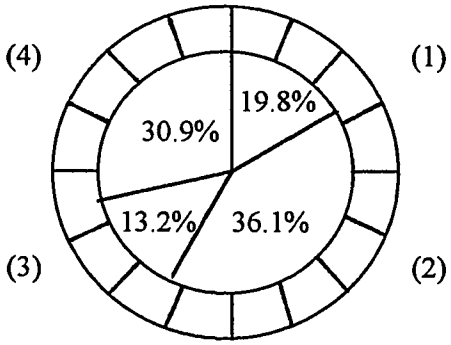

Figure 4. Simple reproduction allocates offspring strings using a roulette wheel with slots sized according to fitness.

Each time we require another offspring, a spin of the weighted roulette wheel yields the reproduction candidate. In this way, more highly fit strings have a higher number of offspring in the succeeding generation. Once a string has been selected for reproduction, an exact replica of the string is made. This string is then entered into a mating pool, a tentative new population, for further genetic operator action.

4.3.2 Crossover

After reproduction, simple crossover may proceed in two steps. First, members of the newly reproduced strings in the mating pool are mated at random. Second, each pair of strings undergoes crossing over as follows: an integer position k along the string is selected uniformly at random between 1 and the string length minus one [1, l-1]. Two new strings are created by swapping all characters between positions $k+1$ and l inclusively. Namely, the mechanics of crossover are in the form of partial string exchanges. For instance, consider two sample strings A_1 and A_2 from the Table 3 in the previous section:

$A_1 = 0\ 1\ 1\ 0 \mid 1$
$A_2 = 1\ 1\ 0\ 0 \mid 0.$

Suppose in choosing a random number between 1 and 4, we obtain a $k = 4$ (as indicated by the separator symbol $\mid$). The resulting crossover yields two new strings where the prime (') means the strings are part of the new generation:

$A'_1 = 0\ 1\ 1\ 0\ 0$
$A'_2 = 1\ 1\ 0\ 0\ 1$

Thus, A_1' and A_2' are the offspring of A_1 and A_2.

We note that the crossover operation is performed under the control of a predefined crossover rate. In other words, crossover is not necessarily carried out every time a new offspring is being generated. If the probability determines that crossover is not performed, a replica of the parent is made and becomes the resulting child.

The mechanics of reproduction and crossover are surprisingly simple, involving random number generation, string copies, and some partial string exchanges. The combined emphasis of reproduction and the structured, though randomized, information exchange of crossover give genetic algorithms much of the power. Consider a population of n strings. Substrings within each string contain various notions of what is important or relevant to the task. Viewed in this way, the population contains not just a sample of n ideas; rather, it contains a multitude of notions and rankings of those notions for task performance. Genetic algorithms ruthlessly exploit this wealth of information by (1) reproducing high-quality notions according to their performance,

and (2) crossing these notions with many other high-performance notions from other strings. Thus, the action of crossover with previous reproduction speculates on new ideas constructed from the high-performance building blocks of past trials.

4.3.3 Mutation

Mutation plays a decidedly secondary role in the operation or genetic algorithms. Mutation is needed because, even though reproduction and crossover effectively search and recombine existing notions, occasional they may become overzealous and lose some potentially useful genetic material. In artificial genetic systems, the mutation operator protects against such an irrecoverable loss. In the simple GA, mutation is the occasional (with small probability) random alteration of the value of a string position. If the binary coding is employed, this simply means changing a 1 to a 0 and vice versa. By itself, if only mutation is used, it is a random walk through the search space. When used sparingly with reproduction and crossover, it is an insurance policy against premature loss of important notions.

We claim that the mutation operator plays a secondary role in the simple GA by noting that the frequency of mutation to obtain good results in empirical genetic algorithm studies is on the order of one mutation per thousand bit (position) transfers. Mutation rates are similarly small (or smaller) in natural population, leading to the conclusion that mutation is appropriately considered as a secondary mechanism of genetic algorithm adaptation.

Other genetic operators and reproductive plans have been abstracted from the study of biological example. However, the three GA operators examined in this section—reproduction, crossover, and mutation—have proved to be both computationally simple and effective in tackling a number of important optimization problems [16].

Genetic algorithms are intrinsically highly parallel algorithms. Holland's earliest speculative work [20] recognized the parallel nature of the reproductive paradigm and the inherent efficiency of parallel processing. However, GAs have not received great attention in the parallel processing literature considering the parallel nature of natural genetic systems. Until recently very little work has been performed in mapping genetic algorithms to existing and proposed parallel hardware. Theoretical and implementation efforts are just now starting to receive increased attention.

In the next chapter, we present our GA implementation for the TA problem—the chromosome representation with standard crossover and mutation operators.

Chapter 5 GA Implementation

In this research, we adopt a straightforward implementation for a genetic algorithm. We use the simple chromosome representation as briefly mentioned in Section 4.2, which is a mapping from the integers 1 through T, the number of terminal sites, to the integers 1 through C, the number of concentrator sites. Standard crossover and mutation operators are incorporated. A generational genetic algorithm strategy with roulette wheel selection is used for evolving the populations.

5.1 Chromosome Representation

The first step for the GA implementation involves choosing a represen-tation for the problem. We use non-binary strings of length T, say $s_1\, s_2 \ldots s_T$, where the value of s_i represents the concentrator to which the ith terminal is assigned. This representation complies with the implementation of a simple genetic algorithm in that a chromosome is built on a one gene for one object basis. This strategy can easily be implemented under any GA package that permits integer permutations, such as LibGA. However, for the sake of comparison, we have also adopted a second GA package, GENEsYs, which allows only binary representations for strings. In this case, some mapping from binary strings to non-binary strings is necessary. For instance, Figure 3 in Section 4.2 indicates a chromosome of length 10 in integer representation. The equivalent chromosome in binary representation used by GENEsYs is illustrated in Figure 5.

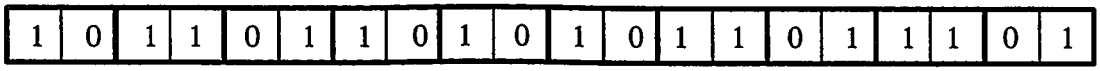

Figure 5. Chromosome using binary representation as incorporated in GENEsYs

Every two bits together represent one allele in the original chromosome; the first two bits, 1 and 0, represent the 2 in the first position of the chromosome in Figure 3, and the next two bits, 1 and 1, represent the 3 in the second position. We should note that, using the binary representation, the chromosome length is longer than that using integer representation since it takes more than one bit for integers greater than 1. In this case, the chromosome is twice as long for the largest integer used is 3, which requires two bits when converting.

Another difficulty, nevertheless, arises when working with GENEsYs using this particular representation. Consider a problem instance with 100 terminal sites and 27 concentrator sites. Using the binary scheme, it takes five bits to represent a concentrator number, ranging from 1 to 27, for any terminal. However, since we have no control over the random number generator and five bits may well represent any number between 1 and 32 inclusively, this suggests that it is likely for a chromosome to consist of not only infeasible but invalid information; e.g., concentrator 30 does not exist in this case. Therefore, it is essential to make sure that the elements in the chromosomes are all valid. In case of an invalid concentrator number, we remedy this problem by forcing the program to select another random number as a replacement until a valid number is generated. The necessity of this repair method does, however, affect the performance of the GA.

5.2 GA Operators

Throughout this study, we use straightforward genetic operators such as roulette wheel selection, uniform crossover, and uniform mutation.

When selecting two parents for generating new offspring, we incorporate the roulette wheel selection technique as presented in Section 4.3.1; i.e., each string occupies a slot on the roulette wheel according to its fitness value, a spin of the wheel thus yielding a string in proportion to its probability. However, we should note that, since the TA problem is a minimization problem, strings with smaller fitness values are considered better. Therefore, a slight modification is required for this strategy. Instead of allocating spaces on the roulette wheel to strings proportional to their fitness values, we allocate each string a slot of size *inversely* proportional to the fitness value. Hence, we can guarantee that a string with a smaller fitness value possesses greater possibility of being selected using this technique.

We take a look again at the four strings given in Table 3 in Section 4.3.1. The fitness values of these strings are 231 (19.8%), 422 (36.1%), 154 (13.2%), and 361 (30.9%), respectively. In case of a maximization problem, the percentages in parentheses are also the percentages they receive on a roulette wheel. On the other hand, if these four strings are used in a minimization problem, they are assigned sectors inversely proportional to their fitness values. More specifically, instead of using the fitness values of the strings, we use the reciprocals of these values when carrying out the computation. Hence a smaller fitness value is associated with a larger sector and

vice versa. The percentages the strings now receive are 27.1, 14.8, 40.7, and 17.4. Figure 6 gives an illustration of the distribution.

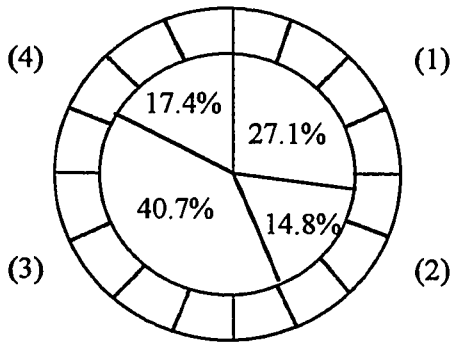

Figure 6. Roulette wheel with slots sized inversely proportional to fitness.

After we have selected the parent strings, we proceed by performing uniform crossover. Namely, all alleles are assigned the same probability and are swapped uniformly. One drawback of employing this technique is that it does not preserve the order in the strings for alleles in different positions in the strings can still be exchanged freely. This problem does have an impact on the TA implementation since the orders are important in the sense that a terminal is not the same as any other terminal—all terminals have different locations. However, we choose to work with uniform crossover, the most straightforward and widely used strategy, for the sake of comparisons between different GA packages.

Our implementation of the mutation operator is identical to that described in Section 4.3.3. Mutation is achieved by traversing the strings and altering the values of some positions according to a pre-defined probability. It is a common practice to use

the reciprocal of the string length as the mutation rate. We will see in later sections that this strategy does not apply well to our implementation.

5.3 Fitness Function with Penalty

Unlike many other approaches that handle constrained optimization problems by a knowledge-based restriction of the search space to feasible solutions, our approach uses a penalty function to cope with constraints. In other words, rather than ignoring the infeasible regions and concentrating only on feasible ones, we do allow infeasibly bred strings to join the population, but for a certain price. A penalty term incorporated in the fitness function is activated, thus reducing the infeasible string's strength relative to the other strings in the population. If the penalty is too harsh, the search is similar to the one that discards infeasible strings. On the other hand, a very mild penalty fails to pressure the search towards feasible solutions. In designing fitness functions for problems that do have infeasible strings, we follow the suggestion found in [23] and make use of the following two principles:

- The fitness functions use graded penalty functions. Two infeasible strings are not treated equally. The penalty is a function of the distance from feasibility. It has been argued that such penalty functions generally outperform other modes of penalties [29].
- The best infeasible string cannot be better than the weakest feasible string. Thus, our fitness function always has an offset term to ensure the strict order between feasible and infeasible strings.

The fitness function implemented for the TA problem is the sum of two terms:

(1) the objective function as given in Section 2.1 which calculates the total cost of all connections. Namely, $Z(x) = \sum_{i=1}^{T} cost_{ij}$, where $x = x_1 x_2 \ldots x_T$ is a solution.

(2) a penalty function used to penalize infeasible strings, where the weight of the penalty reflects the excessive load of the concentrators.

The penalty function employed here consists of the sum of two parts. The first part is the product of the number of terminals and the maximum distance on the grid, which forces an infeasible solution to have a fitness value greater than the largest possible sum of costs. More specifically, the best infeasible solution will always have a greater fitness value than will the worst feasible solution. The second part is the product of the sum of excessive load of concentrators and the number of concentrators that are in fact overloaded. This term imposes a heavier penalty on infeasible solutions with greater excessive load and/or more overloaded concentrators, thus differentiating the degrees of infeasibility among strings.

In the following chapter, we give an overview of the two GA software packages used in this research—GENEsYs and LibGA. We discuss some of their unique features and the options they provide to the users.

Chapter 6 GA Packages

Over the years, there have been several software packages developed that provide a workbench for genetic algorithm research. The majority of these packages use the generational model inspired by GENESIS. A few have adopted the steady-state model introduced in Genitor.

The first widely available genetic algorithm was GENESIS [18], written by John Grefenstette in 1984. Since that time, a variety of genetic algorithm packages haven been developed. Most of these use the generational model. However, a few incorporate the steady-state model introduced with Genitor [34] in 1988. GAucsd [30] is a GENESIS-based genetic algorithm. It offers several bug fixes and an improved user interface. It can additionally distribute experiments over a network of machines. Like GENESIS, GAucsd was designed primarily for use on bit string type chromosomes, which does not work well with order-based problems [6]. While GAucsd can encode the integers using grey-code bit strings, it does not ensure that order is preserved. This can produce chromosomes with invalid allele orderings or alleles with meaningless bit patterns. GAucsd uses a variant of roulette for selection, a two-point crossover, and a bit inverting mutation.

Genitor is a steady-state GA which uses a rank-based, biased selection and weakest chromosome replacement. It has the ability to work with bit, integer and floating point types. However, steady-state genetic algorithms can converge

prematurely. They require large pool sizes and many trials to increase the probability that good solutions are found. Convergent behavior without guarantee of optimality bothers many people who approach genetic algorithms from other, more traditional, optimization backgrounds. There are ways devised to slow down *premature convergence*, but the fact is that genetic algorithms have no convergence guarantees in arbitrary problems. They do sort out interesting areas of a space quickly, but they are a weak method, without the guarantees of more convergent procedures. One solution to this problem is to have the GA ferret out the best regions, then take a locally convergent scheme and climb the local peaks. In this way, one can combine the globality and parallelism of the GA with the more convergent behavior of the local technique.

In this research, we implement genetic algorithms with two existing GA software packages: GENEsYs and LibGA; the former incorporates binary string representation of chromosomes, while the latter provides integer representation.

6.1 GENEsYs

Based on GENESIS 4.5, the GENEsYs 1.0 software package is implemented by Thomas Bäck [3]. Compared to the standard genetic algorithm as defined by Holland in 1975, GENEsYs incorporates some extensions that we shall introduce here. These extensions are developed to perform some experiments with genetic algorithms, and for testing some features which originally stem from Evolution Strategies [28, 31] in the framework of genetic algorithms. The code is a mixture of the original lines written by

Grefenstette and many changes done by Bäck. The software is implemented in C and runs under the UNIX operating system.

Comparing GENESIS 4.5 and GENEsYs, the following extensions incorporated in GENEsYs are most remarkable:

- Either command line options or the setup program can be used for invoking the GA.
- Enhanced data collection features are provided.
- A function table is used, from which the user chooses an objective function when invoking the GA.
- Several extension of the basic GA are implemented, e.g. m-point crossover, uniform crossover, discrete and intermediate recombination, adaptive mutation rates, and Boltzmann selection.

6.1.1 Interface

A typical call of the standard GA is as follows:

ga -P 500 -R 0.01 -C 0.6 -f 37 { -f 200.dat } -t 10000 -e 10 -E &

The options in the above example call configurate the GA to work with a population size of 500 strings (option **P**), mutation rate 0.01 (option **R**), and crossover probability 0.6 (option **C**). In addition, objective function number 37 is selected by option **f**, and a special parameter is passed to f_{37} by the **{ -f 200.dat }** construction. In this particular case, the special parameter is an input file containing one problem instance. Ten independent optimization experiments will be performed (option **e**), each running for

10,000 function evaluations (option **t**). Option **E** enables the *elitism* mechanism, which will be explained in a later section.

6.1.2 Selection

A number of selection schemes are implemented in GENEsYs in order to allow a comparison of different mechanisms. These special selection schemes include the standard *proportional selection* [21], *linear ranking* [4], *Whitley's linear ranking* [35], *uniform ranking* [31], *uniform ranking with copying*, *inverse linear ranking*, and *Boltzmann selection* [17]. Detailed descriptions of these schemes are provided in [2].

In general, each of the selection mechanisms can be used either in their *preservation* or *extinctive* version. In preservation selection schemes, each individual is assigned a non-zero probability of being selected. On the other hand, an extinctive scheme does not allow some individuals to be selected by assigning them a selection probability of zero; these will generally be the worst individuals. The user explicitly defines the degree of extinctiveness of the GA during the startup process. The default mechanism is preservation, but the number of individuals taken into account by selection can be set by the user.

Furthermore, each selection scheme can be turned into an *elitist* one, thus guaranteeing the best individual always gets to be copied at least once to the next generation. The elitist selection strategy stipulates that the best performing structure always survives from one generation to the next. In the absence of this strategy, it is possible that the best structure disappears due to crossover and/or mutation. However,

as De Jong [8] has pointed out, elitism improves local search at the expense of global perspective.

