8-2011

A genetic algorithm for the sensor location problem.

Di Zhang
University of Louisville

A GENETIC ALGORITHM FOR THE SENSOR LOCATION PROBLEM

By

Di Zhang

A thesis
Submitted to the Faculty of the
University of Louisville
J.B. Speed School of Engineering
as Partial Fulfillment of the Requirements
for the Professional Degree

Master of Science

Department of Industrial Engineering
University of Louisville
Louisville, Kentucky

August, 2011

A GENETIC ALGORITHM FOR THE SENSOR LOCATION PROBLEM

Submitted by:

Di Zhang

A thesis Approved on

07 / 21 / 2011

(Date)

by the Following Reading and Examination Committee:

Dr. Lihui Bai Thesis Director

Dr. Gerald Evans

Dr. Csaba Biro

ACKNOWLEDGMENTS

I would like to thank my thesis advisor, Professor Lihui Bai, for her guidance and patience. She gave me this great opportunity to be involved in this challenging project. Her guidance and inspiration along the way were so helpful and her patience and encouragement always gave me confidence.

I also want to thank Dr. Gerald W Evans and Dr. Csaba Biro. Their insightful suggestions and support help me to improve this thesis.

I also want to thank my dear friends. Their friendship makes my life sunny and colorful. Their support on all aspects of my life are greatly appreciated. Finally, but not the least, I want to thank my dear parents for their unconditional love and understanding, whithout which none of my achievements today is possible.

ABSTRACT

A GENETIC ALGORITHM FOR THE SENSOR LOCATION PROBLEM

Di Zhang

July,21st,2011

We study a sensor location problem that minimizes the total number of sensors to install at road intersections in a transportation network so that the traffic flows on the entire network are uniquely determined. We employ the concepts of hidden network and incremental flow in analyzing the problem, and propose a genetic algorithm for its solution for large-size networks. The algorithm is programmed in Matlab and tested on randomly generated network. Numerical results suggest the algorithm is efficient.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Today, traffic congestion is a pressing issue for society and a major concern for urban planners. The 2007 Urban Mobility Report (2007) states that congestion in the U.S. caused 4.2 billion hours of travel delay and 2.9 billion gallons of wasted fuel for a total cost of $78 billion in 2005. Evidently, traffic management and control in urban transportation networks becomes an important function of our society. The recent concept of congestion pricing aims to shift traffic volumes to alternative routes or times via certain pricing policies. This requires monitoring flows on the transportation network in order to accurately estimate the traffic volume on all roads of the network. With the advent of communication technologies such as sensors and vedio cameras, traffic engineers are capable of monitoring traffic networks in real time. However, as often is the case with congested metropolitan areas, the transportation networks

are large enough that only part of the network can be monitored due to the high installation and maintenance costs associated with monitoring devices (e.g., passive and/or active sensors, vedio cameras). Thus, a relevant problem facing traffic engineers is to determine sensor locations that will better estimate the traffic flows on the entire network at the minimum cost.

In this thesis, we study the **Sensor Location Problem (SLP)** that minimizes the total number of sensors required so that the traffic flows on the entire network are uniquely determined. Note that the decisions include not only the minimum number of sensors but also their locations in the network. Specifically, we make similar assumptions as most other related works in the literature. We assume that the network is symmetric and the sensors are place at nodes, i.e., road intersections.

In the operations research and transportation science literature, Bianco et al. (2001) study the SLP and present some theoretical results as well as heuristic methods. However, Morrison (2008) shows a counterexample to point out that the theorem in Bianco et al. (2001) is not true in general. In 2010, Rubin (2010) develop the concepts of "hidden network" and "incremental flow" in an attempt to solve the problem via integer linear programming approach. He proposes an exact

solution method to be solved by general purpose solver such as CPLEX (2010). In our numerical experiments, CPLEX failed to solve problems with even 20 nodes.

Therefore, the focus of this thesis is to develop a heuristic method for the sensor location problem that can solve real-size problems in reasonable CPU time. Using concepts such as turning ratio and turning factors introduced in Morrison (2008) and "hidden network" and "incremental flow" in Rubin (2010), we propose a **genetic algorithm** to solve the SLP for large-scale networks. Our numerical study shows that the genetic algorithm is efficient at solving real size sensor location problems. Furthermore, we lay out several possible improvements to the algorithm that may expedite the solution.

For the remainder of the thesis, Chapter 2 offers a review of limited literature on the sensor location problem that are existant, Chapter 3 formally introduces the problem, Chapter 4 presents the customized genetic algorithm along with two possible improvements, Chapter 5 reports the numerical results under various parameters settings and finally Chapter 6 concludes the thesis with several future directions.

CHAPTER II

LITERITURE REVIEW

The sensor location problem (SLP) is studied in the literature from areas of operations research, graph theory, electric engineering and civil engineering, to name a few. An earlier paper on the sensor location problem is by Yang and Zhou (1998). Given an origin-destination (O-D) distribution, this paper determines the optimal number and locations of traffic counting points in a road network. The O-D matrix is the matrix with potential travel origin along the rows and destinations along the columns. The (i, j) entry of an O-D matrix is the total number of trips from origin i to destination j taken by all commuters in the transportation network. By adopting the O-D matrix method the SLP can be solved using linear algebra. In the paper, Yang and Zhou propose four rules for determining traffic counting locations:

Rule 1 (O-D covering rule): for observing part of trips between any O-D pair, the traffic counting points should be located on a road network.

Rule 2 (maximal flow fraction rule): for a particular O-D

pair, the traffic counting points on a road network should be located at the links so that the flow fraction between this O-D pair out of these links is as large as possible.

Rule 3 (maximal flow-intercepting rule): the chosen links should intercept as many flows as possible under a certain number of links to be observed.

Rule 4 (link independence rule): the traffic counting points should be located on the network so that the resulting traffic counts on all chosen links are not linearly dependent.

Rule 1 is fundamental that any network should satisfy. Rules 2 and 3 could be combined in the objective function. Therefore, each new observed link should produce more information and it is preferred to observe those points that could bring as more information as possible within the network. Rule 4 means that the links which cannot provide any new information should be excluded. Note that it can be difficult to satisfy both rules 2 and 3, as they often conflict with each other. The latter is because roads with high volumes of traffic will generally have only two cases: most of the traffic going to the same places or traffic going to many different places, and these two cases generally do not occur at the same time. Thus, Yang and Zhou (1998) combine these two rules as parameters in a heuristic search function to find the best locations for sensors.

In a related paper, Bianco et al. (2001) defines the SLP as determining the minimum number as well as locations of counting points in order to deduce all traffic flows in a transportation network. They study a network G = (N, A) where N is a set of nodes corresponding to road intersections and A is a set of links corresponding to streets between intersections. M is a set of nodes to place sensors. A link $(v, w) \in A$ goes from its tail-node v to its head-node w, and nodes v and w are said to be adjacent. In the set of nodes N, the node at which trips originate and/or terminate is called a centroid, and the set of centroids is denoted as S. Finally a cutset is the set $C_M = \{(v, w) \in A: v, w \in (M \cup A(M))\}$, where M is the set of monitored nodes and A (M) is the set of nodes adjacent to monitored nodes. With these notation and assumptions, Bianco et al. (2001) present an important proposition to construct a necessary condition to uniquely determine the nodes and flows in the network. Consequently, Bianco et al. (2001) proposed two heuristic methods for determining the lower bound and the upper bound for the number of sensors that will deduce traffic flows on the entire network.

The computational results in Bianco et al. (2001) show that the heuristic methods do not perform well in small networks, but have improved as the network becomes larger, i.e., empirically when the number of nodes is more than 200.

However, as Morrison (2008) points out, the proposition in Bianco et al. (2001) that is important to their heuristic algorithms is incorrect. A counterexample is provided in Morrison (2008). In other words, he establishes a network and a set of monitored nodes that satisfy Bianco et al.'s proposition, but the network is not uniquely determined. Furthermore, Morrison (2008) presents a new approach to the SLP, which is largely a linear algebra method. In particular, Morrison (2008) argues that simply comparing the numbers of unknowns and equations in the linear system as did in Bianco et al. (2001) does not correctly answer the question: whether or not the flows in the digraph are uniquely determined. Obviously one reason is that some of the flow balancing equations are not linear independent. On the other hand, in order to better understand the conditions under which the Bianco et al.'s proposition fails, Morrison (2008) examines the associated incidence matrix of the SLP, and proposes studying an augmented incidence matrix. Finally, perhaps most importantly, Morrison (2008) introduces the notion of canonical link to make use of turning ratio and tuning factors in studying the characteristics of solutions to SLP. Specially, Morrison defines the B-path in the network and states a conjecture that uses B-path to characterize valid solutions to the problem.

