Heuristic Algorithms for the Terminal Assignment Problem

Sami Khuri **Teresa Chiu**

Department of Mathematics and Computer Science
San José State University
One Washington Square
San José, CA 95192-0103
khuri@jupiter.sjsu.edu
Fax: (408)924-5080

Keywords: *combinatorial optimization, terminal assignment problem, genetic algorithms, grouping genetic algorithm*

Abstract

In this paper, applications of heuristic techniques for solving the terminal assignment (TA) problem are investigated. The task here 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, i.e., is within the concentrator's capacity. Under these two hard constraints, an assignment with the lowest possible cost is sought. The proposed cost is taken to be the distance between a terminal and a concentrator.

The heuristic techniques we investigate in this article include greedy-based algorithms, genetic algorithms (GA), and grouping genetic algorithms (GGA) [4]. We elaborate on the different heuristics we use, and compare the solutions yielded by them.

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 and the cost of linking each terminal to a concentrator are 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 intractability of this problem [6] is a motivation for the pursuit of heuristics that produce approximate, rather than exact, solutions.

In the special case where all concentrators have the same capacity and all terminals the same weight, the problem is no longer NP-complete, since the alternating chain algorithm that runs in polynomial-time [6] always yields the optimal solution.

The name "terminal assignment" has its origins in the network industry. We adopt the terminology found in [1], but extend the problem to the general case where concentrators have different capacities. We thus augment the data found in [1] (in which all concentrators have the same capacity), by assigning a capacity (a randomly generated value) to each concentrator.

In the literature of operations research, there are a number of combinatorial optimization problems that share the name "assignment problem", such as the *weighted matching problem* [9], and the *Hitchcock problem* [10], but whose underlying ideas are quite different from that of the terminal 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 genetic-based algorithms.

The potential of genetic algorithms to yield good solutions, even for hard optimization tasks, has been demonstrated by various applications. A recently developed area in the GA research, *grouping genetic algorithm* (GGA) [4], 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 rest of this work is organized in the following fashion. In Section 2 we give the formal definition of the terminal assignment problem. Addressing the different heuristic algorithms adopted in this work, Section 3 introduces the greedy-based heuristic. The GA implementation for the TA problem is given in Section 4, including an elaboration on the two packages adopted in this work, GENEsYs [2] and LibGA [3]. In Section 5, the implementation of the GGA for the TA problem is discussed. In Section 6, the test cases used in this work are introduced, as well as the results and analysis of the performance of the different heuristic algorithms.

2 The Terminal Assignment Problem

The following is a formal definition of the terminal assignment problem where we make use of Stinson's terminology for combinatorial optimization problems [12]. 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:	$r_1, r_2, \ldots, r_C$
Capacities:	$p1, p2, \ldots, p_C$

w_i is the weight, or capacity requirement of terminal l_i. The weights and capacity are positive integers and $w_i < \min\{p_1, p_2, \ldots, p_C\}$ 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 $\vec{x} = x_1 x_2 \ldots x_T$ 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 $i = 1, 2, \ldots, T$ (i.e., all terminals have to be assigned) and $\sum_{i \in R_j} w_i \le pj$, for $j = 1, 2, \ldots, C$ (i.e., 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

$$Z(\vec{x}) = \sum_{i=1}^{T} cost_{ij},$$

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

Optimal solution:

A feasible vector $\vec{x}$ that yields the smallest possible $Z(\vec{x})$.

Example

Table 1 indicates a collection of $T = 10$ terminal sites and $C = 3$ concentrator sites. The weight requirement and the coordinates based on a 100×100 Euclidean grid for each terminal site are specified in Table 1(a). The coordinates for the concentrator sites and their capacities are listed in Table 1(b). The cost of assigning a terminal to a concentrator is the Euclidean distance between them rounded to the nearest integer.

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}.

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)

Table 1: Terminal capacity requirements (weight) and terminal coordinates.