6.1.3 Crossover

Among adjacent pairs of structures in the new population the crossover operator exchanges information (the population is randomly shuffled during the selection phase). In addition to the usual *one-point crossover* [21] which ex-changes information between individuals starting at a position chosen at random, the generalizations to *m-point crossover*, *uniform crossover*, *discrete recombination*, and *intermediate recombination* are also available in the GENEsYs software package. Note that the latter two schemes are taken from Evolution Strategies [31].

The crossover operators can not only be applied to bits which constitute the alleles of an individual, but also to information which encodes mutation rates as will be described in the following section.

6.1.4 Mutation

After the new population has been generated, the mutation operation is applied to each chromosome in the new population. In addition to the commonly used standard mutation mechanism, GENEsYs also provides some adaptive mutation schemes.

Using the standard mutation mechanism, each position $a_i \in \{0, 1\}$ of an individual $\{a_1, a_2, \ldots, a_l\}$ is given a probability p of undergoing mutation [21], where the default value of p is 0.001. If mutation does occur, a random value is chosen from $\{0, 1\}$ for the selected position.

The general idea of adaptive mutation is to incorporate the mutation probability into the individuals' genotype and to allow for an adaptation of mutation rates by the same mechanisms of selection, recombination, and mutation as used for adaptation of the alleles. To accomplish this, each individual is extended by bits encoding either one or more mutation rates. Each mutation rate is encoded by l bits, where the default value of l is 20. The genetic information of the mutation rates is initialized at random.

6.1.5 Options

GENEsYs may be started by using one of two different methods. The first one is based on a program called "setup", which explicitly asks for each parameter of the GA to be input by the user. Default values are announced during this process. Alternatively, all options may be given to the algorithm by specifying them on the command line when calling the program **ga**. In general, this method is more convenient to the user because the complete setup procedure, during which all possible parameter and configuration options have to be passed, is rather a lengthy task. Using command line options, defaults remain unchanged, and the user must only specify the options s/he wishes to modify.

In the following we illustrate some pertinent options:

- General Options

  -e number of independent optimizations
  -f objective function to be optimized
  -h on-line help information
  -t total number of function evaluations per experiment

- GA-Specific Options

  -C crossover rate
  -E elitist selection strategy
  -M mutation scheme
  -P population size
  -R mutation probability
  -S selection scheme
  -Y crossover scheme

6.1.6 Report

A report summarizing the performance of the GA is automatically generated by the "report" program. The report contains the mean, variance and range of several measurements, including on-line performance, off-line performance, the average performance of the current population, and the current best value. On-line performance is the mean of all evaluations, and off-line performance is the mean of the current best evaluations [9].

6.2 LibGA

Incorporating many of the different options and variations of already existing GA packages, LibGA is developed specifically for order-based problems by Arthur L. Corcoran and Roger L. Wainwright [6]. LibGA includes a variety of genetic operators for reproduction, crossover, and mutation. Routines are provided which implement both generational and steady-state genetic algorithms using the genetic operators. Other routines are provided for initialization, reading the configuration file, and generating reports. Moreover, LibGA offers a unique feature of a dynamic generation gap. Similar to GENEsYs, LibGA is also implemented in C under a UNIX platform.

6.2.1 Interface

In the main function, **GA_config**("ga.cfg", obj_fun) is first called to initialize and configure the global parameters as specified in the configuration file. A pointer to these parameters is returned and assigned to **ga_info**. The argument *"ga.cfg"* is the path-name of the configuration file and *obj_fun* is a pointer to the user's objective function. **GA_run**() is then invoked to carry out the genetic algorithm using the parameters in **ga_info**.

The user's objective function is called whenever the genetic algorithm needs to evaluate a chromosome. A pointer to the chromosome in question is passed to the objective function. The objective function must decode the chromosome (if necessary), compute the fitness value, and assign that value to the chromosome.

The recommended way to change LibGA's configuration from the defaults is through the use of a configuration file. The file is parsed by LibGA using a free format. Initially, the entire file contains comments (text following a '#' character), meaning all options are supplied by default values. The user need only remove the '#' character to use the new directive to change the configuration for a program.

6.2.2 Selection

The selection mechanisms provided by LibGA are *uniform-random*, *roulette*, and *rank-biased*. Uniform-random selection simply picks a member of the pool at random, completely ignoring fitness or other factors. Each chromosome in the pool is equally likely to be selected. Roulette selection is the classical selection operator for

generational genetic algorithms as described in Goldberg [16]. Each member of the pool is assigned space on a roulette wheel proportional to its fitness value. The members with the best (max or min) fitness value have the highest probability of being selected. Rank-biased selection is the selection method used in Genitor. The pool is sorted by fitness value and chromosomes are selected using a selection bias parameter. The bias (ranging form 1.0 to 2.0) specifies the amount of preference to be given to the best members of the pool.

Also available in the LibGA package are different replacement mechanisms: *append*, *by-rank*, *first-weaker*, and *weakest*. Used in the classical generational GA to place offspring in the new pool, the append replacement operator appends new chromosomes to an existing pool. The by-rank operator works with a pool ranked by sorted fitness values. If the chromosome has a good enough fitness, it is placed in the pool. In the case of the weakest and first-weaker operators, a chromosome may only be placed in the pool if it can find a weaker chromosome to displace. The first-weaker operator performs a linear scan through the pool to find a weaker chromosome and replaces the first encountered, while the weakest operator replaces the weakest member of the pool with the new chromosome unless the new chromosome is itself weaker.

6.2.3 Crossover

Simple crossover is used for traditional bit string encodings of the chromosome. A random crossover point is selected which divides each parent into two parts. Alternate parts are contributed by each parent to generate two offspring. This is also

known as single point crossover. This does not work for order-based problems as order is not being preserved. Other crossover operators included in LibGA are: *order1*, *order2*, *position*, *cycle*, *PMX*, and *asexual*, where all serve the purpose of preserving order information. The order1, order2, position, cycle, and PMX operators are discussed in Starkweather et al. [32]. The asexual operator is a simple swap of two randomly selected genes.

6.2.4 Mutation

LibGA offers 3 different mutation operators: *simple-invert, simple-random*, and *swap*. The simple-invert and simple-random operators are used in bit string chromosome representations. They both mutate at random, based on the mutation rate. They also randomly pick the bit to be mutated. The difference lies in that simple-invert inverts the bit and simple-random selects the value randomly. Since the random selection could choose the same bit value, one would expect simple-random to invert the bit only half of the time that mutation occurs. The swap mutation operator is implemented for inter chromosome representations. As the other operators, it randomly mutates based on the mutation rate; however, the mutation swaps two randomly selected alleles.

6.2.5 Dynamic Generation Gap

Generation gap is a parameter used in genetic algorithms to specify the amount of overlap among generations desired. In generational genetic algorithms, the offspring are saved in a separate pool until there are as many individuals as in the original pool.

Then the offspring's pool replaces the parent's pool for the next generation. The situation is quite different in the steady-state genetic algorithm where the offspring and parents occupy the same pool. Clearly, these two cases represent the two extremes of overlap between the generations.

In one of his papers, De Jong [10] concluded that generation gap has little importance in a genetic algorithm. He further concluded that the choice of selection and replacement strategies have the most profound effect. However, De Jong based his results on tests performed on genetic algorithms without crossover or mutation. Therefore, this is still an open question.

In the following chapter, we introduce the third heuristic technique used in this research—grouping genetic algorithms. We discuss the implementation issues and why and when the grouping genetic algorithms should be considered.

Chapter 7 Grouping Genetic Algorithms

An important class of computational problems are grouping problems, where the aim is to group together members of a set; namely, to find a proper partition of the set. The standard GAs fare poorly in this domain because of their inherent difficulty of capturing the grouping information pertinent to these problems. A new encoding scheme and genetic operators specially tailored for the grouping problems have been proposed by Emanuel Falkenauer [13], leading to the notion of *Grouping Genetic Algorithms* (GGA).

7.1 The Grouping Problems

Many a problem arising in practice consist in partitioning a set **U** of objects into a collection of mutually disjoint subsets $\mathbf{U}_i$ of **U** such that

$$\cup \mathbf{U}_i = \mathbf{U}$$

and

$$\mathbf{U}_i \cap \mathbf{U}_j = \Phi, \text{ if } i \neq j.$$

Such problems may also be viewed as those where the aim is to group the members of the set **U** into one or more groups of objects, with each object in exactly one group. In most of these problems, not all possible groupings are allowed: a feasible solution to the problem must comply with various hard constraints. That is, usually an object cannot be grouped with all possible subsets of the remaining objects.

The objective of the grouping is to optimize a fitness function defined over the set of all valid groupings. We consider the well-known *Bin Packing Problem* as an example. The bin packing problem (BPP) consists in placing n objects, each of which has a weight ($w_i > 0$), in the minimum number of bins, such that the total weight of the objects in each bin does not exceed the bin's capacity. All bins are assumed to have the same capacity ($C > 0$). In other words, the goal here is to partition n positive numbers (the weights) into the smallest possible number of subsets B_j (the bins), such that the sum of the integers in each subset does not exceed a positive number C. The BPP belongs to the class of NP-hard problems [27]. The hard constraint to be met in this problem is that the sum of sizes of objects in any group (bin) must not exceed the specified capacity, C, and the objective function to be optimized (minimized in this case) is the number of bins used. This is an example of a minimization grouping problem as we would like to use as few bins (groups) as possible.

The bin packing problem can serve as a good indication that the grouping problems are characterized by objective functions that depend on the composition of the groups as a whole instead of on the individuals in any particular group; that is, an object taken in isolation has little or no meaning to the feasibility of a solution.

7.2 Features of GGA

Developed to work with the grouping problems, the grouping genetic algorithm differs from a standard GA in two aspects: (1) a special encoding scheme is used to

incorporate relevant information of grouping problems, and (2) given the new encoding, special genetic operators are designed to suit the chromosomes.

7.2.1 The Encoding

Neither the standard nor the order-based genetic operators are appropriate for the grouping problems, the reason being that the structure of the simple chromosomes only deal with the objects and ignore group entities. As we have noted, the objective function of a grouping problem depends on the groups, but there is no structural counterpart in the chromosomes that is capable of keeping track of this piece of important information. To remedy this problem, the following encoding scheme has been devised: a simple (standard) chromosome is augmented with a *group part*, where the groups are encoded on a *one gene for one group* basis.

More concisely, we consider a chromosome for the bin packing problem. Assume a problem instance with 6 objects. Labeling them from 0 through 5, the object part of the chromosome is of the following form:

0	1	2	3	4	5		
A	D	B	C	A	B	:	. . .

meaning objects 0 and 4 are placed in the bin labeled A, 2 and 5 in bin B, 3 in C, and 1 in D. Up to this point, it is a straightforward encoding of chromosomes in simple GA. The group part of the chromosome contains the grouping information (bins in BPP):

. . . : A B C D.

By looking up the information in the object part, we can easily establish what the group names stand for. Specifically,

$$A = \{0, 4\}, B = \{2, 5\}, C = \{3\}, \text{ and } D = \{1\}.$$

The essence of GGA lies in that the genetic operators will now work with the group part of the chromosomes while the standard object part merely serves to identify the group each object belongs to. This also implies that the operators must be capable of dealing with chromosomes of variable length.

In short, the encoding scheme adopted here makes the genes represent the groups. The rationale is that, in grouping problems, it is the groups that are the meaningful building blocks, i.e., the smallest piece of a solution that conveys information on the expected quality of the solution. This is crucial for the very idea behind the GA paradigm is to perform an exploration of the search space so that the promising regions are identified, together with an exploitation of the information thus gathered, by an increased search effort in those regions. If, on the contrary, the encoding scheme does not allow the building blocks to be exploited and simultaneously serve as estimators of quality of the regions of the search space they occupy, then the GA strategy inevitably fails and the algorithm is little more than a random search.

7.2.2 Crossover

As pointed out in the previous section, a remarkable feature of the crossover operator in GGA is that it works with chromosomes of variable length where the genes represent the groups.

Due to the fact that the hard constraints and the objective function vary among different grouping problems, an approach to combine groups of one problem does not necessarily apply to another problem. Thus, the crossover strategy will not be the same for all problems. However, it will follow the general pattern outlined in the following:

1. Randomly select two crossing sites for each of the two parents. Note that we are working with the group part of the chromosomes, meaning we select a crossing section among the groups.
2. Insert the contents between the two crossing sites of the first parent to the second parent. The position to insert the information may be fixed, say, at the beginning of the crossing section, or may be randomly generated. This process inserts some of the groups from the first parent into the second.
3. Eliminate all objects that now occur twice from their "old" groups in the second parent. Semantically, this means the new membership forces the old to give way. As a consequence, some of the original groups in the second parent have to be altered, and they may not contain all the objects since some of them have to be eliminated.
4. If necessary, local problem-dependent heuristics can be performed according to the hard constraints and the objective function to be optimized.
5. Apply steps 2 through 4 to the two parents with their roles exchanged in order to generate the second offspring.

We now illustrate the above procedures using BPP as an example. A crossover's task consists of producing offspring out of two parents in such a way that the children inherit as much as possible of the meaningful information from both parents. Since it is the group (bin) that conveys important information in BPP, we must find a way to transmit bins from the parents to the children. We consider two chromosomes (with emphases on the group part):

0	1	2	3	4	5					
A	D	B	C	A	B	:	A	B	C	D

and

0	1	2	3	4	5				
b	c	a	c	b	c	:	a	b	c.

First, two crossing sites are chosen at random for each of the two parents, yielding, for instance,

A B | C D |

and

| a b | c.

Next, the bins between the crossing sites in the second chromosome are inserted into the first at the first crossing site, yielding

A B a b C D.

Now some of the objects appear twice in the solution and must be eliminated. The object part of the second chromosome reveals that objects labeled 0, 2, and 4 are to be eliminated from their original membership. Thus, we eliminate group A and B, leaving

a b C D.

With the elimination of the two bins, however, we have also eliminated object 5 entirely. To fix this problem, a local heuristic may be applied to reinsert the missing object. For instance, the First Fit Decreasing Algorithm first sorts the objects by their weights in decreasing order and then uses the First Fit Algorithm which places an object in the first bin that has enough space.

After the first child has been created, we reverse the roles of the two parent strings and repeat the above procedure in order to generate the second child, as instructed by step 5.

7.2.3 Mutation

Similar to the crossover operator, mutation for the grouping problems must also work with the group part of the chromosomes rather than the object part. The implementation details of the mutation operator depend on the particular problem to be optimized; however, two general strategies can be sketched here: (1) creating a new group, and (2) eliminating an existing group. For some problems, a third strategy can also be applied: shuffle a small number of randomly selected objects among their respective groups.

In the case of BPP, we may incorporate a simple mutation operator: given a chromosome, select at random a few bins (groups) and eliminate them; the objects that compose these bins are thus missing from the solution. Taking the chromosome from the previous section as an example, we have

0 1 2 3 4 5
A D B C A B : A B C D.

Suppose at random bins A and C are selected to be eliminated during this phase; namely, objects labeled 0, 3, and 4 are to be eliminated. The remaining group part becomes

C D.

We can now apply the First Fit algorithm to insert the missing objects back into the solution.

7.3 GGA Implementation of the TA Problem

The setup of the GGA implementation is similar to that of the simple GA. We use an augmented form of the simple chromosome as suggested in the previous section. In addition, modifications on some of the GA operators are made in order to work with the group part of the chromosomes. The fitness functions used for the two heuristics are identical and have been introduced in Section 5.3.

In our GGA implementation, a chromosome that represents the optimal solution to the problem instance given in Section 2.2 is of the following form:

2	3	1	2	2	2	3	1	3	1	:	A	3	8	10	B	1	4	5	6	C	2	7	9

Figure 7. Chromosome in GGA representing optimal solution to the example in Section 2.2.

where the first half, identical to the entire chromosome in GA, is the object part and the second half is the group part. In the object part, the value in the *i*th position indicates the concentrator to which the *i*th terminal is assigned. In the group part, we

gather all the terminals connected to one concentrator and specify this relationship by listing these terminals following the concentrator. For instance, terminals 3, 8, and 10 are assigned to concentrator A. In this example, concentrators are labeled with letters to be separated from the labels of terminals. However, in our actual implementation, since it is difficult to handle a data structure that permits both letters and integers, we still label the concentrators with integers. We make the distinction by using larger numbers for concentrators. For example, given a problem instance of 10 terminal sites labeled from 1 through 10, the concentrators will be labeled starting 11, thus different from all terminal labels.

In GGA, we use the same roulette wheel selection as we do in GA. Since the new chromosome architecture does not affect the fitness value of a string, there is no need to modify the existing selection strategy. The crossover operator, however, has been altered to work solely with the group part.