Lastly, we review the most recent work on the sensor location

problem by Rubin (2010). Rubin's approach is integer linear programming and network flows. Particularly, Rubin (2010) works with the same problem as in Bianco et al. (2001) and Morrison (2008). Given a network consisting of two types of nodes: terminal and transit nodes, Rubin (2010) assumes that the ratios between outgoing flows at every node are strictly positive. In Morrison (2008)'s term, the turning ratios (split ratios) are positive. Furthermore, Rubin (2010) works on the SLP problem where the turning ratios at all nodes are not known until the decision is made as to where to place sensors in the network. Rubin (2010)'s approach use the following three basic facts:

Fact 1: Conservation of flow (balancing flow) applies at all transit nodes.

Fact 2: Monitoring a node gives complete knowledge on the flows on all arcs incident to the node. Thus, it is implied that flows entering/leaving the monitored terminal node are known.

Fact 3: If we know the flow on any outward arc at any node, we know the flow on all outward arcs of that node via the turning ratios.

Fact 4: If we know the flows on all outward arcs and all but one inward arc at a transit node, we can deduce the flow on the last inward arc by flow conservation.

Instrumental to Rubin's approach is the concept of ***"hidden network,"*** which is derived from the original network for a given set of monitored nodes. In words, hidden network consists of arcs whose flows cannot be deduced from monitoring the given set of monitored nodes, and all nodes that are incident to these arcs. We will introduce the formal definition of the hidden network in Chapter 4. The benefit of hidden network is that it makes solving the SLP more systematic and algorithmic. Using the hidden network associated with a given set of monitored nodes, Rubin (2010) reduces the problem of uniquely determining the flows in the original network to the one of proving non-existence of any incremental flows in the hidden network. Specifically, suppose we arbitrarily select a set of nodes to monitor in the network, then the flows on the entire network are uniquely determined by monitoring these nodes if and only if the associated hidden network has no non-zero flows (named as "incremental flows") with turning ratios at all nodes being positive. Finally, Rubin (2010) also presents one conjecture to determine whether there exists a nonzero, conservative incremental flow for the hidden network.

In the next Chapter, we will formally define network notations, introduce terminology specific to the sensor location problem, and most importantly, the solution concepts of hidden network and incremental flows. Furthermore, we will provide

illustrative examples to explain these concepts. Finally, we present the mathematical formulation of the sensor location problem.

CHAPTER III

PROBLEM STATEMENT

3.1 Basic Definitions

As in Morrison (2008) and Rubin(2010), we assume that graph is symmetric, i.e., if there exists one arc between two nodes, the opposite arc between the two nodes also exists. In other words, the networks we investigate in this thesis are two-way directed networks. In addition, we assume that sensors are placed at nodes in the network., If one node is monitored, then all flows coming from or going to, this node are known.

Our goal is to deduce the flow on all arcs in the network, so some laws about network flows need to be explained. In Morrison (2008), the flow conservation law at each node v ∈ V is given as follows:

$$\sum_{e \in v^-} f_e - \sum_{e \in v^+} f_e + S_v = 0, \tag{3.1}$$

where v^- is the set of outgoing arcs at node v, and v^+ is the set of incoming arcs at node v, and S_v is the balancing flow

at node v. Node v is called source node if the balancing flow is positive, sink node if the balancing flow is negative and transit node if the balancing flow is zero. More formally, we have the following definition.

Definition 3.1.1. If the **balancing flow** of a node v is zero, node v is a transit node.

Definition 3.1.2. If the **balancing flow** of a node v is non-zero, node v is a terminal node.

Here we follow the same definition as in Rubin (2010). (Note that in Morrison (2008), terminal nodes are called bounded nodes.) In addition, we refer to the nodes with sensor as monitored nodes, and denote the set of monitored node by M, and the set of transit nodes by T. Note that the monitored node could be terminal or transit.

Finally, we formally introduce turning ratio and turning factor as in Morrison (2008).

For every node v, we associate with each outgoing arc vu a real number $c_{vu} \in [0, 1]$, which is the fraction of the total outgoing flow from v that leaves on arc vu. That is,

$$f_{vu} = c_{vu} \sum_{e \in v^-} f_e \tag{3.2}$$

Then we can write the flow of all outgoing arcs from v in terms of a single selected outgoing arc:

$$f_{vu} = \frac{c_{vu}}{c_{vw}} f_{vw} \tag{3.3}$$

For any given node v ∈ N, among all the outgoing arcs, one can be designated as the **canonical exit arc** is defined below.

Definition 3.1.3 (Rubin, 2010). The **canonical exit arc** vw for a node v ∈ N is an arbitrarily selected arc (v, w) ∈ A with tail v.

Definition 3.1.4 (Morrison, 2008). The **turning factor** of arc vu with respect to some arc vw, denoted α_{vu}, is the ratio of the turning ratio of arc vu to the turning ration of arc vw (in general it will be clear from the context what arc the ratio is taken with respect to):

$$\alpha_{vu} = \frac{c_{vu}}{c_{vw}} \tag{3.4}$$

Then the equation (3.3) becomes

$$f_{vu} = \alpha_{vu} f_{vw} \tag{3.5}$$

Using the turning factor, the outgoing flows from node v are completely determined by the flow f_{vw} on the canonical exit arc from node v and the turning factors α_{vu}.

The sensor location problem we investigate assumes that the turning ratios are known everywhere after placing the sensors, and that the turning ratios are evenly distributed. The latter implies that the turning factors are always be one. In practice, the turning ratios can be different, but the methodology we propose in this thesis still apply.

In the next section, we will use examples to illustrate the process of deducing flows for the entire network by using turning factors as well as the balancing flow equations at transit nodes.

3.2 Illustrative Examples

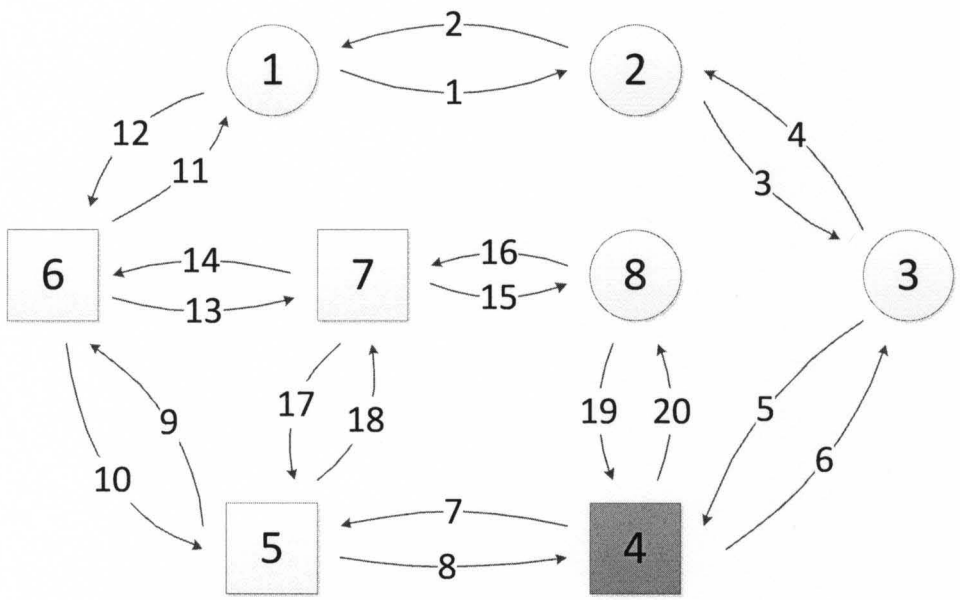

Figure 1 - Example 1 Network

Example 1: Suppose that we have a network with 8 nodes as show in Figure 1. Note that the nodes in circles represent transit nodes at which the balancing flow is zero; and the nodes in squares represent terminal nodes that could be sink or source nodes. Furthermore, indices for nodes are noted inside the circular or square nodes, while those for arcs are noted across the arc-shaped arrows. For example, terminal nodes 6 and 7 are connected by arcs 13 and 14 along opposite directions.