Concentrator	Capacity	Coordinates
1	12	(19, 76)
2	14	(50, 30)
3	13	(23, 79)

Table 2: Concentrator capacities and coordinates.

The total cost, computed as the sum of the costs for the 9 selected links, is 223.

The problem has feasible solutions though. By interchanging terminals l_4 and l_8 for instance, and assigning them to concentrators r_1 and r_2, respectively, r_1 will have enough room to accommodate l_{10}.

In the next section, our first heuristic strategy, based on the greedy technique, is introduced.

3 Greedy Algorithms

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

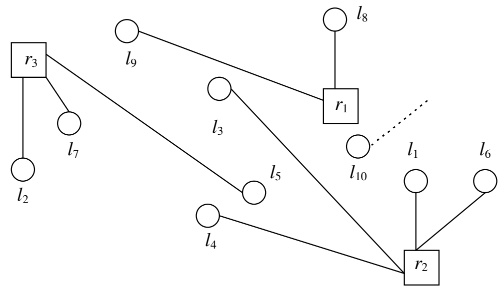

Figure 1: Terminal assignments to concentrators Total $cost = 223$ with terminal 10 stranded.

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 [6].

Procedure Greedy;

```
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
    }
}
```

Procedure Greedy is likely to force the terminals that are considered last to be connected to concentrators that are very far away. It can fail to produce a feasible solution when: (1) the total concentrator capacity is less than the total terminal capacity requirement, or (2) there is not a feasible solution to the problem instance, or (3) the algorithm misses the feasible solution(s). The infeasible solution represented in Figure 1 of the previous section was obtained by Procedure Greedy.

We also implemented and tested another greedy-based algorithm which did not produce good results and is therefore not included in the section in which we report our results. In the unsuccessful attempt, all connections from terminals to concentrators are first sorted according to their costs in 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 algorithm 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.

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

4 The Genetic Algorithms

The first step for the GA implementation involves choosing a representation 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. For example, Figure 2 indicates the chromosome representing a possible solution to the problem instance given in the Example of Section 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

1	2	3	4	5	6	7	8	9	10
2	3	1	2	2	2	3	1	3	1

Figure 2: Chromosome representing a solution to the Example in Section 2.

concentrator 3, terminal 3 to concentrator 1, and so on.

This chromosome, with integer values, can easily be implemented under any GA package that permits integer representation, such as LibGA [3]. However, for the sake of comparison, we have also adopted a second GA package, GENEsYs [2], which allows only binary representations for strings. In this case, some mapping from binary strings to non-binary strings is necessary. For instance, the binary representation of the chromosome of Figure 2 used by GENEsYs, is illustrated in Figure 3.

1	0	1	1	0	1	1	0	1	0	1	0	1	1	0	1	1	1	0	1

Figure 3: 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.

In our study, we use generational genetic algorithms with roulette wheel selection, uniform crossover, and uniform mutation.

As for the fitness function, 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 incorporated in it 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 at 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. In designing fitness functions for problems that do have infeasible strings, we follow the suggestions found in [7] and [8] 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 [11].
- 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 separation 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 which calculates the total cost of all connections. Namely,

   $Z(\vec{x}) = \sum_{i=1}^{T} cost_{ij}$, where $\vec{x} = x_1\, x_2\, \ldots\, x_T$ 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 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.

As mentioned earlier, the two software packages we use for our experimental runs are GENEsYs [2] which was based on Grefenstette's GENESIS [5], and LibGA [3]. Both are implemented in C under the UNIX platform.

In the next section, 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.

5 Grouping Genetic Algorithms

To capture the grouping entity of the problem, the simple (standard) chromosome is augmented with a group part, where the groups are encoded on a one gene for one group basis. For example, the chromosome of Figure 2 with the genetic algorithm (see Section 4) is represented by the string in Figure 4.

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

Figure 4: Chromosome in GGA representing the chromosome of Figure 3 in Section 4.