Crossover is accomplished by first selecting a crossing section for both parent chromosomes. The entire crossing section of the second parent is then injected into the first parent at the beginning of its crossing section. Since new groups are now included, we eliminate the equivalent groups that are originally in the first parent. Consider the group parts of two chromosomes.

1st : A 3 8 10 | B 1 4 5 6 | C 2 7 9
2nd : B 1 3 6 7 9 10 | C 2 5 8 A 4 |

Assume the delimiters (" | ") indicate the crossing sections. After we inject the crossing section of the second parent into the first parent, we have :

A 3 8 10 *C 2 5 8 A 4* B 1 4 5 6 C 2 7 9.

Now we have two groups labeled A and two labeled C. We resolve this redundancy by eliminating groups A and C from their old membership, yielding

C 2 5 8 A 4 B 1 4 5 6.

The string is now free of duplicate groups; however, it is likely that it still contains duplicate objects. In this case, terminals 4 and 5 are duplicate. We need to remove these objects from their old memberships as well, resulting in

C 2 5 8 A 4 B 1 6.

This leaves out terminals 3, 7, 9, and 10 from any group. At this stage we carry out the first-fit heuristic to re-assign these objects: visits the groups one at a time, assigns the object to the first group that is able to service, or to a randomly selected group if none is available.

After the first offspring has been generated, we reverse the roles of the two parents and start the construction of the second offspring. This implementation complies with the steps listed in Section 7.2.2.

The mutation operator is in essence quite different from a random alteration of values in the chromosomes as done in GA implementation. We adopt the strategy of probabilistically removing some groups from the chromosome and reassigning the missing objects. We consider the chromosome in Figure 7. Assume concentrator (group) B is removed due to probability. In doing so, we leave terminals (objects) 1, 4, 5, and 6 unassigned. Consequently, we need to perform a *first-fit heuristic* on these

objects. Namely, for each unassigned terminal, the algorithm looks into the concentrators one by one. The terminal is assigned to the first concentrator that is able to service. The strength of this mutation strategy lies in that, if before mutation a string is feasible, it is highly likely that it will still be feasible after mutation. This is due to the fact that, in the worst case when no other concentrators are available, all the terminals may again be assigned to the original concentrator(s) they belong to.

In the next chapter we present the problem instances we use in this research. These are randomly generated but highly constrained cases. We also discuss a similar work in the literature on the TA problem using the same test cases.

Chapter 8 Test Cases

The greedy algorithm, genetic algorithm, and grouping genetic algorithm are applied to ten different collections of 100 terminal sites and to one collection each of 200, 300, and 400 terminal sites [1]. In each case, the capacity of each concentrator is assumed to be 12, and the capacity requirement (weight) of each terminal is selected to be a random integer in the range 1 - 6. For each problem instance, the number of concentrators is determined in the following fashion: a quotient, Q, is computed by dividing the sum of capacity requirements of the terminals by the capacity of a single concentrator. The number of concentrators is then selected to be the floor of the number that is 7% in excess of Q. For example, the total weight requirement of 100 terminals in the first test case is 364. The computed quotient Q is then 30.33, 7% in excess of which is 32.46. Therefore, the resulting number of concentrator in this case is 32. Table 4 records the number of concentrators for each problem instance.

Number of Terminals	100									
Number of Concentrators	32	32	31	33	27	27	31	27	31	31

Number of Terminals	200	300	400
Number of Concentrators	63	96	128

Table 4. Number of concentrator sites used for each problem instance.

The locations for both the terminal sites and the concentrator sites are randomly generated on a 100 X 100 Euclidean grid. In each case, the cost of a connection between a terminal site and a concentrator site is taken to be the rounded Euclidean distance.

The thirteen problem instances specified in Table 3 have been proposed and experimented by Abuali et al. [1]. The derived conclusion in this particular work was that the resulting GA implementation of the TA problem outperformed the greedy algorithm in all of the test cases. In tightly constrained cases, the GA implementation was able to find good solutions while the greedy algorithm was often unable to find a feasible solution. Moreover, for the nine out of ten cases of 100 terminal sites where the greedy algorithm found a solution, the mean percentage improvement of the best genetic algorithm solution over the best greedy algorithm solution was 10.4%. The results for the data sets of 200, 300, and 400 terminals sites are similar and show significant improvement for the genetic algorithm over the greedy heuristic. The average improvement is 11.8%.

The results revealed in [1] are very encouraging and provide a good indication of the capability of the genetic algorithms when applied on the TA problem. However, in their work only the best costs are recorded; the actual assignments of the terminals (best assignment strings) are not available through either public or private channels. Moreover, there is no indication in their paper of the number of generations each run undergoes.

In the following chapter, we present and compare the results of executing our heuristic algorithms on these test cases.

Chapter 9 Experiments

The three heuristic algorithms are applied to the test cases. We present the results we have obtained followed by an analysis.

9.1 Results

Tables 5.1 and 5.2 summarize the results of executing the heuristic algorithms on all 13 test cases.

	Greedy	Genetic Algorithm		
	Algorithm	GENEsYs	LibGA	GGA
100_1	1203	1153	1138	1115
100_2	1253	1180	1159	1166
100_3	1274	1216	1181	1170
100_4	1438	1394	1344	1359
100_5	1600	1540	1500	1469
100_6	1446	1393	1373	1388
100_7	1961	1917	1838	1863
100_8	1865	1803	1702	1781
100_9	1564	1492	1425	1412
100_10	1367	1251	1216	1225

Figure 5.1. Best solutions of applying heuristics on test cases of 100 terminals each.

	Greedy Algorithm	Genetic Algorithm		
		GENEsYs	LibGA	GGA
200	2002	1939	1898	1919
300	2673	2607	2579	2595
400	3432	3327	3282	3316

Table 5.2. Best solutions of applying heuristics on test cases of 200, 300, and 400 terminals.

The solutions yielded by the GAs and GGA are based on seeding; i.e., the initial population is not randomly generated, but rather a set of feasible solutions given by the greedy algorithm.

9.2 Analysis

The results listed under the greedy algorithm are the best solutions yielded by the our implementation of greedy algorithm after 20,000 executions in each case. All experiments are independent of all others, and the order of assignments is entirely determined by the random number generator. Consequently, we cannot guarantee that the orders in the 20,000 runs are all distinct. In other words, there may be exact duplicates among the solutions. However, we believe the chance of this situation arising is slim considering the number of permutations of 100 integers is extremely large. We also believe the solution should definitely improve if more runs are performed to explore a wider range of the search space. We force the algorithm to stop

after 20,000 iterations in order to make a comparison with the genetic algorithms which also iterate for 20,000 generations.

The three columns under genetic algorithms are results given by our implementations of genetic algorithm using GENEsYs and LibGA, and of the grouping genetic algorithm. We have to pay attention to the issue that these results are obtained using *seeding*; i.e., the initial population is not entirely randomly generated but rather supplied by sources such as an input file containing feasible strings alone. Seeding is a widely used technique when approaching optimization problems provided there exists efficient means to confine the search space to feasible instead of random strings. This method is useful since it allows the GA to reach feasible regions very fast; otherwise, the GA may have to wander for a large number of generations in the search space before the feasible regions can be identified. Seeding is not a necessary ingredient in GA implementations. We incorporate this strategy in our research because, without seeding, our GAs fail to produce better solutions than does the greedy algorithm given the same condition.

Since our greedy algorithm runs very efficiently and yields feasible solutions consistently, we use the greedy algorithm to help identify the feasible regions for the GAs. We run the greedy algorithms for 20,000 iterations and record all the feasible solutions it encounters. From the set of feasible solutions we then randomly choose 500 strings to be used as the initial population for the GAs. This suggests that we use a population size of 500 for all our GA experiments throughout the study.

For each of the GA experiments, we use the same number of generations (20,000), the same population size (500) and the same crossover rate (0.6). Depending on the GA implementation, we adopt different mutation rates. For simple GAs using GENEsYs and LibGA, we abandon the commonly used strategy of taking the reciprocal of chromosome length as the mutation rate; e.g., for a chromosome of length 100, the mutation rate is 0.01. Instead, we take on a higher mutation rate for the experiments; e.g., 0.1 for the same problem instance. This is to cope with the highly constrained nature of the test cases. Namely, the feasible regions are rather sparse compared to the search space. We use higher mutation rate in the hope of discovering other feasible regions in the search space. As it turns out, the better results yielded by higher mutation rates appear to support our assumption.

On the other hand, for the GGA experiments, we use the reciprocal of the number of groups as the mutation rate; e.g., for a problem instance of 30 concentrators, the mutation rate is 0.033. This is not surprising since GGA works with groups.

As indicated in Tables 5.1 and 5.2, for a large majority of the problem instances (9 out of 13), the genetic algorithm implemented using LibGA consistently yields the best solution among the 4 heuristics. The greedy algorithm is the fastest algorithm, but it does not always produce near optimal solutions as can be derived from the tables: the improvement from the greedy algorithm to the GAs ranges between 3.5% and 11%. In all cases, GENEsYs does not perform as well as the other two GA implementations.

The results also seem to suggest that using GGA does not necessarily lead to better solutions even though the TA problem certainly possesses the properties of grouping problems. However, we would like to point out that, since the implementation of GGA is problem dependent in terms of the crossover and mutation operators, the strategies we adopt in this work are possibly not the most suitable for the TA problem. Other possible ways of implementing the GA operators may make a difference.

Chapter 10 Conclusion

This research has demonstrated the applicability of heuristic techniques to find approximate solutions for the terminal assignment problem. In this work, we considered a greedy algorithm, a genetic algorithm, and a grouping genetic algorithm. While the greedy algorithm is specially devised for the TA problem, genetic algorithms are general purpose evolutionary heuristics designed for a wide range of problems instead of any specific problem. However, the genetic algorithms have shown to outperform the greedy algorithm in all thirteen test cases used in this study. Even though our findings may not be conclusive, the results tend to suggest that the genetic algorithms work well with this particular problem. Moreover, it is very possible that different representations and genetic operators can lead to even better solutions.

References

[1] Abuali, Faris N., Schoenefeld, Dale A., and Wainwright, Roger L. (1994). Terminal Assignment in a Communications Network Using Genetic Algorithms. *Proceedings of the 22nd Annual ACM Computer Science Conference*, pp. 74 - 81. ACM Press, Arizona.

[2] Bäck, Thomas and Hoffmeister, Frank (1991). Extended Selection Mechanisms in Genetic Algorithms. *Proceedings of the Fourth International Conference on Genetic Algorithms and Their Applilcations*, pp. 92 - 99. University of California, San Diego, California.

[3] Bäck, Thomas (1992). GENEsYs 1.0. *Software distribution and installation notes*, Systems Analysis Research Group, LSXI, University of Dortmund, Dortmund, Germany.

[4] Baker, James E. (1985). Adaptive Selection Methods for Genetic Algorithms. *Proceedings of the First International Conference on Genetic Algorithm and Their Applications*, pp. 101 - 111. Hillsdale, New Jersey.

[5] Belew R. K. and Booker, L. B., editors (1991). *Proceedings of the 4th International Conference on Genetic Algorithms*. Morgan Kaufmann Publishers, California.

[6] Corcoran, Arthur L. and Wainwright, Roger L. (1993). LibGA: A User-Friendly Workbench for Order-Based Genetic Algorithm Research. *Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing*, pp. 111 - 117. ACM Press, New York.

[7] Cormen, Thomas H., Leiserson, Charles E., and Rivest, Ronald L. (1990). *Introduction to Algorithms*. The MIT Press, Massachusetts.

[8] De Jong, Kenneth A. (1975). An Analysis of the Behavior of a Class of Genetic Adaptive Systems. *Dissertation Abstracts International*. University of Michigan.

[9] De Jong, Kenneth A. (1980). Adaptive System Design. *IEEE Transactions on Systems, Man and Cybernetics*, pp. 566 - 574.

[10] De Jong, Kenneth, A. and Sarma, J. (1993). Generation Gaps Revisited. *Foundations of Genetic Algorithms 2*. Morgan Kaufman, California.

[11] Edmonds, Jack (1971). Matroids and the Greedy Algorithms. *Mathematical Porgramming*, pp. 126 - 136.

[12] Eshelman, Larry, J., editor (1995). *Proceedings of the 6th International Conference on Genetic Algorithms*. Morgan Kaufmann Publishers, California.

[13] Falkenauer, Emanuel (1994). A New Representation and Operators for Genetic Algorithms Applied to Grouping Problems. *Evolutionary Computation*, pp. 123 - 144. The MIT Press, Massachusetts.

[14] Forrest, S., editor (1993). *Proceedings of the 5th International Conference on Genetic Algorithms*. Morgan Kaufmann Publishers, California.

[15] Garey, Michael R. and Johnson, David, S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, California.

[16] Goldberg, David E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Massachusetts.

[17] Goldberg, David E. (1990). A Note on Boltzmann Tournament Selection for Genetic Algorithms and Population-Oriented Simulated Annealing. *Complex Systems*, pp 445 - 460.

[18] Grefenstette, J. J. (1984). GENESIS: A System for Using Genetic Search Procedures. *Proceedings of the Conference on Intelligent Systems and Machines*, pp. 161 - 165.

[19] Grenfenstette, J. J. (1986). Optimization of Control Parameters for Genetic Algorithms. *IEEE Transactions on Systems, Man and Cybernetics*, pp. 122 - 128.

[20] Holland, J. H. (1962). Outline for a Logical Theory of Adaptive Systems. *Journal of the Association for Computing Machinery*, pp. 297 - 314.

[21] Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor, Michigan.

[22] Kershenbaum, A. (1993). *Telecommunications Network Design Algorithms*. McGraw-Hill.

[23] Khuri, Sami, Bäck, Thomas, and Heitkötter, Jörg (1994). An Evolutionary Approach to Combinatorial Optimization Problems. *Proceedings of the 22nd ACM Computer Science Conference*, pp. 66 - 73. Phoenix, Arizona.

[24] Lawler, Eugene L. (1976). *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart, and Winston.

[25] Moret, B. M. E. and Shapiro, H. D. (1991). *Algorithms for P to NP. Volume I: Design and Efficiency*. The Benjamin/Cummings Publishing Company, Inc., California.

[26] Papadimitriou, Christos H. and Steiglitz, Kenneth (1982). *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Inc., New Jersey.

[27] Papadimitriou, Christos H. (1994). *Computational Complexity*. Addison-Wesley, Massachusetts.

[28] Rechenberg, Ingo (1973). *Evolutionsstrategie: Optimierung Technischer Systeme nach Prinzipien der Biologischen Evolution*. Frommann-Holzboog Verlag, Stuttgart.

[29] Richardson, J. T., Palmer, M. R., Liepins, G., and Hilliard, M. (1989). Some guidelines for Genetic Algorithms with Penalty Functions. *Proceedings of the 3rd International Conference on Genetic Algorithms and Their Applications*. Morgan Kaufmann Publishers, California.

[30] Schraudolph, N. N. and Grefenstette J. J. (1992). A User's Guide to GAucsd 1.4. *Technical Report*. University of California, San Diego.

[31] Schwefel, Hans-Paul (1981). *Numerical Optimization of Computer Models*. Wiley, Chichester.

[32] Starkweather, T., McDaniel, S., Mathias, K., Whitley, D., and Whitley, C. (1991). A Comparison of Genetic Sequencing Operators. *Proceedings of the Fourth International Conference on Genetic Algorithms*, pp. 69 - 76. San Mateo, California.

[33] Stinson, D. R. (1987). *An Introduction to the Design and Analysis of Algorithms*. The Charles Babbage Research Center, Manitoba, Canada. 2nd edition.

[34] Whitley, D. and Kauth, J. (1988). GENITOR: A Different Genetic Algorithm. *Proceedings of the Rocky Mountain Conference on Artificial Intelligence*, pp. 118 - 130. Denver, Colorado.

[35] Whitley, Darrell (1989). The GENITOR Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials Is Best. *Proceedings of the Third International Conference on Genetic Algorithms and Their Applications*, pp. 116 - 121. San Mateo, California.