Suppose the sensor is placed at node 4. First, all the flows on the arcs that are incident to node 4 are known immediately, and they are f_5, f_6, f_7, f_8, f_{19}, and f_{20}. Secondly, using turning ratios, we can deduce all the outging flows from nodes 3, 5

and 8 respectively. For example, because the outging flow f_5 from node 3 is known, the outgoing flow from node 3 $f_4 = c_{4,5} * f_5$ becomes known as well. Similarly, we can deduce outgoing flows f_9 and f_{18} from node 5 by $f_9 = c_{9,8} * f_8$ and $f_{18} = c_{18,8} * f_8$, respectively, because f_8 is known. Also, flow f_{16} is known due to $f_{16} = c_{16,19} * f_{19}$. Thirdly, observing that node 3 is a transit node, we apply the following flow conservation equation:

$$f_3 + f_6 - f_4 - f_5 = 0 \qquad (3.6)$$

In equation (3.6), f_5 and f_6 are known from reading the sensor, and f_4 is deduced as described above. So one can easily calculate f_3 from equation (3.6). Right now flows on the arc 3, 4, 5, 6, 7, 8, 9, 16, 18, 19, 20 are known.

Similarly, we can deduce all the flows in the network and finally get the whole network known. Observing that node 8 is also a transit node, we apply the flow conservation equation below:

$$f_{15} + f_{20} - f_{19} - f_{16} = 0 \qquad (3.7)$$

And f_{20}, f_{19}, f_{16} are already known above, f_{15} can be calculated easily from equation (3.7). Having f_{15} known, we can deduce the outgoing flow f_{14}, f_{17} from node 7 by $f_{14} = c_{14,15} * f_{15}$ and $f_{17} = c_{17,15} * f_{15}$. Similarly, having f_3 known, f_2 can be deduced

by $f_2 = c_{2,3} * f_3$, and observing node 2 which is a transit node, f_1 can be known from the following equation:

$$f_1 + f_4 - f_2 - f_3 = 0 \qquad (3.8)$$

In the equation (3.8), f_4, f_3 are known before and f_2 is deduced above. Then from node 1, we can deduce the f_{12} using turning factor, $f_{12} = c_{12,1} * f_1$. And then observing node 1, f_{11} can be known using following equation:

$$f_{11} + f_2 - f_{12} - f_1 = 0 \qquad (3.9)$$

In the equation (3.9), f_2, f_{12}, and f_1 are known. Finally, observing node 6, we can deduce f_{13} and f_{10} seperately by $f_{13} = c_{13,11} * f_{11}$ and $f_{10} = c_{10,11} * f_{11}$. Rightnow the whole network is revealed.

Example 1 shows that by repeatedly applying the flow conservation equation (3.1) at transit nodes and the turning ratio equation (3.3) at any node, one can reveal the flows on the entire network completely. However, this is not the case with the next example.

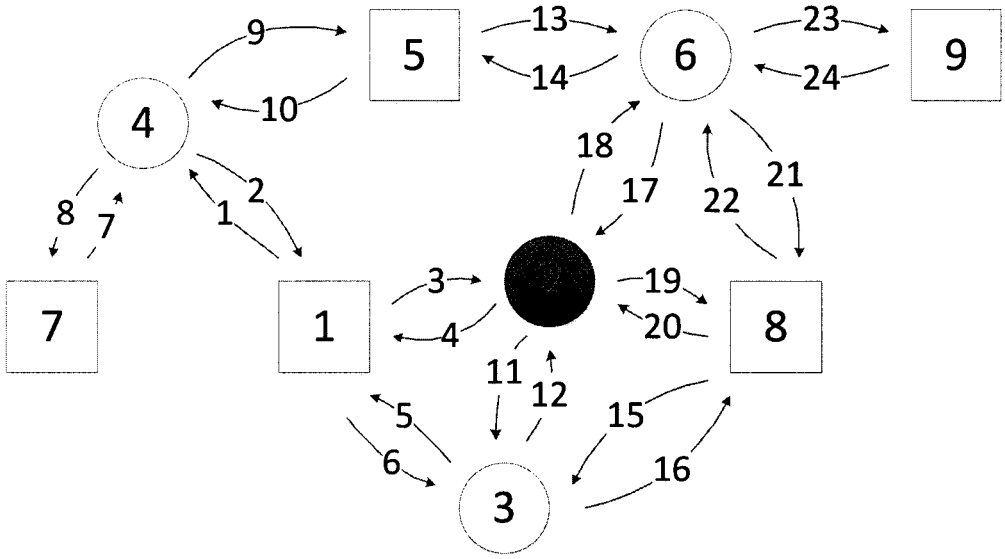

Figure 2 - Example 2 Network

Example 2: Suppose that we have a network with 9 nodes as shown in Figure 2. In this network, nodes 1, 5, 7, 8, and 9 are terminal, and nodes 2, 3, 4, and 6 are transit nodes.

Intuitively, in order to get as much information as possible, the sensor should be placed on the node where most arcs are incident to. Suppose we place a sensor at node 2. Immediately, we know the flow on that arcs that are incident to node 2, namely, arcs 3, 4, 11, 12, 17, 18, 19 and 20. Then, using the turning ratio equation at node 1 and flow f_3 we can deduce the flow on arcs 1 and 6. Then, using the turning factor equation at node 3 and flow f_{12}, we can deduce the flows on arcs 5 and 16. Similarly, using the turning factor equation at node 8 and flow f_{20}, we can deduce the flows on arcs 15 and 22. Moreover, using the turning factor equation at node 6 and flow f_{17}, we

can deduce the flows on arcs 14, 21 and 23.

Thus far, we know the flows on arcs 1, 3, 4, 5, 6, 11, 12, 14, 15, 16, 17, 18, 19, 20, 21, 22 and 23. It can be shown that without additional sensors, flows on arcs 2, 7, 8, 9, 10, 13 and 24 will remain unveiled.

In an attempt to completely reveal the network, Example 3 places an additional sensor at node 1 as shown in Figure 3.

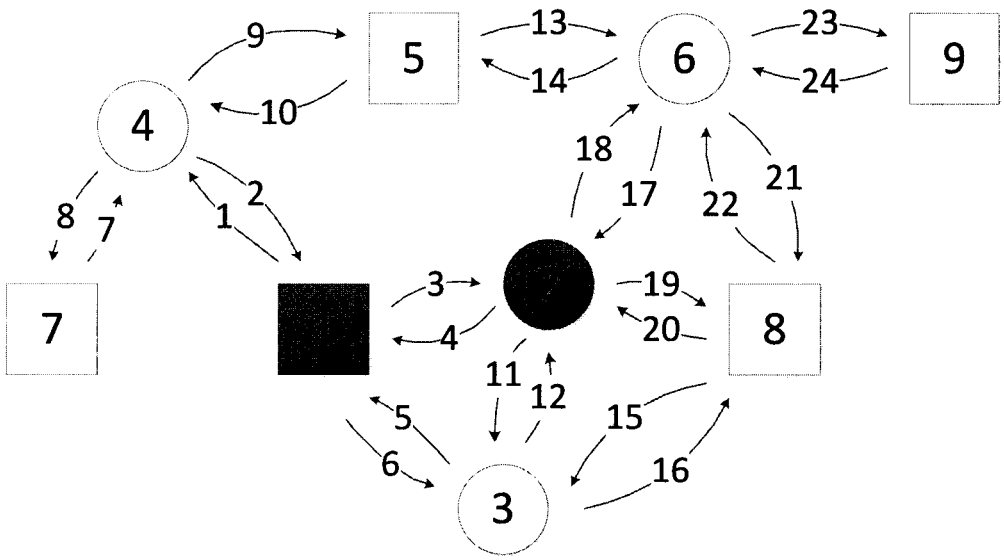

Figure 3 - Example 3 Network

Example 3: With monitoring node 1, the flow on arc 2 becomes known now. Consequently, we can deduce the flows on arcs 8 and 9. But the remaining part of the network, i.e., flows on arcs 7, 10, 13 and 24 are still not determined. Indeed, one can show that if the second sensor is place at node 5 instead of node 1, the entire network would be uniquely determined.

This makes sense as nodes 1 and 2 are spatially next to each other, the added benefit of monitoring node 1 may not be as great at monitoring node 5 which is farer from the already monitored node 2.

Examples 2 and 3 show that for some transportation network, it is necessary to monitor more than one nodes in order for the flows on the entire network to be revealed. Thus, the goal in this thesis is to solve the problem that places the minimum number of sensors so that the entire network flows are uniquely determined. Additionally, Example 3 suggests that when placing multiple sensor becomes necessary, keeping these sensor fairly dispersed may be effective.

Finally, it is worth noting that another variant of the sensor location problem is to determine where to place a pre-determined number of sensors in a transportation network so that the number of arcs whose flows are uniquely determined is maximal. This is a practical problem when the network under consideration is large and the budget only allows for a very limited number of sensors to be installed. This variant of the problem is similar to maximal-covering location problem, we will discuss it in Chapter 6.