The first half is the object part and is identical to the entire chromosome in GA, while the second half is the group part. In the object part, the value in the ith position indicates the concentrator to which the ith 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.

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.

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

Assume that 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 duplicates. 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 qthe first–fit heuristic to re–assign these objects: it visits the groups one at a time, assigns the object to the first group that is able to service it, 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 guidelines suggested by [4].

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 4. Assume that 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 it. 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 section, we present and compare the results obtained by executing our heuristic algorithms on several problem instances.

6 The Experimental Runs

The greedy algorithm, the genetic algorithm, and the grouping genetic algorithm are applied to ten different collections of 100 terminal sites. As mentioned in the Introduction, we used the problem instances found in [1] to which each concentrator is given a capacity: a random integer in the range 15–25. In [1], each concentrator has the same capacity value of 12. The number of concentrators for the 10 problem instances of size 100 is between 27 and 33.

The results of executing all three heuristic algorithms with the 10 problem instances are summarized in Table 2.

The results tabulated under the Greedy Algorithm in Table 2 are the best results we obtained after 20,000 runs on each of the 10 problem instances. For all the GA experiments, we use the same number of generations (20,000),

Problem Instances	Greedy Algorithm	Genetic Algorithm			
		GENEsYs	LibGA	GGA	Seeding
100_1	928	963	928	1115	Y
100_2	935	973	935	1157	N
100_3	846	881	846	1002	Y
100_4	1076	1123	1069	1321	Y
100_5	1116	1178	1111	1360	N
100_6	1071	1106	1070	1208	Y
100_7	1315	1404	1279	1631	N
100_8	1211	1225	1186	1568	N
100_9	1088	1169	1065	1291	Y
100_10	902	977	901	1117	Y

Table 3: Best solutions of applying the heuristics with the 10 problem instances.

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 use mutation rates that vary between 0.025 and 0.1. 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. Seeding was used for some of the runs with GA and GGA and are marked with "Y" in Table 2. In other words, for these instances, the initial population was formed by randomly generating 80% of the strings, while the remaining 20% were feasible strings obtained by the greedy algorithm.

As indicated in Table 2, for all of the problem instances, the genetic algorithm implemented using LibGA yields the best solution among the genetic algorithm heuristics, and in 7 cases gives better solutions than the greedy–based algorithm. For the remaining 3 instances (100_1, 100_2, and 100_3), it shares the best solutions with Procedure Greedy, which was specifically tailored for the TA problem. We strongly suspect that for these 3 problem instances, the solutions obtained by LibGA (and Procedure Greedy) are the global optima. 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.

7 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. Even though our findings may not be conclusive, the results tend to suggest that the genetic algorithms work well with the TA problem, which is after all, a highly constrained, combinatorial optimization problem.

Acknowledgments

The authors would like to thank Dr. John Mitchem for comments and suggestions that improved this work, and Helko Lehmann for getting the paper in camera–ready form.

References

[1] Abuali, F., Schoenefeld, D., and Wainwright, R. (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, T. (1992). GENEsYs 1.0. *Software distribution and installation notes*, Systems Analysis Research Group, LSXI, University of Dortmund, Dortmund, Germany.

[3] Corcoran, A. and Wainwright, R. (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.

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

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

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

[7] Khuri, S., Bäck, T., and Heitkötter, J. (1994). An Evolutionary Approach to Combinatorial Optimization Problems. *Proceedings of the 22nd ACM Computer Science Conference*, pp. 66–73. Phoenix, Arizona.

[8] Khuri, S., Bäck, T., and Heitkötter, J. (1994). The Zero/One Multiple Knapsack Problem and Genetic Algorithms. *Proceedings of the 1994 ACM Symposium on Applied Computing*, pp. 188–193. Phoenix, Arizona.

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

[10] Papadimitriou, C. and Steiglitz, K. (1982). *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Inc., New Jersey.

[11] Richardson, J., Palmer, M., 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.

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