Appendix

IMPLEMENTATION

OF

GREEDY ALGORITHM

```
/*****************************************************

     file : GREEDY.C

   author : Teresa L. Chiu   July 27, 1994

  purpose : Implementation of a greedy algorithm for
            Terminal Assignment Problem

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

/* include section */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <values.h>

/* constant */
#define TRUE  1
#define FALSE 0

/* typedef section */
struct node_type
{
  int               conc;
  int               dist;
  struct node_type *next;
};

typedef struct node_type *NODE;

struct cost_type
{
  NODE              head;
  NODE              tail;
  int               terminal;
  struct cost_type *next;
};

typedef struct cost_type *C_NODE;

struct c_list
{
  C_NODE head;
  C_NODE tail;
};

typedef struct c_list *COSTS;

/* function prototype */
void read_input (FILE *, int, int, int *, int **, int **);
void get_cost (int, int, int **, int **, COSTS *);
int connect (int, int, int, int *, COSTS, int **, long int *);
void print_table (FILE *, int, int **, long, int);
void free_list (COSTS *);
```

```
/* function implementation */
int main (int argc, char **argv)
{
  int    TerSize,            /* number of terminals */
         ConSize,            /* number of concentrators */
         Capacity,           /* capacity requirement of concentrators */
         j,
         count,              /* number of terminals unassigned */
         num = 0;            /* number of feasible solutions */
  long   i,
         runs,               /* number of runs - command line parameter */
         cost,               /* fitness value for one solution */
         best = MAXLONG;     /* overall best cost obtained */
  int    *TerWeight,         /* weight requirements of terminals */
         **TerCoord,         /* coordinates of terminals */
         **ConCoord,         /* coordinates of concentrators */
         **ConnectList;      /* actual assignment of terminals */
  time_t t;                  /* for random number generator */
  COSTS  CostList;           /* costs from terminals to concentrators */
  FILE   *fpw,               /* output file - command line parameter */
         *fpr;               /* input file - command line parameter */

  if (argc!=4)
  {
    printf ("\nUSAGE : greedy inputfile outputfile runs\n\n");
    exit (1);
  }

  if ((fpw = fopen (argv[2], "w")) == NULL)
  {
    printf ("\nCould not open file for writing.\n");
    exit (1);
  }

  fprintf (fpw, "\n****** %s ******\n ", argv[1]);

  /* convert command line argument */
  runs = atol (argv[3]);
  srand ((unsigned) time (&t));

  if ((fpr=fopen(argv[1], "r")) == NULL)
  {
    printf ("\nCould not open %s for reading.\n", argv[1]);
    exit (1);
  }

  /* obtain basic information of problem instance - number of terminals,
     number of concentrators, and concentrator capacity */
  fscanf (fpr, "%d %d %d", &ConSize, &TerSize, &Capacity);

  if ((TerWeight = (int *) calloc (TerSize, sizeof (int))) == NULL ||
      (TerCoord = (int **) calloc (TerSize, sizeof (int *))) == NULL ||
      (ConCoord = (int **) calloc (ConSize, sizeof (int *))) == NULL)
  {
    printf ("\nNot enough memory.\n");
    exit (1);
  }
```

```
for (i=0; i<TerSize; i++)
  if ((TerCoord[i] = (int *) calloc (2, sizeof (int))) == NULL)
  {
    printf ("\nNot enough memory.\n");
    exit (1);
  }

for (i=0; i<ConSize; i++)
  if ((ConCoord[i] = (int *) calloc (2, sizeof (int))) == NULL)
  {
    printf ("\nNot enough memory.\n");
    exit (1);
  }

/* obtain information of terminals and concentrators */
read_input (fpr, TerSize, ConSize, TerWeight, TerCoord, ConCoord);
/* compute costs of linking terminals to concentrators */
get_cost (TerSize, ConSize, TerCoord, ConCoord, &CostList);

if ((ConnectList = (int **) calloc (TerSize, sizeof (int *))) == NULL)
{
  printf ("\nNot enough memory.\n");
  exit (1);
}

for (j=0; j<TerSize; j++)
{
  if ((ConnectList[j] = (int *) calloc (2, sizeof (int))) == NULL )
  {
    printf ("\nNot enough memory.\n");
    exit (1);
  }
}

for (i=0; i<runs; i++)
{
  /* attempt to connect the terminals to concentrators */
  count = connect (TerSize, ConSize, Capacity, TerWeight, CostList,
                   ConnectList, &cost);

  /* test if all terminals have been assigned */
  if (count==0)
  {
    /* test if the cost if better than previous ones */
    if (cost<best)
    {
      print_table (fpw, TerSize, ConnectList, i, cost);
      best = cost;
    }

    /* another feasible solution */
    num++;
  }
}

fprintf (fpw, "\nTotal of %d feasible solution(s).\n", num);
```

```
  if (num>0)
    fprintf (fpw, "Best cost calculated for this run : %ld\n\n", best);

  /* clean up memory locations and close files*/
  for (j=0; j<TerSize; j++)
    free (ConnectList[j]);
  free (ConnectList);
  free (TerWeight);

  for (i=0; i<TerSize; i++)
    free (TerCoord[i]);
  free (TerCoord);

  for (i=0; i<ConSize; i++)
    free (ConCoord[i]);
  free (ConCoord);
  free_list (&CostList);

  fclose (fpr);
  fclose (fpw);

  return 0;
}

/* Reads information of terminals (weights and coordinates) and
   concentrators (coordinates).
*/
void read_input (FILE *fp, int TerSize, int ConSize, int *TerWeight,
                 int **TerCoord, int **ConCoord)
{
  int i, j;

  /* terminal coordinates and weights */
  for (i=0; i<TerSize; i++)
    fscanf (fp, "%d %d %d", &(TerCoord[i][0]), &(TerCoord[i][1]),
            &(TerWeight[i]));

  /* concentrator coordinates */
  for (i=0; i<ConSize; i++)
    fscanf (fp, "%d %d", &(ConCoord[i][0]), &(ConCoord[i][1]));
}

/* Computes individual costs of linking terminals to concentrators.
   Cost is taken to be the rounded Euclidean distance between two
   locations.
*/
void get_cost (int TerSize, int ConSize, int **TerCoord, int **ConCoord,
               COSTS *CostList)
{
  int    i, j, cost, hold1, hold2;
  NODE   ntemp, hold;
  C_NODE ctemp;

  if (((*CostList) = (COSTS) malloc (sizeof (struct c_list))) == NULL)
  {
    printf ("\nNot enough memory.\n");
    exit (1);
```

```
}

(*CostList)->head = NULL;
(*CostList)->tail = NULL;

/* store costs of linking terminal i to all concentrators */
for (i=0; i<TerSize; i++)
{
  ctemp = (C_NODE) malloc (sizeof (struct cost_type));
  if (ctemp==NULL)
  {
    printf ("\nNot enough memory.\n");
    exit (1);
  }

  ctemp->head = NULL;
  ctemp->tail = NULL;
  ctemp->terminal = i;
  ctemp->next = NULL;

  /* link to the end of cost_list */
  if ((*CostList)->head == NULL)
    (*CostList)->head = ctemp;
  else
    (*CostList)->tail->next = ctemp;
  (*CostList)->tail = ctemp;

  /* individual costs are being computed here */
  for (j=0; j<ConSize; j++)
  {
    hold1 = TerCoord[i][0] - ConCoord[j][0];
    hold2 = TerCoord[i][1] - ConCoord[j][1];
    cost = sqrt (hold1 * hold1 + hold2 * hold2);

    ntemp = (NODE) malloc (sizeof (struct node_type));
    if (ntemp==NULL)
    {
      printf ("\nNot enough memory.\n");
      exit (1);
    }

    ntemp->conc = j;
    ntemp->dist = cost;
    ntemp->next = NULL;

    /* link to ctemp in increasing order of cost */
    if (ctemp->head==NULL)
      ctemp->head = ctemp->tail = ntemp;
    else
    {
      /* start sorting */
      hold = ctemp->head;
      if (hold->dist>ntemp->dist)
      {
        ntemp->next = hold;
        ctemp->head = ntemp;
      }
```

```
        else
        {
          while (hold->next!=NULL && hold->next->dist<ntemp->dist)
            hold = hold->next;
          if (hold->next==NULL)
          {
            ctemp->tail->next = ntemp;
            ctemp->tail = ntemp;
          }
          else
          {
            ntemp->next = hold->next;
            hold->next = ntemp;
          }
        }
      }
    }
  }
}

/* Connects terminals to concentrators in a random order. A terminal
   is connected to its closest concentrator possible. Returns the number
   of terminals that cannot be assigned.
*/
int connect (int TerSize, int ConSize, int Capacity, int *TerWeight,
             COSTS CostList, int **ConnectList, long *cost)
{
  int     i, term_no, num,
          count = TerSize,
          done = FALSE,
          flag;
  int    *ConWeight;          /* capacity taken up in concentrators */
  C_NODE  ctemp, hold;
  NODE    ntemp;

  if ((ConWeight = (int *) calloc (ConSize, sizeof (int))) == NULL)
  {
    printf ("\nNot enough memory.\n");
    exit (1);
  }

  for (i=0; i<ConSize; i++)
    ConWeight[i] = 0;

  *cost = 0;

  /* assign a randomly chosen terminal */
  do
  {
    num = ConSize;
    term_no = rand () % count;

    /* go to the appropriate terminal */
    ctemp = CostList->head;
    for (i=0; i<term_no; i++)
    {
      hold = ctemp;
```

```
      ctemp = ctemp->next;
    }

    /* number of terminal that is now to be assigned */
    term_no = ctemp->terminal;

    flag = FALSE;
    ntemp = ctemp->head;

    /* find the closest concentrator that still has room */
    do
    {
      /* connect if concentrator not overloaded */
      if (ConWeight[ntemp->conc]+TerWeight[term_no]<=Capacity)
      {
        ConWeight[ntemp->conc] += TerWeight[term_no];
        ConnectList[term_no][0] = ntemp->conc;
        ConnectList[term_no][1] = ntemp->dist;
        *cost += ntemp->dist;
        flag = TRUE;
      }
      else
      {
        /* try the next closest concentrator */
        num--;
        ntemp = ntemp->next;
      }
    } while (num>0 && !flag);

    if (num==0)
      /* some terminal cannot be assigned at all */
      done = TRUE;
    else
    {
      /* move the terminal to the end of list */
      CostList->tail->next = ctemp;
      CostList->tail = ctemp;
      if (ctemp==CostList->head)
        CostList->head = ctemp->next;
      else
        hold->next = ctemp->next;
      ctemp->next = NULL;
      count--;
    }
  }
  while (!done && count > 0);

  free (ConWeight);
  return count;
}

/* Prints out the assignment of terminals and the total cost if the
   solution is feasible.
*/
void print_table (FILE *fp, int TerSize, int **ConnectList, long num,
                  int cost)
{
```

```
  int i;

  fprintf (fp, "\nTotal cost of run %ld : %d\n", ++num, cost);
  fprintf (fp, "\nThe assignment : \n");
  for (i=0; i<TerSize; i++)
    fprintf (fp, "%d ", ++ConnectList[i][0]);
  fprintf (fp, "\n");
  fprintf (fp, "\n=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=\n");
}

/* Frees up memory location of CostList.
*/
void free_list (COSTS *CostList)
{
  C_NODE ctemp;
  NODE   ntemp;

  while ((*CostList)->head!=NULL)
  {
    ctemp = (*CostList)->head;
    while (ctemp->head!=NULL)
    {
      ntemp = ctemp->head;
      ctemp->head = ctemp->head->next;
      free (ntemp);
    }
    (*CostList)->head = ctemp->next;
    free (ctemp);
  }
  free (*CostList);
}

/*************************** End of File ****************************/
```