In the next section, we will introduce some network notations that are important to our solution approach to the SLP.

3.3 Node Arc Incidence Matrix

The **incidence matrix** is a standard way of representing a graph. Below we state the definition of the node arc incidence matrix (NAIM).

Definition 3.3.1. The **Node Arc Incidence Matrix** (NAIM) of a directed graph D is a $|N| \times |A|$ matrix where the $(i, j)^{th}$ entry is -1 if the tail node of arc j is at node i, 1 if the head node of arc j is at node i, and 0 if arc j is not incident to node i.

Based on the definition above, the NAIM of Example 1 is as follows:

$$NAIM_1 = \begin{matrix} -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & -1 & 1 \end{matrix}$$

Similarly, the NAIM of Example 2 is:

$$NAIM_2 = \begin{bmatrix}
-1 & 1 & -1 & 1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & -1 & 1 & 0 & 0 & -1 & 1 & -1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1
\end{bmatrix}$$

From these two node-arc incidence matrices, we observe the following properties of such matrices. These properties are helpful when we randomly generate networks for our computational tests. First, the element of any NAIM should be 0, 1 or -1. Second, each row or column cannot be all zeros. Third, there must be one and exactly one pair of 1 and -1 in any column. Finally, in a symmetric graph, there must be at least one pairs of 1 and -1 in each row.

Similarly, one can use **Transit-Node Vector** (T) to represent the information about terminal versus transit nodes in the diagraph. The k^{th} element of this vector is 1 if node k is a transit node and 0 if node k is a terminal node. Below is the TNV for Example 1:

$$T_1 = (1 \quad 1 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1) .$$

Similarly, the TNV for Example 2 is:

$$T_2 = (0 \quad 1 \quad 1 \quad 1 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0) .$$

Finally, we can use **Monitored-Node Vector** (M) to represent the information about monitored nodes in the diagraph. The m^{th} element of this vector is 1 if node m is monitored and 0 if node m is not monitored. Below is the MNV for Example 1:

$$M_1 = (0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0).$$

Similarly, the MNV for Example 2 is

$$M_2 = (0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0),$$

and the MNV for Example 3 is

$$M_3 = (1 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0).$$

Once the diagraph is represented in the NAIM format, it is convenient to program our algorithm in Matlab.

3.4 Hidden Network

In the literature, Rubin (2010) proposes the concept of ***hidden network*** as we introduce below. The benefit of studying hidden network is that it reduces the sensor location problem, highly combinatorial in nature, to a network flow problem. The latter can be formulated as a mixed integer linear programming problem, and solved to optimality by general-purpose solver

such as CPLEX (CPLEX, 2010) when the network size is small.

Definition 3.4.1. (Rubin, 2010). A diagraph D = (N, A), N is the set of nodes and A is the set of arcs. Given a proposed monitoring set $M \subseteq N$, the corresponding **hidden network** $H_M = (N_M, A_M) \subseteq D$ consists of those arcs $a \in A$ whose flows f_a cannot be deduced from monitored flows f_m, $m \in M$ and those nodes $n \in N$ with at least one "hidden" arc incident on them.

Based on this definition, the hidden network should be the subset of the original network that is unknown. For Example 1, there is no hidden network because flows on all arcs can be deduced from monitoring node 4. For Example 2, the hidden network is shown in Figure 4 and it consists of undetermined arcs 2, 7, 8, 9, 10, 13 and 24 and nodes that are incidental to these arcs, i.e., nodes 1, 4, 5, 6, 7 and 9.

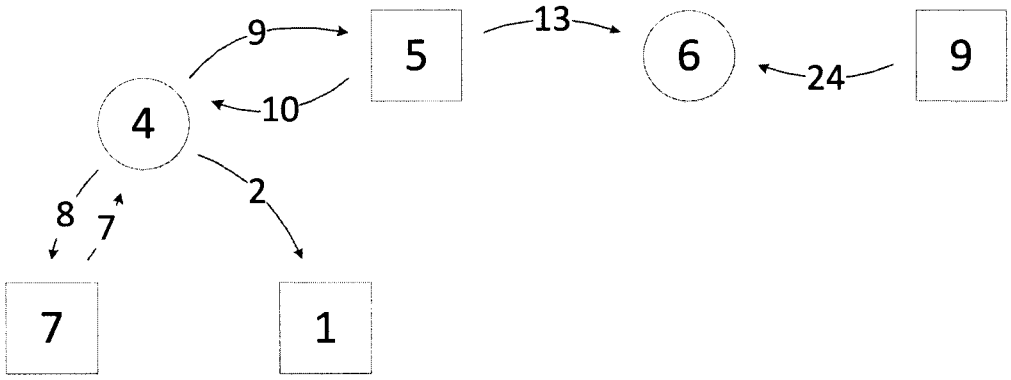

Figure 4 - Example 2 Hidden Network

For Example 3, the hidden network is shown in Figure 5 and it consists of undetermined arcs 7, 10, 13 and 24 and nodes 4, 5, 6, 7 and 9.

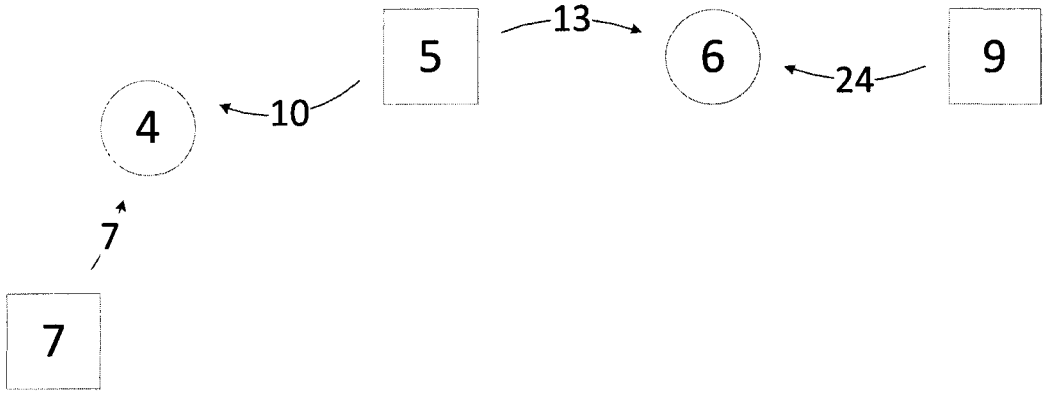

Figure 5 - Example 3 Hidden Network

3.5 Incremental flow and Feasibility Check

In this section, we introduce the concept of ***Incremental Flow***, which is important in reducing the sensor location problem to a network flow problem in the hidden network. Suppose we are given a monitoring set M, and we have constructed the associated hidden network H_M. Then, the flows on the original network are uniquely determined by monitoring nodes in M if and only if there is no non-zero incremental flow in the associated hidden network H_M. Here the **non-zero incremental flow** in H_M has to satisfy two conditions:

Condition 1: flow conservation at all transit nodes in H_M.

Condition 2: incremental flows can take on positive, or zero, or negative values, but the turning factors for all arcs in H_M are strictly positive.

In essence, checking the existence of incremental flows can be done in the following way. Arbitrarily select a node n_0 in the hidden network and assert an incremental flow of $\varepsilon > 0$ on its canonical exit arc. Therefore, the issue is whether that flow can be extended to a feasible incremental flow within the hidden network.

Let δ_n ($n \in N_M$) denote the out-degree of node n, and let $T_M = T \cap N_M$ denote the set of terminal nodes in H_M. Then Rubin (2010) presents the following conjecture.

Conjecture 3.5.1. (Rubin, 2010) A nonzero, conservative incremental flow exists for (a component of) H_M only if

$$\sum_{n \in N_M} \delta_n \geq | N_M \setminus T_M | + 1 \qquad (3.10)$$

Notice that this conjecture uses the degrees of freedom. On the left hand side of the inequality (3.10) is the number of unknowns, whereareas the right hand side of the inequality is the number of equations we could obtain at transit nodes plus the flow 1 we impose on the cannocial exit arc.

In our algorithm, we use conjecture 3.5.1 to perform a feasibility check for a given monitoring set M. In particular, for a monitoring set M, if the inequality (3.7) is violated then we determine that there does not exist a non-zero incremental flow thus the network is uniquely determined. In other words, when inquality (3.7) is not satisfied then we consider the current monitoring set is feasible. We will provide detailed explaination in the next chapter.

CHAPTER IV

A GENETIC ALGORITHM FOR the SENSOR LOCATION PROBLEM

4.1 Constructing Hidden Network

To solve the sensor location problem for large size networks, we choose to use optimization-based methodology. In particular, we employ the concept of hidden network introduced by Rubin (2010). In order to construct the hidden network $H_M=(N_M,A_M)$ for a given monitoring set M, we need an algorithm that essentially processes the network the same way as we have done in Examples 1, 2 and 3 in Chapter 3. Below is a formal description of the algorithm by Rubin (2010).

1. Initialization: set $N_M = N$, $A_M = A$ and $Q = \{(i, j) \in A: i \in M \cup j \in M\}$. Note that Q is a set of arcs to be processed.

2. While $Q \neq \emptyset$:

   a) Select any $(i, j) \in Q$ and remove it from both Q and A_M.

   b) For each $(i, k) \in A_M$ $(k \neq j)$, add (i, k) to Q.

c) For $k \in \{i, j\}$

i. If node k has both in-degree 0 and out-degree 0, remove k from N_M; else

ii. If node k is a transit node with in-degree 1 and out-degree 0, with $(h, k) \in A_M$, add (h, k) to Q; else

iii. If node k is a transit node with in-degree 0 and out-degree ≥ 1, add every $(k, h) \in A_M$ to Q.

One can confirm that when applied to the three examples in Chapter 3, the above algorithm produces the correct hidden networks.

4.2 Genetic Algorithm

As mentioned in Chapter 3, the solution concepts of hidden network and incremental flow reduce the highly combinatorial sensor location problem to a linear integer programming problem. The latter allows for the solution for small size networks via general-purpose software such as CPLEX (2010). However Rubin (2010) reported that when the network size becomes medium to large, the mixed integer formulation of the SLP cannot be solved efficiently. In fact, our experience of solving the associated mixed integer program for the SLP, suggested that CPLEX 11.0 failed for networks with 15 nodes.

Consequently, a heuristic algorithm that provides "good" solutions in reasonable CPU time for medium to large size networks is needed.

In the operations research and optimization literature, the genetic algorithm (GA) has been widely accepted as an efficient heuristic for global optima. In a nutshell, GA imitates the process of natural evolution, and by evaluating, selecting, breeding and filtering within randomly generated candidate solutions to obtain the optimal solution. In general, genetic algorithm has the following basic steps.

1. Initialization

   At the beginning of the genetic algorithm, a certain number of individual solutions are randomly generated and together they comprise the initial population.

2. Evaluation

   After the initial population is created, a fitness score will be used to rate each individual solution within the population. Such a fitness score should reflect: 1. if the solution satisfies the constraints of the optimization problem under consideration; 2. how well the solution optimizes the objective function.

3. Selection

Once the initial population is created, we have to determine which individuals, or, solutions, to be selected as parents to reproduce offsprings, i.e., new solutions. Inspired by human evolution, one would make those solutions with good fitness scores have higher probability to be selected for reproduction. Done this way, individuals, or solutions, from one generation to next, are likely to improve (in terms of optimizing the objective value). Ideally, when enough generations are carried over, the best solutions of the last generation should converge to the optimal solution.

In short, genetic algorithm often assigns those solutions with good fitness score larger probability of being selected for reproduction, and those with poor fitness score smaller probability. Note that in order to promote diversity for reaching global optima, GA usually does not assign zero probability to individuals with poor fitness scores.

4. Reproduction

Once parents are selected, the next step is to generate offsprings, i.e., new solutions, to form the next generation. The most popular methods of creating

offsprings are crossover and mutation. The crossover process typically takes a set of alleles from one of the parents, and then switching them with the alleles of the other parent. It makes a new solution share many characteristics of its parents and gathers them into one better solution as the offspring. Mutation on the other hand simply change some alleles of one of the parents. The goal of mutation is to increase diversity of the population, which can effectively avoid trapping at local optima.

In many GA practices, it is not necessary to eliminate both parents once they reproduce offsprings. The ***elitism strategy*** always keep certain percentage of the best solutions within current generation to be carried over to the next generation.

5. Termination

In the genetic algorithm, the process will be repeated until the termination conditions the user sets are met. Below are some common terminating conditions.

a) A solution that satisfies some minimum criteria is found.

b) A fixed number of generations is reached.

c) A fixed amount of computation time is reached.

d) Successive iterations no longer produce better results.

4.3 A Genetic Algorithm for the Sensor Location Problem

In this section we introduce our specific genetic algorithm in Matlab for the SLP problem.

First, we construct a function to randomly generate a symmetric network with designated set of terminal nodes. In particular, we randomly create a node-arc incidence matrix representing a symmetric network, and randomly create a binary vector representing the terminal-node vector for the network created. Note that such randomly created NAIM should satisfy the four properties discussed in Section 3.3.

Next we will customize GA for the sensor location problem.

1. Solution encoding:

   A binary vector of length |N| represents a solution, where x_j=1 if node j is monitored, 0 otherwise. For example, the solution for example 1 is $M_1 = (0\ \ 0\ \ 0\ \ 1\ \ 0\ \ 0\ \ 0\ \ 0)$.

2. Initialization:

   Given a randomly generated the diagraph (NAIM, T), we then randomly generate P binary vectors of size |N|, each

representing a solution to the SLP. Herein, P ($P<<2^{|N|}$) is referred to as the generation size. These P solutions constitute the initial population.

In our GA, everytime a solution x is created we check whether it is indeed a feasible solution, i.e., the monitoring set M associated with solution x will uniquely determine the flows on the entire network. Particularly, we use conjecture 3.5.1 for this feasibility check. If inequality (3.7) is violated, then it is suggested that there is no non-zero incremental flow in the hidden network. Thus, the flows on the entire network are uniquely determined. If inequality (3.7) holds, we simply replace the infeasible solution(s) with new randomly generated solution(s) and check the feasibility again. This process repeats until all initial solutions are feasible.

3. Evaluation:

Once initial solutions are obtained, we need to evaluate their fitness scores. These fitness scores are used in ranking all solutions within the population. Because the objective is to minimize the total number of sensors, we first rank the solutions in ascending order based on the number of sensors required in the solution, i.e., $s_i = \sum x_i$. Then we assign the the i-th ranked solution a fitness score (f_i) is calculated as follows:

$$f_i = \frac{10(P-i)}{P-1}, \qquad (3.11)$$

where P is the population size. One can see that (3.10) ensures that the first-ranked solution has a fitness score of 10, and the last-ranked solution has a fitness score of 0. If several solutions are tied in the ranking based on s, then all of them will be counted as one individual and ranked together. Consequently, they will have the same fitness score.

4. Selection:

We employ the roulette wheel selection method to choose solutions as parents to mate and produce offsprings. The roulette wheel selection method probabilistically selects individuals according to their fitness scores. The probability of an individual being selected (F_i) is given by following equation:

$$F_i = \frac{f_i}{\sum_{i=1}^{P} f_i}, \qquad (3.12)$$

where f_i is the fitness score of the i-th ranked individual. Clearly, equation (3.11) ensures that solutions with higher fitness scores have higher probability to be selected as parents. Intuitively, this is desired because these

solutions tend to produce better offsprings. Furthermore, it should be noted that the roulette wheel selection method assigns low but not zero probability for solutions with low fitness score to be selected to breed. This is important to diversify among solutions and help the algorithm to reach global optima.

5. Crossover:

After parents are selected, we apply single point crossover to each pair of parents. In our algorithm, we make the crossover happen at each allele with equal probability. Once the crossover point is selected on both parents' strings, and all alleles after that point in either parent is swapped each other. Figure 6 illustrates this basic type of crossover.

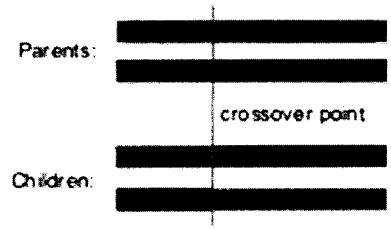

Figure 6 - Single Point Crossover

The single point crossover process may have three results: 1. it creates two new offsprings; 2. it creates one new offspring and one offspring that is the same as one of the parents; 3. it creates two offsprings that are the same as the parents. To minimize the possibility of cases 2 and 3 , we use the binary "mask" in conjection with the single

point crossover. For example, suppose two parents are X_1 = (1, 0, 1, 0, 0, 1) and X_2 = (0, 0, 1, 1, 0, 0). Then we randomly select one node, say, node 3, and randomly generate a mask, say, (0, 0, 1, 1, 1, 1). The single point crossover process will make two offsprings S_1 and S_2. In creating S_1, if $mask_i = 0$, then $S_1(i)$ follows parent X_1, i.e., $S_1(i) = X_1(i)$. On the other hand, if $mask_i = 1$, then $S_1(i)$ follows parent X_2, i.e., $S_1(i) = X_2(i)$. Therefore, S_1=(1, 0, 1, 1, 0, 0). Similarly, in creating offspring S_2, if $mask_i = 0$, then $S_1(i)$ follows parent X_2, i.e., $S_1(i) = X_2(i)$. On the other hand, if $mask_i = 1$, then $S_1(i)$ follows parent X_1, i.e., $S_1(i) = X_1(i)$. Then S_2=(0, 0, 1, 0, 0, 1).