IMPLEMENTATION

OF

GENETIC ALGORITHM

(GENEsYs)

```
/*****************************************************************/
/*                                                               */
/*  Copyright (c) 1996                                           */
/*  Teresa L. Chiu                                               */
/*                                                               */
/*  Department of Math & Computer Science                        */
/*  San Jose State University                                    */
/*  San Jose, CA 95192                                           */
/*                                                               */
/*  e-mail: chiu@mathcs.sjsu.edu                                 */
/*                                                               */
/*  Permission is hereby granted to copy all or any part of      */
/*  this program for free distribution.   The author's name      */
/*  and this copyright notice must be included in any copy.      */
/*                                                               */
/*****************************************************************/

/*
 *    file:   f_37.c
 *
 *    author: Teresa L. Chiu, March 25, 1996
 *
 *    Terminal Assignment Problem
 *
 */

/* include section */
#include "../define.h"
#include "../extern.h"

/* constant */
#define MAX_DIST 141          /* maximum distance on 100 X 100 grid */
#define TRUE     1
#define FALSE    0

extern FUNCTION  f_tab[];

double
f_37 (x, Length, FsbFlg)
  register int   x[];
  register int   Length;
  register int  *FsbFlg;
{
  static int  Flg = TRUE,
              TerSize,        /* number of terminals */
              ConSize,        /* number of concentrators */
              Capacity,       /* capacity requirement of concentrators */
              bit_no;         /* bits required for each terminal */

  static int  *TerWeight,     /* weight requirements of terminals */
              *ConWeight,     /* capacity taken up in concentrators */
              **TerCoord,     /* coordinates of terminals */
              **ConCoord,     /* coordinates of concentrators */
              **Costs,        /* costs from terminals to concentrators */
              *y;             /* integer representation of strings */

  double      term1, term2, log(), sqrt(), pow();
```

```
int         i, j, hold,
            CostSum = 0,    /* cost of an assignment */
            Violation = 0, /* number of overloaded concentrators */
            Penalty = 0;    /* penalty for infeasible strings */

FILE        *fp;            /* information of problem intance */

if (Flg)
{
  /* retrieve input data */
  if ((fp = fopen (f_tab[F_nbr].DatFilNam, "r")) == NULL)
  {
    printf ("%s/f_37 : Couldn't open datafile (%s)\n",  _GA,
            f_tab[F_nbr].DatFilNam);
    exit (2);
  }

  /* number of concentrators */
  readval (fp, "%d", &ConSize);
  if (ConSize < 1)
  {
    printf ("%s/f_37 : Can't connect terminals to %d concentrators\n",
            _GA, ConSize);
    exit (2);
  }

  /* calculate number of bits in vector x needed for one terminal */
  bit_no = ceil (log ((double) ConSize) / log (2.0));

  /* number of terminals */
  readval (fp, "%d", &TerSize);
  if (TerSize * bit_no != Length)
  {
    printf ("%s/f_37 : Wrong data dimension (%d, %d, %d)\n",
            _GA, TerSize, Length, bit_no);
    exit (2);
  }

  /* concentrator capacity */
  readval (fp, "%d", &Capacity);
  if (Capacity < 1)
  {
    printf ("%s/f_37 : Unacceptable concentrator capacity (%d)\n",
            _GA, Capacity);
    exit (2);
  }

  if ((TerWeight = (int *) calloc (TerSize, sizeof (int))) == NULL ||
      (TerCoord = (int **) calloc (TerSize, sizeof (int *))) == NULL ||
      (ConCoord = (int **) calloc (ConSize, sizeof (int *))) == NULL ||
      (Costs = (int **) calloc (TerSize, sizeof (int *))) == NULL ||
      (ConWeight = (int *) calloc (ConSize, sizeof (int))) == NULL ||
      (y = (int *) calloc (TerSize, sizeof (int))) == NULL)
  {
    printf ("%s/f_37 : Calloc failed\n", _GA);
    exit (2);
  }
```

```
  for (i=0; i<TerSize; i++)
    if ((TerCoord[i] = (int *) calloc (2, sizeof (int))) == NULL ||
        (Costs[i] = (int *) calloc (ConSize, sizeof (int))) == NULL)
    {
      printf ("%s/f_37 : Calloc failed\n", _GA);
      exit (2);
    }

  for (i=0; i<ConSize; i++)
    if ((ConCoord[i] = (int *) calloc ( 2, sizeof (int))) == NULL)
    {
      printf ("%s/f_37 : Calloc failed\n", _GA);
      exit (2);
    }

  /* terminal coordinates and weights */
  for (i=0; i<TerSize; i++)
  {
    readval (fp, "%d", &(TerCoord[i][0]));
    readval (fp, "%d", &(TerCoord[i][1]));
    readval (fp, "%d", &(TerWeight[i]));
  }

  /* concentrator coordinates */
  for (i=0; i<ConSize; i++)
  {
    readval (fp, "%d", &(ConCoord[i][0]));
    readval (fp, "%d", &(ConCoord[i][1]));
  }

  /* compute the costs of linking terminals and concentrators */
  for (i=0; i<TerSize; i++)
    for (j=0; j<ConSize; j++)
    {
      term1 = (double) (TerCoord[i][0] - ConCoord[j][0]);
      term2 = (double) (TerCoord[i][1] - ConCoord[j][1]);
      Costs[i][j] = (int) sqrt (term1 * term1 + term2 * term2);
    }

  fclose (fp);
  Flg = FALSE;
}

/* convert binary vector x into equivalent integer vector y */
for (i=0; i<TerSize; i++)
{
  y[i] = 0;
  for (j=0; j<bit_no; j++)
    y[i] += x[i*bit_no+j] * (int)pow(2, bit_no-j-1);
  if (y[i] >= ConSize)
  {
    y[i] = rand () % ConSize;
    hold = y[i];
    for (j=0; j<bit_no; j++)
    {
      x[i*bit_no+j] = hold/(int)pow(2, bit_no-j-1);
      hold -= x[i*bit_no+j];
```

```
      }
    }
  }

  *FsbFlg = TRUE;
  for (i=0; i<ConSize; i++)
    ConWeight[i] = 0;
  for (i=0; i<TerSize; i++)
    ConWeight[y[i]] += TerWeight[i];
  for (i=0; i<ConSize; i++)
    /* any overloaded concentrator makes the assignment infeasible */
    if (ConWeight[i] > Capacity)
    {
      *FsbFlg = FALSE;
      Violation++;
      Penalty += (ConWeight[i] - Capacity);
    }

  /* cost of assignment */
  for (i=0, CostSum=0; i<TerSize; i++)
    CostSum += Costs[i][y[i]];

  if (!*FsbFlg)
    return (CostSum + PenConst * TerSize + Penalty * Violation);
  else
    return CostSum;
}

int
f_37R (x, n)
  register double x[];
  register int    n;
{
  return 0;
}

/*************************** End of File ***************************/
```

```
/*****************************************************************/
/*                                                               */
/*  Copyright (c) 1992                                           */
/*  Thomas Baeck                                                 */
/*  Computer Science Department, LSXI                            */
/*  University of Dortmund                                       */
/*  Baroper Str. 301                                             */
/*  P.O. Box 50 05 00                                            */
/*  D-4600 Dortmund 50                                           */
/*  Germany                                                      */
/*                                                               */
/*  e-mail: baeck@home.informatik.uni-dortmund.de                */
/*          baeck@ls11.informatik.uni-dortmund.de                */
/*                                                               */
/*  Permission is hereby granted to copy all or any part of      */
/*  this program for free distribution.   The author's name      */
/*  and this copyright notice must be included in any copy.      */
/*                                                               */
/*****************************************************************/

/*
 *      file:     define.h
 *
 *    author:     Thomas Baeck
 *
 *   created:     October 7th, 1992
 *
 *   purpose:     Type and constant definitions for the
 *                Genetic Algorithm software.
 *
 *  modified:     January 13th, 1992
 *
 *  Added a constraint component to the function table
 *  entries. Rst returns 1, if a constant is violated,
 *  0 else.
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

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

#define INTSIZE         32        /* bits per unsigned int */

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

#define   C_LBRACE      '{'               /* left  curly brace */
#define   S_LBRACE      "{"
#define   C_RBRACE      '}'               /* right curly brace */
#define   S_RBRACE      "}"
#define   S_WHITESPACE  " \t\r\n\v\f"     /* white space characters */
#define   S_STRDEL      "\'\""            /* list of string delimiters */
#define   C_COMMENT     '#'               /* comment character */
#define   S_COMMENT     "#"
```

```
/*
 * Several constants for parameterization etc.
 */

#define _PTHSEP         "/"             /* filename separator */

#define _GA             "GENEsYs 2.11"
#define _SIZ            128         /* message sizes */
#define _LARGE_VAL      1.0e+39     /* large floating point value */
#define _EPS            1.0e-2      /* small value */

#define _FEW            (ga->ga_LdaVal / 20)      /* a few bits */

#define GA_GRY_COD      'G'         /* gray code */
#define GA_STD_COD      'B'         /* standard binary code */

#define GA_STD_MTT      'S'         /* standard mutation */
#define GA_ADT_MTT      'A'         /* adaptive mutation rates, AMIM */
#define GA_ADX_MTT      'X'         /* adaptive mutation rates, AMEM */

#define GA_STD_SEL      'P'         /* proportional selection */
#define GA_BZM_SEL      'B'         /* Boltzmann selection */
#define GA_BRK_SEL      'R'         /* Baker's ranking method */
#define GA_IRK_SEL      'I'         /* inverse Baker's ranking */
#define GA_WRK_SEL      'W'         /* Whitley's ranking method */
#define GA_MLR_SEL      'M'         /* (m,l)-selection, randomized */
#define GA_MLC_SEL      'C'         /* (m,l)-selection as in ES */
#define GA_TRN_SEL      'T'         /* tournament selection */

#define GA_STD_REC      'S'         /* standard x-point crossover */
#define GA_UFM_REC      'U'         /* uniform crossover */
#define GA_DCT_REC      'D'         /* discrete recombination as in ES */
#define GA_IMD_REC      'I'         /* intermediate recombination (ES) */
#define GA_RID_REC      'R'         /* random intermediate recombination */
#define GA_NOP_REC      '_'         /* no recombination */

/*
 * Default values. These are chosen such that by default
 * a standard GA (according to GENESIS) results.
 */

#define D_INBIAS   (double)0.5    /* default bias for RNG */
#define D_FCTNBR   0              /* default objective function index */
#define D_FCTDIM   30             /* default objective function dimens. */
#define D_CODE     GA_GRY_COD     /* default code is Gray */

#define D_TOTEXP   1              /* default total number of exp. */
#define D_TOTTRL   1000           /* default total number of trials */

#define D_MUEVAL   50             /* default value of mue (popsize) */
#define D_LDAVAL   50             /* default value of lambda */
#define D_RHOVAL   1              /* default left extinctive bound */

#define D_CRSRAT   0.6            /* default crossover rate */
#define D_CRSPNT   2              /* default number of crossover points */
#define D_RECSCM   "S_"           /* default recombination mechanism */
```

```
#define D_MTTSCM  GA_STD_MTT      /* default mutation mechanism */
#define D_MTTNXT  0L              /* default initial mut. position */
#define D_MTTRAT  0.001           /* default mutation rate */
#define D_SIGNBR  0               /* default number of adaptive rates */
#define D_MTTBIT  0               /* default number of mutation bits */

#define D_SLTSCM  GA_STD_SEL      /* default selection mechanism */
#define D_WDWSIZ  5               /* default window size for scaling */
#define D_ETAVAL  1.1             /* default expected value for ranking */
#define D_TRNSIZ  2               /* default tournament size */
#define D_TMPCTL  0.99            /* default temperature (Boltzmann) */
#define D_THSHLD  0.5             /* default threshold (Boltzmann) */
#define D_ACTTMP  10.0            /* default initial temp (Boltzmann) */
#define D_CHNLGT  5               /* default chain length (Boltzmann) */

#define D_GENGAP  1.0             /* default generation gap */

#define D_ORGSED  123456789       /* default seed for RNG */

#define D_GFXIVL  10              /* default graphics update interval */

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

/*
 * The general parameter structure of the GA.
 */

typedef struct {                  /* an GA_Dsc structure */

  unsigned long ga_TotTrl,        /* total number of trials */
                ga_OrgSed;        /* original seed for RNG */

  int           ga_AllFlg,        /* evaluate all structures */
                ga_DonFlg,        /* termination flag */
                ga_DbgFlg,        /* debugging flag */
                ga_QryFlg,        /* directory name query flag */
                ga_GfxFlg,        /* flag for online-visualization */
                ga_ObjFlg,        /* output flag object variables */
                ga_MttFlg,        /* output flag mutation rates */
                ga_BitFlg;        /* output flag bitmap dumps */

  int           ga_FctNbr,        /* objective function index */
                ga_FctDim,        /* objective function dimension */
                ga_TotExp,        /* total number of experiments */
                ga_GenCnt,        /* generation counter */
                ga_TrlCnt,        /* actual trial counter */
                ga_GfxIvl,        /* graphics update interval */
                ga_DtaIvl;        /* data collection interval */

  char          ga_CodScm;        /* encoding mechanism */

  int           ga_ObjLen,        /* length per object variable */
                ga_ObjTot;        /* total genotype length */

  int           ga_MueVal,        /* number of parents */
                ga_LdaVal,        /* number of offspring */
                ga_RhoVal;        /* left extinctive boundary */
```

```
char            ga_RecScm[3];     /* recombination mechanism */
int             ga_CrsPnt;        /* number of crossover points */
double          ga_CrsRat;        /* crossover rate */

char            ga_MttScm;        /* mutation mechanism */
unsigned long   ga_MttNxt;        /* next position for mutation */
int             ga_SigNbr,        /* number of mutation rates */
                ga_MttLen,        /* number of bits per mutation rate */
                ga_MttTot;        /* total mutation rate bits */
double          ga_MttRat;        /* mutation rate */

char            ga_SltScm;        /* selection mechanism */
int             ga_WdwSiz,        /* scaling window size */
                ga_ChnLgt,        /* markov chain length */
                ga_TrnSiz,        /* tournament selection size */
                ga_EltFlg;        /* elitist selection flag */
double          ga_EtaVal,        /* maximum expected value */
                ga_TmpCtl,        /* temperature control value */
                ga_ThsHld,        /* threshold for Boltzmann selection */
                ga_ActTmp,        /* actual temperature Boltzmann */
                ga_Worst,         /* worst within WdwSiz generations */
                *ga_WstHst;       /* history of last worst values */

double          ga_GenGap;        /* generation gap */

double          ga_InBias;        /* bias for population initialization */

char            ga_Suffix[_SIZ],    /* file name suffix */
                ga_OutFil[_SIZ],    /* output file name */
                ga_RepFil[_SIZ],    /* report file name */
                ga_VarFil[_SIZ],    /* variance file name */
                ga_LogFil[_SIZ],    /* logging file name */
                ga_ObjFil[_SIZ],    /* file for object variables */
                ga_MttFil[_SIZ],    /* file for mutation rates */
                ga_BitFil[_SIZ],    /* bitmap dump file */
                ga_FmtStr[_SIZ];    /* format string */

double          ga_AllBst,        /* all time best performance */
                ga_CurBst,        /* current best performance */
                ga_CurAvg,        /* current average performance */
                ga_CurWst,        /* current worst performance */
                ga_BstAvg,        /* average of mue best individuals */
                ga_MttMin,        /* minimum mutation rate */
                ga_MttAvg,        /* average mutation rate */
                ga_MttMax;        /* maximum mutation rate */

double          ga_BiasVl;        /* bias value */
int             ga_LostBt,        /* number of lost bits */
                ga_ConvBt;        /* number of converged bits */

double          ga_Online,        /* online measure */
                ga_Ofline;        /* offline measure */

int             *ga_ObjBit,       /* output specification object vars. */
                *ga_MttBit,       /* output specification mut. rates */
                *ga_IndBit;       /* output specification individuals */
```

```
} GA_Dsc;

/*
 * An individual.
 */

typedef struct {  /* an GA_Ind structure */

  int           *ind_ObjBit,     /* object variable genotype */
                *ind_MttBit;     /* mutation rate genotype */

  double        *ind_ObjVar,     /* object variable vector */
                *ind_MttRat,     /* standard deviation vector */
                ind_SltPrb,      /* selection probability */
                ind_FctVal,      /* objective function value */
                ind_FitVal;      /* fitness value */

  int           ind_EvlFlg,      /* evaluation flag */
                ind_FsbFlg;      /* feasibility flag */
} GA_Ind;

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

typedef struct {                 /* a scoring tuple for selection */

  int           ind_Score,       /* scoring value */
                ind_Index;       /* index of individual */

} GA_Score;

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

/*
 * The RNG stems from Grefenstette's original GENESIS program.
 * Note, that it needs a global variable
 * 'unsigned int Seed', which is updated by the RNG.
 */

#define MASK          ~(~0<<(INTSIZE-1))
#define PRIME         65539
#define SCALE         0.4656612875e-9

#define Rand()        ((Seed = ((Seed * PRIME) & MASK)) * SCALE)
/*
#define Rand()        ( drand48() )
*/

#define Randint(Lo,Hi)   ((int) ((Lo) + ((Hi)-(Lo)+1) * Rand()))

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

typedef struct {   /* a statistical measures structure for performing
                      the Kolmogoroff-Smirnov test */

  double        ks_BstVal,       /* best value after a run */
                ks_RelFrq,       /* relative frequency of that value */
                ks_EstPrb,       /* estimated probability */
```

```
                ks_NorPrb,      /* probability from normal distrib. */
                ks_Delta1,      /* delate to previous */
                ks_Delta2;      /* actual delta */

  int           ks_BstCnt;      /* frequency counter */

} KS_Entry;

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

/*
 * Function table primitives go here; from GENEsYs 1.0
 */

#define MOPT          3         /* maximum option number per function */

#define VRBL          0         /* variable dimension */
#define STRC          1         /* strict dimension setting */
#define EXCP          2         /* an exception */

#define REAL          0         /* real vectors */
#define PERM          1         /* permutations */
#define BINY          2         /* pseudoboolean function */

#define DUMMY         "dummy"   /* dummy string */

/*
 * The objective function structure from GENEsYs 1.0
 */

typedef struct {
  int     dim;                  /* dimension of the function */
  int     DimObl;               /* dimension obligate flag */
  int     MrkFct;               /* function characteristic marker */
  double  umin,                 /* lower and upper bounds for */
          umax;                 /* each object variable */
  double  (*f)();               /* pointer to the function */
  int     (*Rst)();             /* pointer to constraint function */
  char   *fnm;                  /* filename of the function */
  char   *descr;                /* textual description */
  char    DatFilNam[_SIZ];      /* data file name */
  double  CstVal[MOPT];         /* constant external parameters */
  char   *CstDsc[MOPT];         /* constant description */
  struct {
          int     OVal;         /* integer option value */
          char   *ODsc[MOPT];   /* option description */
          char   *OTtl;         /* option title */
  } FctOpt[MOPT];               /* integer options and descriptions */
} FUNCTION;

/*** end of file ***/
```

```
/*****************************************************************/
/*                                                               */
/*  Copyright (c) 1992                                           */
/*  Thomas Baeck                                                 */
/*  Computer Science Department, LSXI                            */
/*  University of Dortmund                                       */
/*  Baroper Str. 301                                             */
/*  P.O. Box 50 05 00                                            */
/*  D-4600 Dortmund 50                                           */
/*  Germany                                                      */
/*                                                               */
/*  e-mail: baeck@home.informatik.uni-dortmund.de                */
/*          baeck@ls11.informatik.uni-dortmund.de                */
/*                                                               */
/*  Permission is hereby granted to copy all or any part of      */
/*  this program for free distribution.   The author's name      */
/*  and this copyright notice must be included in any copy.      */
/*                                                               */
/*****************************************************************/

/*
 *      file:     extern.h
 *
 *    author:     Thomas Baeck
 *
 *   created:     July 20th, 1992
 *
 *   purpose:     Compatibility file for function table,
 *                extern declarations.
 *
 *  modified:
 *
 */

extern  FUNCTION     f_tab[];     /* function table */
extern  unsigned int Seed;        /* Seed for random number generator */
extern  int          Gen;         /* -> eps_GenCnt */
extern  int          FctNbr;      /* -> eps_FctNbr + 1 */
extern  int          F_nbr;       /* -> eps_FctNbr */

/*** end of file ***/
```