6. Mutation:

When offsprings are created from parents, and we apply mutation on them in order to increase diversity within the population and to ultimiately reach global optima. Each allele of each new solution has the equal probability to mutate. In particular, we randomly select a node j, and change its current X_j from 0 to 1 or from 1 to 0.

From the crossover and mutation processes, it is likely that the new solution is infeasible. So checking feasibility again is necessary. Once a solution is found to be infeasible, we replace it with a new (randomly generated) solution. This indeed is similar to the *immigration*

strategy in the GA literature.

7. Elitism:

Once offsprings are created and mutation is operated, we again rank the new generation. The bottom 10% individuals of this new generation is replaced with the top 10% individuals of the previous generation. This is an exercise of the so-called elitism strategy in GA.

4.4 Several Improvements

In this section, we offer some preliminary thoughts on further improvements of the GA described in Section 4.3.

The first improvement is a screening procedure for any feasible solution. The idea is to use shortest path algorithm to calculate the distance between any two nodes, and to make sure that sensors are placed at nodes that are at least 2 units away from each other. In other words, we would eliminate any solutions that suggest to place sensors at adjacent nodes. Below is the outline of this screening process.

A Distance-based Screening for Feasible Solutions

For each binary solution X,

```
for i = 1 to n
   if X_i = 1 then
      for j = 1 to n
         if (j ≠ i) and (X_j = 1) then
            if distance(i, j) = 1
            then X_j = 0
            endif
         endif
      endfor
   endif
endfor
```

Again, here distance(i, j) is the shortest path between node i and j. (we assign travel cost between any two nodes to be 1.) For instance, in Example 3, distance(1, 2) = 1 and distance(1, 5) = 2 (the shortest path uses links 1 and 9 instead of links 3, 18 and 14).

The second improvement is a neighborhood search process to

convert an infeasible solution to a feasible solution. In particularly, we add a sensor at the node with the highest outdegree. We call this a insertion type of neighborhood search. Below is the outline of this search process.

A Neighborhood Search for Converting infeasible to feasible solutions

Step 1: choose the terminal node j with the highest outdegree in the Hidden Network, let $X_j = 1$ and update the solution X.

Step 2: Update the Hidden Network with new vector X.

Step 3: Run the conjecture with the updated X and obtain the new Hidden Network. If the X is feasible, then stop and keep the solution X. Otherwise, go to Step 1.

CHAPTER V

NUMERICAL RESULTS

In this chapter, we report numerical results on randomly generated networks for the proposed genetic algorithm for the sensor location problem. We note that the present thesis only implements the customized GA for the SLP introduced in Section 4.3. The further advancements discussed in Section 4.4 are to be implemented and investigated in future research.

As discussed in Section 3.3, random (symmetric) networks are generated via the construction of the associated node-arc incidence matrices that satisfy four properties. In order to study the scalability of the algorithm we vary the network size in several ways. First, we change the number of nodes in a network. Secondly, we change the arc density (in percentage) in a network. Particularly, if a network has a fixed number of nodes, say n, then the maximum number of arcs is $n \times (n-1)$. Then, for example, a 20% arc density calls for $0.2n \times (n-1)$ arcs within the network. Thirdly, we change the percentage of terminal nodes in the network.

In testing the proposed GA, our evaluation of the method

consists of two parts. One is the computational time in CPU seconds, the other is the convergence quality at the termination. Particularly we calculate the percentage of the best solution in the final generation and consider that 90% or above is a good convergence quality.

Because genetic algorithms generally have many parameters to fine tune, we study the effect of these parameters on solution time as well as solution quality. The parameters we consider include: 1. generation size, which is the number of generations our GA is allowed to run before termination; 2. population size, which is the number of solutions in each generation; 3. crossover rate, which is the probability of the occurrence of crossover at each allele; 4. mutation rate, which is the probability of the occurrence of mutation at each allele; 5. elitism rate, which is the percentage of top ranked solutions that are automatically kept from one generation to next.

Finally, the genetic algorithm proposed in Section 4.3 was programmed using Matlab, and the CPU time reported herein were from a computer platform with a 2.0 GHZ Intel Core 2 Duo CPU and 2GB of RAM.

5.1 Effects of Network Size and GA Parameters on the CPU Time

In this section, we first study how GA parameters including population size and generation size affect the solution time of the proposed GA. Secondly, we evaluate how the network configuration including the number of nodes, the arc density and the percentage of terminal nodes, affects the solution time.

To ease presentation, we use the following notation throughout the chapter.

P=population size for GA; G=generation size for GA

N=number of nodes in the network

AD=arc density in the network

PT=percentage of terminal nodes in the network

Table 1

CPU Time for Variable Population Size

Example	Population Size				
	50	100	200	500	1000
1	3.432	7.3944	15.039	37.955	79.67
2	3.5412	6.864	14.071	37.035	75.052
3	3.588	6.6768	14.212	36.473	75.317
4	3.8532	8.0653	15.85	40.56	82.197
5	3.7752	7.4412	15.538	39.64	80.668
6	3.5412	7.3476	13.962	37.003	72.447
Average	3.62	7.30	14.78	38.11	77.56

Table 1 displays the CPU time for various population sizes 50, 100, 200, 500 and 1000, with other parameters fixed as

G=50, N=15, AD=20% and PT=20%. For each population size, we run six instances and report the individual as well as the average CPU times. From the Table 1, we observe that the average CPU time is 3.62 seconds for the population size of 50, and it increases to 7.3 seconds with the population size doubles. When the population size is set to be the highest 1000, the average CPU time amounts to 77.56 seconds, slightly over 1 minute.

In summary, Figure 6 plots the relation between the population size and the CPU time. It clearly shows that the average CPU time increases with the population size, approximately, in a linear fashion.

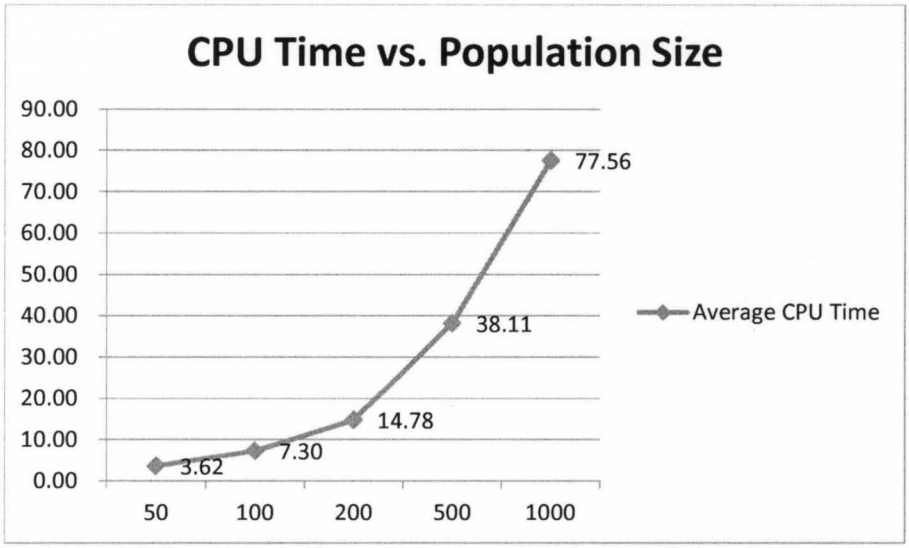

Figure 6 - CPU Time for Variable Population Size

Table 2

CPU Time for Variable Generation Size

Example	Generation Size					
	50	100	200	300	500	1000
1	7.3944	14.649	27.16	43.602	69.888	132.835
2	6.864	13.635	26.879	40.264	67.19	133.178
3	6.6768	13.276	27.129	40.467	65.988	132.679
4	8.0653	15.132	30.093	44.32	74.834	153.552
5	7.4412	13.931	26.629	40.888	68.406	138.903
6	7.3476	13.635	26.395	43.899	73.009	139.949
Average	7.30	14.04	27.38	42.24	69.89	138.52

Table 2 displays the CPU time for various generation sizes: 50, 100, 200, 300, 500 and 100, with other parameters fixed as P=100, N=15, AD=20% and PT=20%. As in the first test, we also run 6 instances for each generation size, and calculate the average CPU time. From Table 2, we can see that the average CPU time increases from 7.3 seconds to 69.89 seconds and then to 138.52 seconds (over 2 minutes) as the generation size increases from 50 to 500 to 1000. Furthermore, Figure 7 below shows that not only the CPU time increases with the generation size, but it increases at a more rapid rate than it does with the population size.

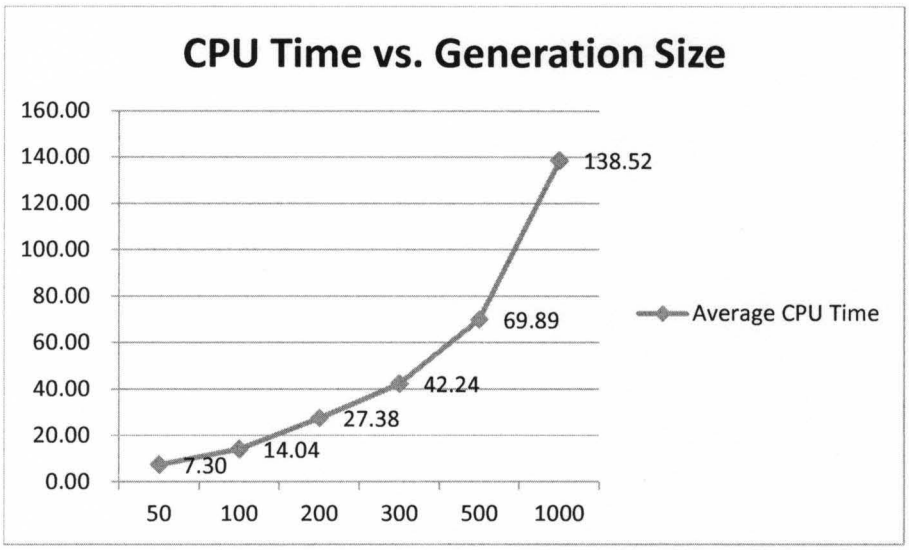

Figure 7 - CPU Time for Variable Generation Size

Next we study the effect of network size, i.e., the number of nodes on the CPU time. Table 3 displays the CPU time at various network sizes: 10, 15, 20 and 30, with other parameters fixed as P=100, G=50, AD=20% and PT=20%. From the Table 3, we observe that the CPU time increases sharply from 7.3 seconds to 166.56 seconds (nearly 3 minutes) to 2028.25 seconds (more than 30 minutes) when the number of nodes increases from 15 to 30 to 50.

Table 3

CPU Time for Variable Network Size

Example	Network Size				
	10	15	20	30	50
1	1.6536	7.3944	24.695	168.7775	1.98E+03
2	1.1856	6.864	24.258	164.0039	2.08E+03
3	1.4664	6.6768	27.503	168.5591	2.00E+03
4	1.2948	8.0653	24.757	168.5279	2.01E+03
5	1.4664	7.4412	24.617	168.9491	2.06E+03
6	1.3728	7.3476	23.977	160.5406	2.03E+03
Average	1.41	7.30	24.97	166.56	2028.25

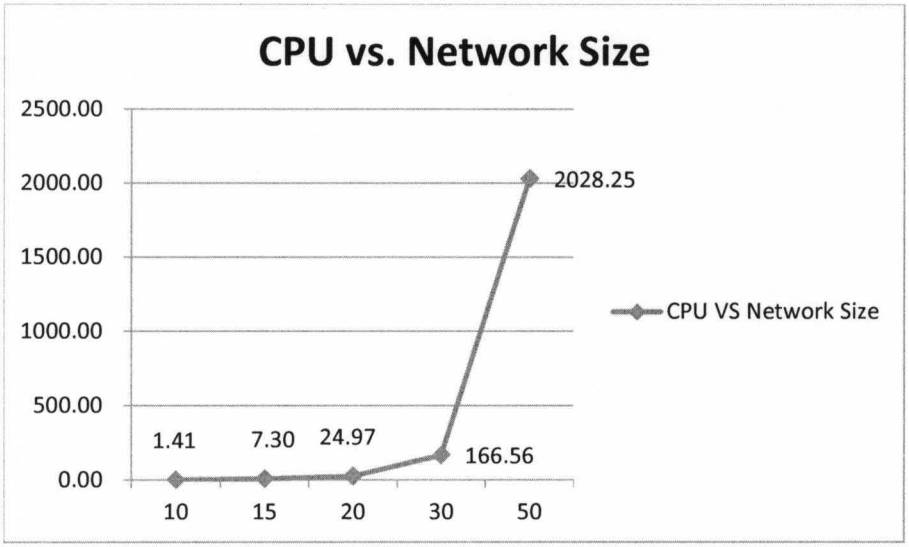

Figure 8 - CPU Time for Variable Network Size

Additionally, Figure 8 shows that effect of the network size on the CPU time is significantly increased when the number of nodes exceeds 30.

Table 4 displays the CPU time for variable arc densities 10%, 15%, 20%, 30% and 50%, with other parameters fixed as P=100%, G=50%, N=15 and PT=20%. Clearly, Table 4 shows that increase in the number of arcs leads to increase in the CPU time. But comparing Tables 3 and 4, one can see that the effect of the number of arcs or arc density on the CPU time is not as strong as that of the number of nodes. The comparison between Figures 8 and 9 confirms this observation.

Table 4

CPU Time for Variable Arc Density

Example	Arc Density(%)				
	10%	15%	20%	30%	50%
1	2.9172	4.4148	7.3944	14.181	33.774
2	2.496	4.0872	6.864	14.961	38.173
3	3.1356	4.2744	6.6768	13.151	35.911
4	2.886	4.4616	8.0653	14.633	38.111
5	3.2604	5.0388	7.4412	14.914	38.283
6	2.6208	4.602	7.3476	13.962	37.409
Average	2.89	4.48	7.30	14.30	36.94

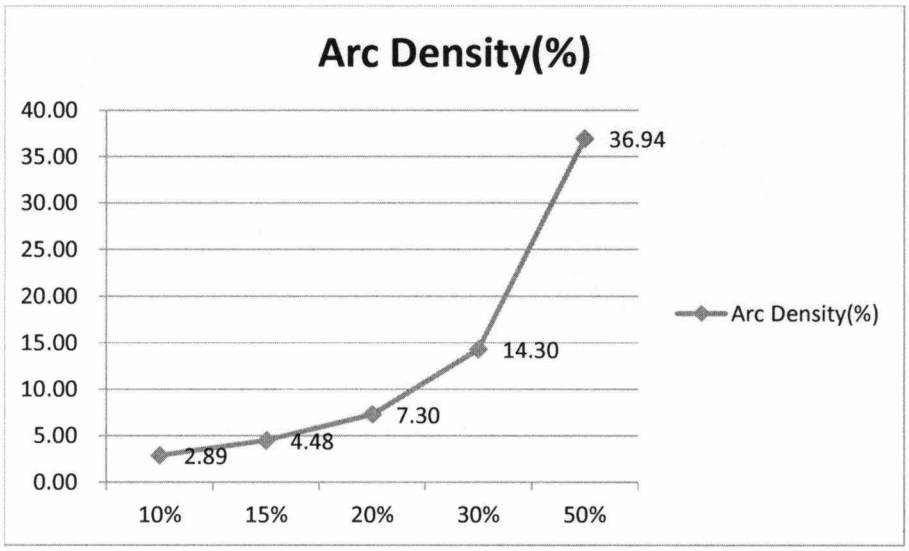

Figure 9 - CPU Time for Variable Arc Density

Finally, Table 5 and Figure 10 collectively show that the percentage of terminal nodes in the network has little effect on the CPU time. This is a bit surprising, as the number of terminal nodes generally affects the size of the hidden network. However, it is envisioned that the percentage of terminal nodes will affect the solution quality of the proposed GA.