IMPLEMENTATION

OF

GENETIC ALGORITHM

(LibGA)

```
/************************************************************

     file : TA.C

   author : Teresa L. Chiu   October 18, 1995

  purpose : Implementation of Genetic Algorithm for Terminal
            Assignment Problem

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

/* include section */
#include "ga.h"
#include "getopt.h"

/* constant */
#define TERM 141            /* maximum distance on 100 X 100 grid */

long TerSize,               /* number of terminals */
     ConSize,               /* number of concentrators */
     Capacity,              /* capacity requirement of concentrators */
     Violation;             /* number of overloaded concentrators */
long *TerWeight,            /* weight requirements of terminals */
     *ConWeight,            /* capacity taken up in concentrators */
     **TerCoord,            /* coordinates of terminals */
     **ConCoord,            /* coordinates of concentrators */
     **Costs;               /* costs from terminals to concentrators */

/* function prototype */
long obj_fun ();
void getdata (FILE *);
void getcost ();
int  feasible (Chrom_Ptr);
long cost (Chrom_Ptr);
long penalty (Chrom_Ptr);
void cleanup (FILE *);

/* function implementation */
int main (int argc, char **argv)
{
  int         c;            /* extract user option */
  FILE        *fp;          /* information of problem instance */
  GA_Info_Ptr ga_info;      /* pointer to pertinent GA information */
  char        filename[BUFSIZ] = "ta.cfg";
                            /* configuration file for GA */

  /* extract user option */
  while ((c = getopt (argc, argv, "f:h")) != EOF)
  {
    switch (c)
    {
      case 'f':             /* get input file name */
        sscanf (optarg, "%s", filename);
        break;
      case 'h':             /* usage information */
        usage ();
        exit (1);
```

```
        break;
      }
    }

    /* initialize the genetic algorithm */
    ga_info = GA_config ("ta.cfg", obj_fun);

    if ((fp = fopen (filename, "r")) == NULL)
    {
      perror (filename);
      exit (1);
    }

    /* initialize problem data */
    getdata (fp);
    /* compute costs of connecting terminals to concentrators */
    getcost ();

    /* redefine some settings */
    ga_info->chrom_len = TerSize;
    ga_info->rand_minint = 1;
    ga_info->rand_maxint = ConSize;

    /* run the GA */
    GA_run (ga_info);

    /* reset the GA */
    GA_reset (ga_info, "ta.cfg");

    cleanup (fp);
    return 0;
}

/* Outputs usage information of program execution. This program takes
   1 command line argument, the input file.
*/
int usage ()
{
    fprintf (stderr, "\nusage : ta [-f input-file] [-h]\n\n");
    return 0;
}

/* User specified objective function. If string is feasible, return
   the total cost of connections; if not, penalty is calculated.
*/
long obj_fun (Chrom_Ptr chrom)
{
    if (feasible (chrom))
      chrom->fitness = cost (chrom);
    else
      chrom->fitness = cost (chrom) + penalty (chrom);

    return 0;
}

/* Reads problem instance from input file. Information includes number
   of concentrators, number of terminals, capacity requirement of
```

```
   concentrators, terminal coordinates, terminal weights, and
   concentrator coordinates.
*/
void getdata (FILE *fp)
{
  long i;

  /* number of concentrators and number of terminals */
  fscanf (fp, "%d", &ConSize);
  fscanf (fp, "%d", &TerSize);

  if (ConSize<1)
  {
    printf ("can't connect terminals to %d concentrators\n", ConSize);
    exit (2);
  }

  /* concentrator capacity */
  fscanf (fp, "%d", &Capacity);

  if (Capacity<1)
  {
    printf ("unacceptable concentrator weight...\n");
    exit (2);
  }

  if ((TerWeight = (long*) calloc (TerSize, sizeof (long))) == NULL ||
      (TerCoord = (long**) calloc (TerSize, sizeof (long*))) == NULL ||
      (ConCoord = (long**) calloc (ConSize, sizeof (long*))) == NULL ||
      (Costs = (long**) calloc (TerSize, sizeof (long*))) == NULL ||
      (ConWeight = (long*) calloc (ConSize, sizeof (long))) == NULL)
  {
    printf ("calloc failed...\n");
    exit (2);
  }

  for (i=0; i<TerSize; i++)
  {
    if ((TerCoord[i] = (long*) calloc (2, sizeof (long))) == NULL ||
        (Costs[i] = (long*) calloc (ConSize, sizeof (long)))== NULL)
    {
      printf ("calloc failed...\n");
      exit (2);
    }
  }

  for (i=0; i<ConSize; i++)
  {
    if ((ConCoord[i] = (long*) calloc (2, sizeof (long))) == NULL)
    {
      printf ("calloc failed...\n");
      exit (2);
    }
  }

  /* terminal coordinates and weights */
  for (i=0; i<TerSize; i++)
```

```
    fscanf (fp, "%d %d %d", &(TerCoord[i][0]), &(TerCoord[i][1]),
            &(TerWeight[i]));

  /* concentrator coordinates */
  for (i=0; i<ConSize; i++)
    fscanf (fp, "%d %d", &(ConCoord[i][0]), &(ConCoord[i][1]));
}

/* Computes the costs of assigning terminals to concentrators. The cost
   is taken to be the rounded Euclidean distance between two locations.
*/
void getcost ()
{
  long   i, j;
  double term1, term2;       /* components in computing distance */

  for (i=0; i<TerSize; i++)
    /* distance between two locations on grid */
    for (j=0; j<ConSize; j++)
    {
      term1 = (double) (TerCoord[i][0] - ConCoord[j][0]);
      term2 = (double) (TerCoord[i][1] - ConCoord[j][1]);
      Costs[i][j] = (long) sqrt (term1 * term1 + term2 * term2);
    }
}

/* Returns 1 if chromosome is feasible, and 0 otherwise. If any
   concentrator is overloaded, the string is infeasible.
*/
int feasible (Chrom_Ptr chrom)
{
  long i, flag = TRUE;

  for (i=0; i<ConSize; i++)
    ConWeight[i] = 0;

  for (i=0; i<chrom->length; i++)
    ConWeight[(int)(chrom->gene[i])-1] += TerWeight[i];

  Violation = 0;

  for (i=0; i<ConSize; i++)
    /* any overloaded concentrator makes the assignment infeasible */
    if (ConWeight[i]>Capacity)
    {
      flag = FALSE;
      Violation++;
    }

  return flag;
}

/* Computes the total cost of a given assignment. This excludes the
   penalty term for infeasible strings.
*/
long cost (Chrom_Ptr chrom)
{
```

```
  long i,
       CostSum = 0;           /* sum of costs */

  for (i=0; i<TerSize; i++)
    CostSum += Costs[i][(int)(chrom->gene[i])-1];

  return CostSum;
}

/* Compute the penalty term of given chromosome. If string is feasible,
   penalty is 0. First term of the penalty function forces the best
   infeasible string worse than the worst feasible string, while the
   second term differenciates between infeasible strings.
*/
long penalty (Chrom_Ptr chrom)
{
  long i,
       sum = 0;               /* total excess load of concentrators */

  for (i=0; i<ConSize; i++)
    sum += MAX (ConWeight[i] - Capacity, 0);

  sum *= Violation;           /* graded penalty */
  sum += TerSize * TERM;      /* penalty term */

  return sum;
}

/* Frees up memory locations and closes input file.
*/
void cleanup (FILE *fp)
{
  long i;

  free (TerWeight);

  for (i=0; i<TerSize; i++)
    free (TerCoord[i]);
  free (TerCoord);

  for (i=0; i<ConSize; i++)
    free (ConCoord[i]);
  free (ConCoord);

  free (ConWeight);
  free (Costs);

  fclose (fp);
}

/*************************** End of File ***************************/
```

```
/*==========================================================================
| (c) Copyright Arthur L. Corcoran, 1992, 1993.  All rights reserved.

| Genetic Algorithm Definitions - GA.H

| Goldberg's Terminology
| ----------------------
| Chromosome = string
| Gene       = feature, character, detector
| Allele     = feature value
| Locus      = string position
| Genotype   = structure
| Phenotype  = parameter set, alternative solution, a decoded structure
| Epistasis  = nonlinearity
==========================================================================*/

/*--------------------------------------------------------------------------
| Header files
--------------------------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <ctype.h>

#if defined(__BORLANDC__)
#include <process.h>
#include <alloc.h>
#elif !defined(__STDC__)
#include <malloc.h>
#endif

/*--------------------------------------------------------------------------
| Constants
 --------------------------------------------------------------------------*/
#define VERSION   "1.03j"
#define COPYRIGHT "(c) Copyright Arthur L. Corcoran, 1992, 1993.
                   All rights reserved.\n\
                   Modifications (c) Copyright joke@Germany.EU.net, 1995.
                   All rights reserved."
#define FALSE         0
#define TRUE          !(FALSE)
#define OK            0
#define ERROR         !(OK)
#define UNSPECIFIED  -1

/*--- Data type --- */
#define DT_BIT        0          /* Bit string */
#define DT_INT        1          /* Integers */
#define DT_INT_PERM   2          /* Integer Permutation */
#define DT_REAL       3          /* Reals */

/*--- Method to generate initial pool --- */
#define IP_NONE         0x00
#define IP_INTERACTIVE  0x01
#define IP_FROM_FILE    0x02
#define IP_RANDOM       0x04
```

```
/*--- Type of output report --- */
#define RP_NONE    0
#define RP_MINIMAL 1
#define RP_SHORT   2
#define RP_LONG    3

/*--- Magic cookies for validation ---*/
#define NL_cookie 0x00000000     /* NULL cookie */
#define CF_cookie 0x11111111     /* ga_info (config) cookie */
#define PL_cookie 0x22222222     /* pool cookie */
#define CH_cookie 0x33333333     /* chrom cookie */

/*-------------------------------------------------------------------
| Type definitions
-------------------------------------------------------------------*/
/*--- A function pointer ---*/
typedef int (*FN_Ptr) ();

/*--- A function table ---*/
typedef struct
{
  char *name;                    /* Function name */
  FN_Ptr fun;                    /* Function pointer */
}
FN_Table_Type, *FN_Table_Ptr;

/*--- A Gene (or allele) is a bit, int, float, etc. ---*/
typedef double Gene_Type, *Gene_Ptr;

/*--- A Chromosome ---*/
typedef struct
{
  long magic_cookie;             /* For validation */
  Gene_Ptr gene;                 /* Encoding */
  int length;                    /* Length of gene */
  double fitness;                /* Fitness value of chromosome */
  float ptf;                     /* Percent of total fitness */
  int index;                     /* My index */
  int idx_min, idx_max;          /* Reserved */
  int parent_1, parent_2;        /* Indices of parents */
  int xp1, xp2;                  /* Crossover points */
}
Chrom_Type, *Chrom_Ptr;

/*--- A Pool ---*/
typedef struct
{
  long magic_cookie;              /* For validation */
  Chrom_Ptr *chrom;               /* Chromosomes */
  int size, max_size;             /* Number of chromosomes */
  double total_fitness;           /* Total fitness of pool */
  double min, max, ave, var, dev; /* Current pool fitness stats */
  int min_index, max_index;       /* Index of min/max chromosomes */
  int best_index;                 /* Index of best chromosome */
  int minimize;                   /* Minimize pool [y/n]? */
  int sorted;                     /* Is pool sorted [y/n]? */
}
```

```
Pool_Type, *Pool_Ptr;

/*--- GA configuration info ---*/
typedef struct
{
/*--- Basic info ---*/
  long magic_cookie;            /* For validation */
  char user_data[80];           /* User data file (unused) */
  int rand_seed;                /* Seed for random number generator */
  int rand_minint;              /* Minimal random integer in initial pool
                                   (-joke) */
  int rand_maxint;              /* Maximal random integer in initial pool
                                   (-joke) */
  int datatype;                 /* Data type flag */
  int ip_flag;                  /* Initial pool generation method flag */
  char ip_data[80];             /* Data file name (IP_FROM_FILE) */
  int pool_size;                /* Pool size (IP_RANDOM) */
  int chrom_len;                /* Chromosome size (IP_RANDOM) */
  int iter, max_iter;           /* Number of iterations for ga */
  int minimize;                 /* Minimize EV_fun? */
  int elitist;                  /* Use elitism? */
  int converged;                /* Has ga converged? */
  int use_convergence;          /* Use convergence? */
  float bias;                   /* Selection bias */
  float gap;                    /* Generation gap */
  float x_rate;                 /* Crossover rate */
  float mu_rate;                /* Mutation rate */
  float scale_factor;           /* Scale for fitness <= 0 */

  /*--- Functions ---*/
  FN_Ptr GA_fun;                /* GA */
  FN_Ptr SE_fun;                /* Selection */
  FN_Ptr X_fun;                 /* Crossover */
  FN_Ptr MU_fun;                /* Mutation */
  FN_Ptr EV_fun;                /* Evaluation */
  FN_Ptr RE_fun;                /* Replacement */

  /*--- Reports ---*/
  int rp_type;                  /* Type of output report */
  int rp_interval;              /* Output report interval */
  FILE *rp_fid;                 /* Output report fid */
  char rp_file[80];             /* Output report file name */

  /*--- Pools ---*/
  Pool_Ptr old_pool, new_pool;

  /*--- Stats ---*/
  Chrom_Ptr best;               /* Best chromosome */
  int num_mut, tot_mut;         /* Mutation statistics */
}
GA_Info_Type, *GA_Info_Ptr;

/*-----------------------------------------------------------------------
| Pseudo-functions
-----------------------------------------------------------------------*/
#ifdef GENESIS_RNG
#define INTSIZE                 32
```

```
#define MASK                    ~(~0<<(INTSIZE-1))
#define PRIME                   65539
#define SCALE                   0.4656612875e-9
#define Rand()                  ((Seed = ((Seed*PRIME) & MASK))*SCALE)
#define Randint(low,high)       ((int) ((low) + ((high)-(low)+1) * Rand()))
#define SEED_RAND(seed)         Seed = (unsigned long)seed
#define RAND_FRAC()             Rand()
#define RAND_DOM(lo,hi)         Randint(lo,hi)
#define RAND_BIT()              Randint(0,1)
extern unsigned long Seed;
#else

/*--- random number in [0..1] ---*/
#if defined(__BORLANDC__)
#define SEED_RAND(seed) (srand((seed)))
#define RAND_FRAC() ((double)rand()/RAND_MAX)
#else
#define SEED_RAND(seed) (srandom((seed)))
#define RAND_FRAC() ((double)random()*(1.0/2147483647.0))
#endif

/*--- random number in domain [lo..hi] ---*/
#define RAND_DOM(lo,hi) ((int)
                         floor(RAND_FRAC()*(((hi)-(lo))+0.999999))+(lo))

/*--- random bit ---*/
#define RAND_BIT()  ((RAND_FRAC()>=.5)? 1 : 0 )
#endif

/*--- min and max ---*/
#define MIN(a,b) ((a < b) ? (a) : (b))
#define MAX(a,b) ((a > b) ? (a) : (b))

#define UT_warn(message) {fprintf(stderr,"WARNING: %s\n", message);}
#define UT_error(message) {fprintf(stderr,"ERROR: %s\n", message);
                           exit(1);}
#define UT_iswap(a, b) {int tmp; tmp = *(a); *(a) = *(b); *(b) = tmp;}

/*---------------------------------------------------------------------
| Function prototypes
---------------------------------------------------------------------*/
extern char *GA_name (), *SE_name (), *X_name (), *MU_name (),
            *RE_name (); extern char *FN_name ();
extern Chrom_Ptr SE_fun (), CH_alloc ();
extern Pool_Ptr PL_alloc ();
extern GA_Info_Ptr GA_config (), CF_alloc ();

/*************************** End of File ***************************/
```

```
/* Declarations for getopt : GETOPT.H
   Copyright (C) 1989, 1990, 1991, 1992 Free Software Foundation, Inc.

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2, or (at your option)
   any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
*/

#ifndef _GETOPT_H_
#define _GETOPT_H_

/* For communication from 'getopt' to the caller.
   When 'getopt' finds an option that takes an argument,
   the argument value is returned here.
   Also, when 'ordering' is RETURN_IN_ORDER,
   each non-option ARGV-element is returned here.
*/

extern char *optarg;

/* Index in ARGV of the next element to be scanned.
   This is used for communication to and from the caller
   and for communication between successive calls to 'getopt'.

   On entry to 'getopt', zero means this is the first call; initialize.

   When 'getopt' returns EOF, this is the index of the first of the
   non-option elements that the caller should itself scan.

   Otherwise, 'optind' communicates from one call to the next
   how much of ARGV has been scanned so far.
*/

extern int optind;

/* Callers store zero here to inhibit the error message 'getopt' prints
   for unrecognized options.
*/

extern int opterr;

/* Describe the long-named options requested by the application.
   The LONG_OPTIONS argument to getopt_long or getopt_long_only is a
   vector of 'struct option' terminated by an element containing a name
   which is zero.

   The field 'has_arg' is:
```

```
   no_argument        (or 0) if the option does not take an argument,
   required_argument  (or 1) if the option requires an argument,
   optional_argument  (or 2) if the option takes an optional argument.

   If the field 'flag' is not NULL, it points to a variable that is set
   to the value given in the field 'val' when the option is found, but
   left unchanged if the option is not found.

   To have a long-named option do something other than set an 'int' to
   a compiled-in constant, such as set a value from 'optarg', set the
   option's 'flag' field to zero and its 'val' field to a nonzero
   value (the equivalent single-letter option character, if there is
   one).  For long options that have a zero 'flag' field, 'getopt'
   returns the contents of the 'val' field.
*/

struct option
{

#ifdef __STDC__
  const char *name;
#else
  char *name;
#endif

  /* has_arg can't be an enum because some compilers complain about
     type mismatches in all the code that assumes it is an int.
  */
  int has_arg;
  int *flag;
  int val;
};

/* Names for the values of the 'has_arg' field of 'struct option'. */

enum _argtype
{
  no_argument,
  required_argument,
  optional_argument
};

#ifdef __STDC__
extern int getopt (int argc, char *const *argv, const char *shortopts);
extern int getopt_long (int argc, char *const *argv,
                        const char *shortopts,
                        const struct option *longopts, int *longind);
extern int getopt_long_only (int argc, char *const *argv,
                             const char *shortopts,
                             const struct option *longopts, int *longind);

/* Internal only.  Users should not call this directly. */
extern int _getopt_internal (int argc, char *const *argv,
                             const char *shortopts,
                             const struct option *longopts, int *longind,
                             int long_only);
```

```
#else /* not __STDC__ */
extern int getopt ();
extern int getopt_long ();
extern int getopt_long_only ();

extern int _getopt_internal ();
#endif /* not __STDC__ */

#endif /* _GETOPT_H_ */

/************************** End of File ***************************/
```

```
#=====================================================================
# (c) Copyright Teresa L. Chiu, 1995. All rights reserved.
#
# "Terminal Assignment Problem" configuration file
#=====================================================================
#
#---------------------------------------------------------------------
# User data file
# This information is not used by the GA, however, it is a convenient
# way to input a data file name or other information to your application.
#---------------------------------------------------------------------
# user_data datafile

#---------------------------------------------------------------------
# Seed for random number generator
#
# Usage: rand_seed my_pid
#        rand_seed number
#
#    my_pid = use system pid as random seed
#    number = seed for random number generator, a positive integer
#
# DEFAULT: rand_seed 1
#---------------------------------------------------------------------
rand_seed my_pid
# rand_seed 1

#---------------------------------------------------------------------
# The data type of the allele
#
# Usage: datatype [bit | int | int_perm | real]
#
#    bit      = bit string
#    int      = integers
#    int_perm = permutation of integers
#    real     = real numbers
#
# DEFAULT: int_perm
#---------------------------------------------------------------------
# datatype bit
datatype int
# datatype int_perm
# datatype real

#---------------------------------------------------------------------
# How to initialize the pool
#
# Usage: initpool [random | from_file filename | interactive]
#
#    random      = generate at random based on
#                      datatype, chrom_len, & pool_size
#    from_file   = read from a file
#       filename = the name of the file to read from
#    interactive = read from stdin
#
# DEFAULT: initpool random
```

```
#-----------------------------------------------------------------------
# initpool random
initpool from_file   initpool.dat
# initpool interactive

#-----------------------------------------------------------------------
# Chromosome length, needed when "initpool random" selected
#
# Usage: chrom_len length
#
#    length = chromosome length, a positive integer
#
# DEFAULT: chrom_len 10
#-----------------------------------------------------------------------
# chrom_len 25

#-----------------------------------------------------------------------
# Pool size, needed when "initpool random" selected
#
# Usage: pool_size size
#
#    size = pool size, a positive integer
#
# DEFAULT: 100
#-----------------------------------------------------------------------
pool_size 500

#-----------------------------------------------------------------------
# When to stop the GA
#
#    Convergence means when the variance = 0, or equivalently, when
#    all the fitness values in the pool are identical.
#
#    Iterations means the number of generations for the generational
#    model and the number of trials for the steady state model. Numbers
#    must be given as positive integers.  It takes roughly pool_size/2
#    iterations of the steady state model to equal one iteration of
#    the generational model.
#
# Usage: stop_after convergence
#        stop_after number [use_convergence | ignore_convergence]
#
#    convergence            - stop when the GA converges
#    number                 - stop after specified number of iterations
#       use_convergence     - will stop early if GA converges (default)
#       ignore_convergence - WILL NOT stop early even if GA converges
#
# DEFAULT: stop_after convergence
#-----------------------------------------------------------------------
# stop_after convergence
# stop_after 500
stop_after 20000 use_convergence
# stop_after 1000 ignore_convergence

#-----------------------------------------------------------------------
# GA Type:
#
```

```
# Usage: ga [generational | steady_state]
#
#    generational = generational GA
#    steady_state = steady-state GA
#
# WARNING: This directive has the following side effects:
#
#         GA type          Directives set as a side effect
#       ------------       -------------------------------
#       generational       selection        roulette
#                          replacement      append
#                          rp_interval      1
#
#       steady-state       selection        rank_biased
#                          replacement      by_rank
#                          rp_interval      100
#
# DEFAULT: ga generational
#-----------------------------------------------------------------------
# ga generational               # most commonly used
# ga steady_state               # used by Genitor

#-----------------------------------------------------------------------
# Generation gap:
#
#    The generation gap  represents a percentage of the population to
#    copy (clone) to the new pool at each generation.  This only makes
#    sense in a GA with two pools as in the generational model.  A gap
#    of 0.0 is the traditional generational algorithm.  As the gap
#    increases, it becomes more like a steady-state algorithm.  A gap
#    of 1.0 essentially disables crossover since only reproduction
#    occurs.
#
# Usage: gap number
#
#    number = generation gap, valid range = [0.0 .. 1.0]
#
# DEFAULT: gap 0.0
#-----------------------------------------------------------------------
# gap 0.3

#-----------------------------------------------------------------------
# Selection method:
#
# Usage: selection [roulette | rank_biased | uniform_random]
#
#    roulette       = Roulette wheel
#    rank_biased    = Ranked, biased selection as in Genitor
#    uniform_random = Pick one at random
#
# DEFAULT: selection roulette
#-----------------------------------------------------------------------
# selection roulette            # use with generational GA
# selection rank_biased         # use with steady-state GA
# selection uniform_random      # experimental

#-----------------------------------------------------------------------
```

```
# Selection bias
#
# Usage: bias number
#
#    number = selection bias, valid range = [1.0 .. 2.0]
#             Only used for rank_biased selection
#
# DEFAULT: bias 1.8
#-----------------------------------------------------------------------
# bias 1.1

#-----------------------------------------------------------------------
# Crossover method:
#
# Usage: crossover [simple | uniform | order1 | order2 | position |
#                   cycle | pmx | uox | rox | asexual]
#
#    simple     = children get alternate "halves" of parents
#    uniform    = alleles swapped uniformly
#    order1     = order based
#    order2     = order based
#    position   = order based
#    cycle      = order based
#    pmx        = order based
#    uox        = uniform order
#    rox        = relative order
#    asexual    = swap two alleles
#
# DEFAULT: crossover order1
#-----------------------------------------------------------------------
# crossover simple
crossover uniform
# crossover order1            # use ony with integer permutations
# crossover order2            # use ony with integer permutations
# crossover position          # use ony with integer permutations
# crossover cycle             # use ony with integer permutations
# crossover pmx               # use ony with integer permutations
# crossover uox               # use ony with integer permutations
# crossover rox               # use ony with integer permutations
# crossover asexual           # use ony with integer permutations

#-----------------------------------------------------------------------
# Crossover Rate
#
# Usage: x_rate number
#
#    number = crossover rate (percentage), valid range = [0.0 .. 1.0]
#             A crossover rate of 0.0 disables crossover
#
# DEFAULT: x_rate 1.0
#-----------------------------------------------------------------------
x_rate 0.6

#-----------------------------------------------------------------------
# Mutation method:
#
# Usage: mutation [simple_invert | simple_random | swap]
```

```
#
#    simple_invert = invert a bit
#    simple_random = random bit value
#    swap          = swap two alleles
#
# DEFAULT: mutation swap
#-----------------------------------------------------------------------
# mutation simple_invert      # use only with bits
# mutation simple_random      # use only with bits
# mutation swap               # use with any datatype

#-----------------------------------------------------------------------
# Mutation Rate
#
# Usage: mu_rate number
#
#    number = mutation rate (percentage), valid range = [0.0 .. 1.0]
#             A mutation rate of 0.0 disables mutation
#
# DEFAULT: mu_rate 0.0
#-----------------------------------------------------------------------
mu_rate 0.1

#-----------------------------------------------------------------------
# Replacement method:
#
# Usage: replacement [append | by_rank | first_weaker | weakest]
#
#    append       = append to new pool, as in generational GA
#    by_rank      = insert in sorted order, as in Genitor
#    first_weaker = replace first weaker found in linear scan of pool
#    weakest      = replace weakest member of the pool
#
# DEFAULT: replacement append
#-----------------------------------------------------------------------
# replacement append           # use with roulette (generational GA)
# replacement by_rank          # use with rank_biased (steady-state GA)
# replacement first_weaker     # experimental
# replacement weakest          # experimental

#-----------------------------------------------------------------------
# Objective of GA:
#
# Usage: objective [minimize | maximize]
#
#    minimize = minimize evaluation function
#    maximize = maximize evaluation function
#
# DEFAULT: objective minimize
#-----------------------------------------------------------------------
# objective minimize
# objective maximize

#-----------------------------------------------------------------------
# Elitism
#
#    Elitism has two actions.  For a generational GA, elitism makes two
```

```
#    copies of the best performer in the old pool and and places them in
#    the new pool, thus ensuring the most fit chromosome survives.  The
#    other action works with both models.  In this case, elitism picks
#    the best two chromosomes from the parents and children.  Thus, if a
#    child is not as fit as either parent, it will not be placed in the
#    new pool.  Selecting elitism in LibGA performs both actions.
#
# Usage: elitism [true | false]
#
#    true  = ensure best members survive until next generation
#    false = no guarantee best will survive
#
# DEFAULT: elitism true
#-------------------------------------------------------------------------
# elitism true
# elitism false

#-------------------------------------------------------------------------
# Report type
#
# Usage: rp_type [none | minimal | short | long]
#
#    none     = output nothing
#    minimal  = output configuration and final result
#    short    = output minimal + statistics only
#    long     = output short + dump pool
#
# DEFAULT: rp_type short
#-------------------------------------------------------------------------
# rp_type none
# rp_type minimal
# rp_type short
# rp_type long

#-------------------------------------------------------------------------
# Report interval
#
# Usage: rp_interval number
#
#    number = interval between reports, a positive integer
#
# DEFAULT: rp_interval 1
#-------------------------------------------------------------------------
# rp_interval 10

#-------------------------------------------------------------------------
# Output report filename
#
# Usage: rp_file file_name [file_mode]
#
#    file_name = name of report file
#    file_mode = optional file mode for fopen()
#       a      = append (DEFAULT)
#       w      = overwrite
#
# DEFAULT: (write to stdout)
#-------------------------------------------------------------------------
```

```
# rp_file ga.out
# rp_file ga.out a
# rp_file ga.out w

/*************************** End of File ****************************/
```

IMPLEMENTATION

OF

GROUPING GENETIC ALGORITHM

```
/*************************************************************

     File : GGA.C

   Author : Teresa L. Chiu   January 5, 1996

  Purpose : Implementation of the Grouping Genetic Algorithm,
            specifically designed for the Terminal Assignment
            Problem.

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

/* include section */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <values.h>
#include <time.h>

/* constant */
#define MAX_DIST 141     /* maximum distance on 100 X 100 grid */
#define TRUE     1
#define FALSE    0

/* variable declaration */
long   PoolSize,         /* pool size - command line parameter */
       Generation,       /* generation - command line parameter */
       TerSize,          /* number of terminals */
       ConSize,          /* number of concentrators */
       Capacity;         /* capacity requirement of concentrators */
long   **TerCoord,       /* coordinates of terminals */
       **ConCoord,       /* coordinates of concentrators */
       *TerWeight,       /* weight requirements of terminals */
       *ConWeight,       /* capacity taken up in concentrators */
       **Cost;           /* costs from terminals to concentrators */
long   **Pool,           /* population pool */
       **MatePool;       /* mating pool for genetic operators */
long   **List,           /* terminals assigned to each concentrator */
       *Mix,             /* temporary array for crossover */
       *Conc,            /* temporary array for crossover */
       *Dup,             /* temporary array for mutation */
       *Best;            /* overall best assignment obtained */
long   term,             /* first term in penalty function (fixed) */
       size,             /* length of each string in pool */
       len,              /* length of array Mix */
       min = MAXINT;     /* overall minimum cost of assignment */
int    Feasible;         /* feasibility of an individual string */
float  X_Over,           /* crossover rate - command line parameter */
       Mutation;         /* mutation rate - command line parameter */
double FitSum,           /* total sum of fitness values (for selection) */
       prob;             /* random probability (for crossover) */
FILE   *info,            /* information of problem instance */
       *init;            /* initial population strings */

/* function prototype */
void usage ();
void getdata ();
```

```
void calc_cost ();
void init_pool ();
long fitness (long *);
long penalty (long *);
void run ();
long roulette ();
void swap (long *, long *);
void crossover (long, long, long, long, long, long);
void heuristic (long, long, long, long);
void first_fit (long, long);
void mutate (long);
void stat (long);
void copy_back ();
void print_best ();
void cleanup ();

/* function implementation */
int main (int argc, char **argv)
{
  long   i;
  time_t t;             /* for random number generator */

  if (argc != 7)
  {
    usage ();
    exit (1);
  }

  if ((info = fopen (argv[1], "r")) == NULL ||
      (init = fopen (argv[6], "r")) == NULL)
  {
    printf ("\nCan't open file for read...\n");
    exit (2);
  }

  srand((unsigned) time(&t));

  /* save the command line parameters */
  printf ("\n");
  for (i=0; i<7; i++)
    printf ("%s ", argv[i]);

  /* convert the command line parameters */
  PoolSize = atoi (argv[2]);
  Generation = atoi (argv[3]);
  X_Over = atof (argv[4]);
  Mutation = atof (argv[5]);

  /* store information of problem instance */
  getdata ();
  /* compute costs of connecting terminals to concentrators */
  calc_cost ();
  /* initialize the population by seeding */
  init_pool ();

  for (i=0; i<Generation; i++)
  {
```

```
    /* perform genetic operations */
    run ();
    /* gather statistics of current generation */
    stat (i+1);
    /* replace the population pool by the mating pool - generational GA */
    copy_back ();
  }

  /* output best assignment after a specified number of generations */
  print_best ();
  cleanup ();

  return 0;
}

/* Outputs usage information of program execution.
   This program takes a total of 7 command line arguments.
*/
void usage ()
{
  printf ("\nusage : gga inputfile poolsize generation crossover ");
  printf ("mutation initpool\n\n");
}

/* Reads problem instance from input file. Information includes number
   of concentrators, number of terminals, capacity requirement of
   concentrators, terminal coordinates, terminal weights, and
   concentrator coordinates.
*/
void getdata ()
{
  long i;

  /* numbers of concentrators and terminals, and concentrator capacity */
  fscanf (info, "%ld", &ConSize);
  fscanf (info, "%ld", &TerSize);
  fscanf (info, "%ld", &Capacity);

  /* first term in penalty function */
  term = MAX_DIST * TerSize;
  /* length of each string in pool */
  size = (2 * TerSize) + ConSize + 2;
  /* length of array Mix */
  len = 2 * (TerSize + ConSize);

  if ((TerWeight = (long *) calloc (TerSize, sizeof (long))) == NULL ||
      (ConWeight = (long *) calloc (ConSize, sizeof (long))) == NULL ||
      (TerCoord = (long **) calloc (TerSize, sizeof (long *))) == NULL ||
      (ConCoord = (long **) calloc (ConSize, sizeof (long *))) == NULL ||
      (List = (long **) calloc (ConSize, sizeof (long *))) == NULL ||
      (Mix = (long *) calloc (len, sizeof (long))) == NULL ||
      (Dup = (long *) calloc (TerSize, sizeof (long))) == NULL ||
      (Best = (long *) calloc (size, sizeof (long))) == NULL ||
      (Conc = (long *) calloc (ConSize, sizeof (long))) == NULL)
  {
     printf ("\nCalloc failed...\n");
     exit (3);
```

```
  }

  for (i=0; i<TerSize; i++)
    if ((TerCoord[i] = (long *) calloc (2, sizeof (long))) == NULL)
    {
      printf ("\nCalloc failed...\n");
      exit (3);
    }

  for (i=0; i<ConSize; i++)
    if ((ConCoord[i] = (long *) calloc (2, sizeof (long))) == NULL ||
        (List[i] = (long *) calloc (TerSize+1, sizeof (long))) == NULL)
    {
      printf ("\nCalloc failed...\n");
      exit (3);
    }

  /* terminal coordinates and weights */
  for (i=0; i<TerSize; i++)
    fscanf (info, "%ld%ld%ld", &(TerCoord[i][0]), &(TerCoord[i][1]),
                               &TerWeight[i]);

  /* concentrator coordinates */
  for (i=0; i<ConSize; i++)
    fscanf (info, "%ld%ld", &(ConCoord[i][0]), &(ConCoord[i][1]));
}

/* Computes the costs of assigning terminals to concentrators. The cost
   is taken to be the rounded Euclidean distance between two locations.
*/
void calc_cost ()
{
  long   i, j;
  double term1, term2;  /* components in computation of distance */

  if ((Cost = (long **) calloc (TerSize, sizeof (long *))) == NULL)
  {
    printf ("\nCalloc failed...\n");
    exit (3);
  }

  for (i=0; i<TerSize; i++)
  {
    if ((Cost[i] = (long *) calloc (ConSize, sizeof (long))) == NULL)
    {
      printf ("\nCalloc failed...\n");
      exit (3);
    }

    /* distance between two locations on grid */
    for (j=0; j<ConSize; j++)
    {
      term1 = (double) (TerCoord[i][0] - ConCoord[j][0]);
      term2 = (double) (TerCoord[i][1] - ConCoord[j][1]);
      Cost[i][j] = (long) (sqrt(term1*term1+term2*term2));
    }
  }
```

```
}

/* Creates the initial population.  Seeding is used in this program;
   i.e., a specified number of strings are being read from input file.
*/
void init_pool ()
{
  long i, j, k, pos, index;

  if ((Pool = (long **) calloc (PoolSize, sizeof (long *))) == NULL ||
      (MatePool = (long **) calloc (PoolSize, sizeof (long *))) == NULL)
  {
    printf ("\nCalloc failed...\n");
    exit (3);
  }

  FitSum = 0.0;

  for (i=0; i<PoolSize; i++)
  {
    if ((Pool[i] = (long *) calloc (size, sizeof (long))) == NULL ||
        (MatePool[i] = (long *) calloc (size, sizeof (long))) == NULL)
    {
      printf ("\nCalloc failed...\n");
      exit (3);
    }

    for (j=0; j<ConSize; j++)
      List[j][0] = 0;

    /* store the strings in population pool as well as in the array
       List according to the concentrator each terminal belongs to */
    for (j=0; j<TerSize; j++)
    {
      fscanf (init, "%ld", &(Pool[i][j]));
      List[Pool[i][j]][++List[Pool[i][j]][0]] = j;
    }

    pos = TerSize;

    /* copy the sorted information inList back to population pool */
    for (j=0; j<ConSize; j++)
    {
      Pool[i][pos++] = j + TerSize;
      for (k=0; k<List[j][0]; k++)
        Pool[i][pos++] = List[j][k+1];
    }

    /* compute the fitness value of string */
    Pool[i][pos] = fitness (Pool[i]);

    /* save the best assignment so far encountered */
    if (Pool[i][pos] < min)
    {
      min = Pool[i][pos];
      index = i;
    }
```

```
    FitSum += 1.0 / Pool[i][pos++];
    Pool[i][pos] = Feasible;
  }

  for (i=0; i<size; i++)
    Best[i] = Pool[index][i];

  printf ("\n\nINITIAL GENERATION : BEST FITNESS VALUE = %ld\n", min);
}

/* Computes the fitness value of the given string. Fitness value is the
   total sum of assignment costs, plus penalty term if infeasible.
*/
long fitness (long *chrom)
{
  long i,
       sum = 0;          /* sum of costs */

  for (i=0; i<TerSize; i++)
    sum += Cost[i][chrom[i]];

  return (sum + penalty (chrom));
}

/* Computes the penalty term of the given string. If string is feasible,
   penalty is 0. First term of the penalty function forces the best
   infeasible string to be worse than the worst feasible string, while
   the second term differenciates between infeasible strings.
*/
long penalty (long *chrom)
{
  long i,
       sum = 0,          /* total excess load of concentrators */
       violation = 0;    /* number of overloaded concentrators */

  for (i=0; i<ConSize; i++)
    ConWeight[i] = 0;

  /* capacity taken up in each concentrator */
  for (i=0; i<TerSize; i++)
    ConWeight[chrom[i]] += TerWeight[i];

  Feasible = TRUE;

  for (i=0; i<ConSize; i++)
    /* any overloaded concentrator makes the string infeasible */
    if (ConWeight[i] > Capacity)
    {
      sum += ConWeight[i] - Capacity;
      violation++;
      Feasible = FALSE;
    }

  if (Feasible)
    return 0;
  else
    return (sum * violation + term);
```

```
}

/* Selects two strings from the population pool and performs crossover
   and mutation. Elitism is incorporated meaning that the best string
   in previous generation is preserved in the new generation
*/
void run ()
{
  long mate1,           /* first parent */
       mate2,           /* second parent */
       site_11,         /* first crossing site of first parent */
       site_12,         /* second crossing site of first parent */
       site_21,         /* first crossing site of second parent */
       site_22;         /* second crossing site of second parent */
  long i;

  /* elitism - preserving the best string in previous generation */
  for (i=0; i<size; i++)
  {
    MatePool[0][i] = Best[i];
    MatePool[1][i] = Best[i];
  }

  for (i=2; i<PoolSize; i+=2)
  {
    /* use roulette wheel to select the two parents */
    mate1 = roulette ();
    mate2 = roulette ();

    /* randomly choose two crossing sites for each parent */
    site_11 = rand () % (ConSize+1);
    do
      site_12 = rand () % (ConSize+1);
    while (site_12 == site_11);

    if (site_12 < site_11)
      swap (&site_11, &site_12);

    site_21 = rand () % (ConSize+1);
    do
      site_22 = rand () % (ConSize+1);
    while (site_22 == site_21);

    if (site_22 < site_21)
      swap (&site_21, &site_22);

    /* probability for crossover */
    prob = (double) rand () / RAND_MAX;

    /* generate first offspring */
    crossover (mate1, mate2, site_11, site_21, site_22, i);
    mutate (i);
    /* generate second offspring */
    crossover (mate2, mate1, site_21, site_11, site_12, i+1);
    mutate (i+1);
  }
}
```

```
/* Selects strings using the roulette technique. Each string has a
   probability of being selected inversely proportional to its fitness
   value; strings with smaller fitness values have higher chance of
   being chosen. */
long roulette ()
{
  long   i = -1;
  double goal,          /* sector on wheel to be met */
         sum = 0.0;     /* accumulated sectors */

  /* sector on the roulette wheel to be met */
  goal = (double) rand () / RAND_MAX;

  /* look for the string that falls into the specified sector */
  if (sum == goal)
    return 0;

  while (sum < goal && i < PoolSize-1)
    sum += (1.0 / Pool[++i][size-2]) / FitSum;

  return i;
}

/* Swaps the values of two variables.
*/
void swap (long *first, long *second)
{
  long change;

  change = *first;
  *first = *second;
  *second = change;
}

/* Performs crossover of two strings. Contents between the two crossing
   sites of the second parent are injected in the first crossing site
   of the first parent.
*/
void crossover (long mate1, long mate2, long site1, long site2,
                long site3, long pos)
{
  long i,
       loc = 0,         /* number of elements in array Mix */
       num = 0,         /* number of concentrators being injected */
       total = 0;       /* number of duplicates after injection */
  long index1 = TerSize,
       index2 = TerSize;

  /* perform crossover if probability is under crossover rate */
  if (prob < X_Over)
  {
    /* copy contents before first crossing site of mate1 */
    for (i=0; i<site1; i++)
    {
      Mix[loc++] = Pool[mate1][index1];

      while (Pool[mate1][++index1] < TerSize)
```

```
        Mix[loc++] = Pool[mate1][index1];
    }

    /* copy contents between two crossing sites of mate2 */
    for (i=0; i<site2; i++)
      while (Pool[mate2][++index2] < TerSize);

    for (i=site2; i<site3; i++)
    {
      Mix[loc] = Pool[mate2][index2] + ConSize;
      Conc[num++] = Mix[loc];
      loc++;

      while (Pool[mate2][++index2] < TerSize)
      {
        Mix[loc++] = Pool[mate2][index2];
        Dup[total++] = Pool[mate2][index2];
      }
    }

    /* copy contents after the first crossing site of mate1 */
    for (i=index1; i<size-2; i++)
      Mix[loc++] = Pool[mate1][i];

    /* perform heuristic algorithm to handle duplicate elements */
    heuristic (pos, total, loc, num);
  }
  else                    /* no crossover performed */
  {
    for (i=0; i<TerSize; i++)
      MatePool[pos][i] = Pool[mate1][i];

    for (i=TerSize; i<size-2; i++)
    {
      MatePool[pos][i] = Pool[mate1][i];
      Mix[i-TerSize] = Pool[mate1][i];
    }

    for (i=size-2; i<size; i++)
      MatePool[pos][i] = Pool[mate1][i];
  }
}

/* Performs a heuristic strategy to remove redundent concentrators;
   the old concentrators give way to newly injected concentrators.
*/
void heuristic (long pos, long total, long loc, long num)
{
  long i, j, index, hold_c, hold_n, done, sum;
  long hold = TerSize + ConSize;    /* maximum label of concentrators */

  /* remove old concentrators if new concentrators with the same
     labels are injected */
  for (i=0; i<num; i++)
  {
    j = -1;
```

```
  /* look for the ith concentrator to be replaced */
  while (Mix[++j] != Conc[i]-ConSize);
  hold_c = j;
  hold_n = 1;

  while (++j<loc && Mix[j]<TerSize)
    hold_n++;

  /* remove the concentrator by advancing the rest of the array to
     overwrite the concentrator and the terminals assigned to it */
  loc -= hold_n;
  if (hold_c < loc)
    for (j=hold_c; j<loc; j++)
      Mix[j] = Mix[j+hold_n];
}

/* after we remove the duplicate concentrators, we may still need
   to remove duplicate elements in the remaining concentrators */
for (i=0; i<total; i++)
{
  j = 0;
  done = FALSE;

  /* look for the ith duplicate element */
  while (j<loc && !done)
  {
    if (Mix[j]>=hold)
      while (++j<loc && Mix[j]<TerSize);
    else
    {
      if (Mix[j++]==Dup[i])
      {
        hold_n = j-1;
        done = TRUE;
      }
    }
  }

  /* if the element is found to be duplicated */
  if (done)
  {
    /* remove the element by advancing the remainder of the array
       by one position */
    for (j=hold_n; j<loc-1; j++)
      Mix[j] = Mix[j+1];
    loc--;
  }
}

/* relabel the newly injected concentrators */
for (i=0; i<loc; i++)
  if (Mix[i] >= hold)
    Mix[i] -= ConSize;

/* check if all terminals are assigned */
for (i=0; i<TerSize; i++)
  Dup[i] = i;
```

```
hold = 0;

for (i=0; i<loc; i++)
{
  if (Mix[i] >= TerSize)
    index = Mix[i] - TerSize;
  else
  {
    MatePool[pos][Mix[i]] = index;
    hold++;
    Dup[Mix[i]] = -1;
  }
}

for (i=0; i<ConSize; i++)
  List[i][0] = 0;

if (hold < TerSize)   /* not all terminals are assigned */
{
  for (i=0; i<ConSize; i++)
    ConWeight[i] = 0;

  index = 0;

  for (i=0; i<ConSize; i++)
  {
    j = Mix[index] - TerSize;

    while (++index<loc && Mix[index]<TerSize)
    {
      List[j][++List[j][0]] = Mix[index];
      ConWeight[j] += TerWeight[Mix[index]];
    }
  }

  /* use the first-fit technique to assign usassigned terminals */
  first_fit (pos, TerSize-hold);
}
else                  /* all terminals are assigned */
{
  for (i=0; i<TerSize; i++)
    List[MatePool[pos][i]][++List[MatePool[pos][i]][0]] = i;
}

index = TerSize;
hold = -1;

/* copy the resulting offspring to matepool */
for (i=0; i<ConSize; i++)
{
  while (Mix[++hold]<TerSize);

  MatePool[pos][index++] = Mix[hold];
  hold_c = Mix[hold] - TerSize;

  for (j=0; j<List[hold_c][0]; j++)
    MatePool[pos][index++] = List[hold_c][j+1];
```

```
  }

  MatePool[pos][index++] = fitness (MatePool[pos]);
  MatePool[pos][index] = Feasible;
}

/* Uses the first-fit technique to assign terminals. For terminals that
   are not assigned, find the first concentrator that can accommodate
   its capacity requirement. If no such concentrators exist, this
   terminal is randomly assigned to one concentrator.
*/
void first_fit (long pos, long total)
{
  long i, j, done, index = -1;

  /* assign all terminals currently not connected to any concentrator */
  for (i=0; i<total; i++)
  {
    /* look for the next unassigned terminal */
    while (Dup[++index] == -1);
    done = FALSE;
    j = 0;

    /* try to assign to the first available concentrator */
    while (j<ConSize && !done)
    {
      if (TerWeight[Dup[index]]+ConWeight[j] <= Capacity)
      {
        done = TRUE;
        List[j][++List[j][0]] = Dup[index];
        ConWeight[j] += TerWeight[Dup[index]];
        MatePool[pos][Dup[index]] = j;
      }
      else
        j++;
    }

    if (!done)          /* randomly choose a concentrator */
    {
      j = rand () % ConSize;
      List[j][++List[j][0]] = Dup[index];
      ConWeight[j] += TerWeight[Dup[index]];
      MatePool[pos][Dup[index]] = j;
    }
  }
}

/* Performs mutation on given string. The mutation operator walks
   through the group part of the string; for each concentrator it
   determines if mutation is to occur. If yes, all the terminals
   originally assigned to that concentrator are to be reassigned
   using the first fit strategy.
*/
void mutate (long pos)
{
  long i, j, index, hold;
  long total = 0,       /* number of unassigned terminals */
```

```
     flag = FALSE,
     loc = TerSize + ConSize;
long hold_c, hold_n, done;

for (i=TerSize; i<size-2; i++)
  Mix[i-TerSize] = MatePool[pos][i];

hold = -1;

for (i=0; i<TerSize; i++)
  Dup[i] = -1;

/* test each concentrator */
for (i=0; i<ConSize; i++)
{
  /* look for the next concentrator */
  while (Mix[++hold]<TerSize);

  /* perform mutation if the probability is less than mutation rate */
  if ((double) rand () / RAND_MAX < Mutation)
  {
    flag = TRUE;
    hold_c = hold + 1;
    hold_n = 0;

    /* remove all terminals assigned to this concentrator */
    while (hold+1<loc && Mix[hold+1]<TerSize)
    {
      hold_n++;
      hold++;
      Dup[Mix[hold]] = Mix[hold];
      total++;
    }

    loc -= hold_n;
    if (hold_c < loc)
      for (j=hold_c; j<loc; j++)
        Mix[j] = Mix[j+hold_n];
  }
}

/* if mutation is ever performed, some elements are missing from
   the assignment and thus need to be reassigned */
if (flag)
{
  for (i=0; i<ConSize; i++)
    List[i][0] = 0;

  for (i=0; i<ConSize; i++)
    ConWeight[i] = 0;

  index = 0;

  for (i=0; i<ConSize; i++)
  {
    j = Mix[index] - TerSize;
```

```
      while (++index<loc && Mix[index]<TerSize)
      {
        List[j][++List[j][0]] = Mix[index];
        ConWeight[j] += TerWeight[Mix[index]];
      }
    }

    /* use the first-fit technique to assign usassigned terminals */
    first_fit (pos, total);

    index = TerSize;
    hold = -1;

    /* copy the resulting offspring to matepool */
    for (i=0; i<ConSize; i++)
    {
      while (Mix[++hold]<TerSize);

      MatePool[pos][index++] = Mix[hold];
      hold_c = Mix[hold] - TerSize;

      for (j=0; j<List[hold_c][0]; j++)
        MatePool[pos][index++] = List[hold_c][j+1];
    }

    MatePool[pos][index++] = fitness (MatePool[pos]);
    MatePool[pos][index] = Feasible;
  }
}

/* Calculates the sum of fitness values for roulette wheel selection
   for the next generation. Also saves the best assignment obtained
   up to the current generation.
*/
void stat (long gen)
{
  long i, j, hold, count = -1;

  FitSum = 0.0;

  for (i=0; i<PoolSize; i++)
  {
    hold = MatePool[i][size-2];
    FitSum += 1.0 / hold;

    /* test if the fitness value is better than the best so far */
    if (hold < min)
    {
      min = hold;
      count = i;
    }
  }

  if (count != -1)
    for (i=0; i<size; i++)
      Best[i] = MatePool[count][i];
```

```
  if (gen%50==0)
    printf ("\nGENERATION %ld : BEST FITNESS VALUE = %ld\n", gen, min);
}

/* Replaces the entire population pool by the mating pool.
   This is the strategy used by generational genetic algorithms.
*/
void copy_back ()
{
  long i, j;

  for (i=0; i<PoolSize; i++)
    for (j=0; j<size; j++)
      Pool[i][j] = MatePool[i][j];
}

/* Outputs the overall best assignment obtained, both the cost and the
   actual assignment from terminals to concentrators.
*/
void print_best ()
{
  long i;

  printf ("\nBEST ASSIGNMENT FOR THIS RUN (COST = %ld) :\n\n", min);
  for (i=0; i<TerSize; i++)
    printf ("%d ", Best[i]);
  printf ("\n");
}

/* Frees up memory locations and closes files.
*/
void cleanup ()
{
  long i;

  for (i=0; i<ConSize; i++)
  {
    free (ConCoord[i]);
    free (List[i]);
  }

  for (i=0; i<TerSize; i++)
  {
    free (TerCoord[i]);
    free (Cost[i]);
  }

  for (i=0; i<PoolSize; i++)
  {
    free (Pool[i]);
    free (MatePool[i]);
  }

  free (TerCoord);
  free (ConCoord);
  free (TerWeight);
  free (ConWeight);
```

```
  free (Cost);
  free (Pool);
  free (MatePool);
  free (List);
  free (Mix);
  free (Dup);
  free (Best);
  free (Conc);

  fclose (info);
  fclose (init);
}

/***************************** End of File *****************************/
```