Table 5

CPU Time for Variable Terminal Node Density

Example	Terminal Node Density (%)				
	10%	15%	20%	30%	50%
1	6.9888	6.6612	7.3944	6.942	6.6144
2	7.1136	7.2696	6.864	7.176	6.7704
3	6.7548	6.708	6.6768	6.9888	6.4584
4	7.2228	6.7392	8.0653	6.3804	6.4584
5	7.1916	7.0356	7.4412	6.6456	7.6752
6	6.864	6.9732	7.3476	6.8016	6.162
Average	7.02	6.90	7.30	6.82	6.69

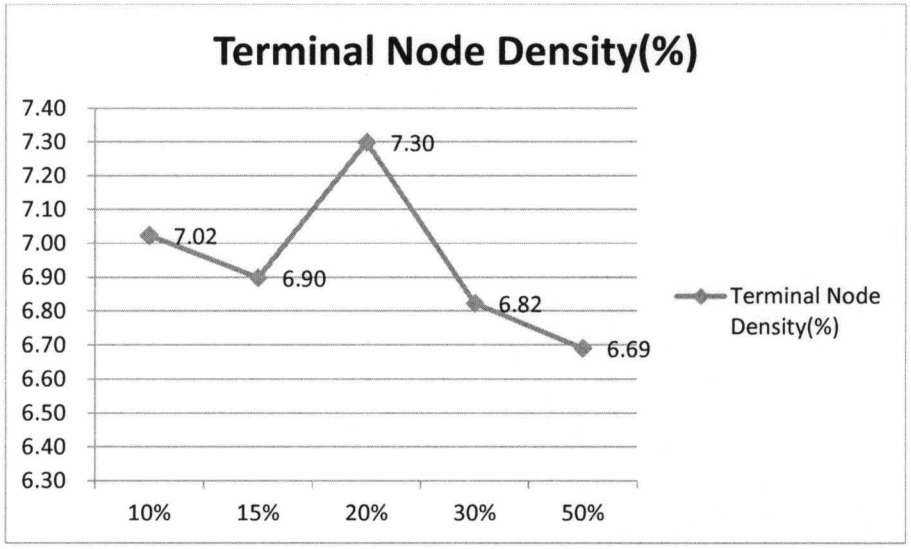

Figure 10 - CPU Time for Variable Terminal Node Density

5.2 Effects of Network Size and GA Parameters on Solution Quality

In this section, we will study the quality of our genetic algorithm. Particularly, we use the percentage of the "best solution" in the final population. We consider the GA converges well if such percentage is 90% or above. Similar to Section 5.1, we vary GA parameters including generation size, population size, mutation rate and etilism rate, and see how they affect the solution quality. In addition, we vary network size including the number of nodes, arc density and terminal percentage to see their respective effects on the solution quality.

Table 6

Quality for Variable Generation Size = 50

G = 50	Network Size				
	10	15	20	30	50
1	100%	100%	100%	95%	69%
2	100%	100%	100%	100%	78%
3	100%	100%	100%	100%	72%
4	100%	100%	98%	100%	71%
5	100%	100%	100%	100%	83%
6	100%	99%	100%	100%	77%
Average	100%	99.83%	99.67%	99.17%	75%

Table 6 shows the average solution quality for various network sizes: N=10, N=15, N=20, N=30 and N=50, when other parameters are fixed as G=50, P=100, AD=20% and PT=20%. From Table 6, one can see that the proposed GA converges well for network size of 30 or smaller with an average percentage of "best solution" (in the final population) being approximately 100%. When the network size becomes 50, the average percentage of "best solution" drops to 75%. This suggest that we may need to increase the generation size in order to achieve better convergence. Figure 11 depicts how the network size affect the percentage of "best solutions" in the final population.

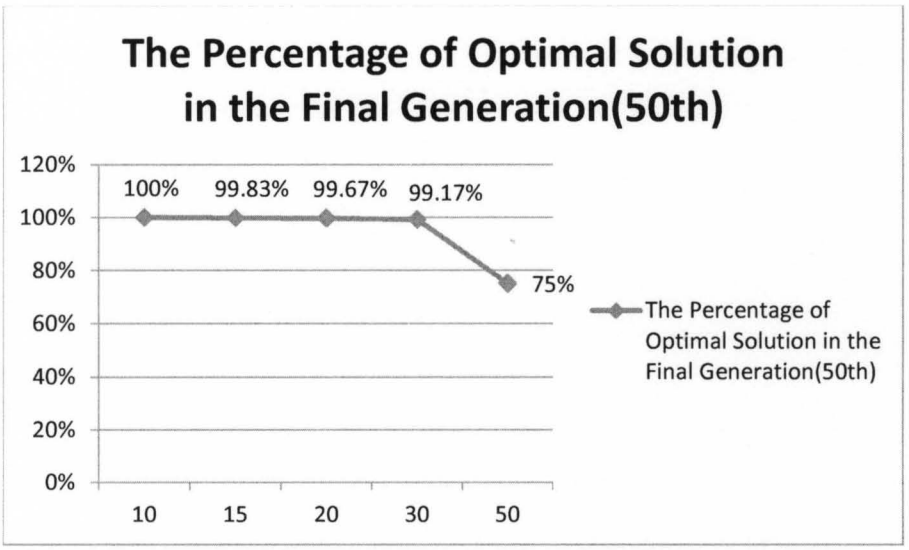

Figure 11 - Quality for Variable Generation size = 50

In order to investigate if increasing generation size helps achieve better convergence, we repeat the same experient as done in Table 6 except that the generation size is 100. The results are presented in Table 7 and Figure 12. Interestingly, the average percentage of "best solution" for the network size 50 drops, instead of increases, slightly from 75% to 73.67%. Furthermore, the average percentages for other networks sizes are the same or slightly increased. Comparing Tables 6 and 7 suggests that for larger networks, simply running our GA for more generations may not be helpful to achieve global solutions. More work is needed to improve the algorithm itself.

Table 7

Quality for Variable Generation Size = 100

G = 100	Network Size				
	10	15	20	30	50
1	100%	100%	100%	100%	65%
2	100%	100%	100%	96%	70%
3	100%	100%	100%	96%	76%
4	100%	100%	99%	100%	83%
5	100%	100%	100%	100%	69%
6	100%	100%	100%	100%	79%
Average	100%	100%	99.83%	98.67%	73.67%

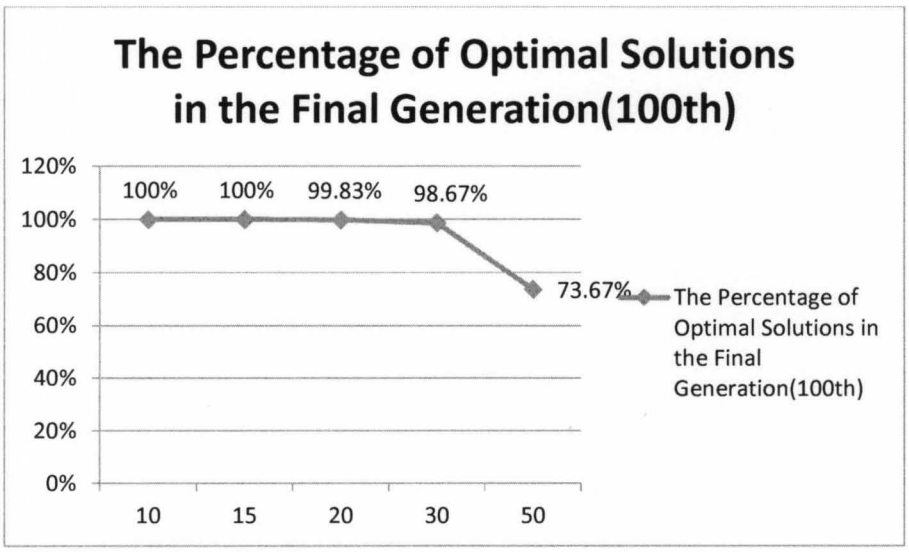

Figure 12 - Quality for Variable Generation Size = 100

Tables 8 and 9 are to investigate if increasing population size helps achieve a better convergence. In particular, Table 8 displays the average percentages "best solutions" for various network sizes when the population size is 50, while Table 9 displays similar information when the population size is 100. Both tables are parameterized as G=50, AD=20% and PT=20%. Study Tables 8 and 9 (or Figures 13 and 14) collectively, we observe the following. First, the proposed GA can solve

small network sizes 10, 15, 20 and 30 with an average percentage of "best solution" being 90% or higher. Second, network size 50 still poses a difficult instance of SLP. The proposed GA (P=50) only has 67% percent of the solutions in the final population that are the "best" incumbent. Third, as the population size increases from 50 (Table 8) to 100 (Table 9), this percentage increases from 67% to 75%. This suggests that increasing the population size is more effective in achieving better convergence than increasing the generation size.

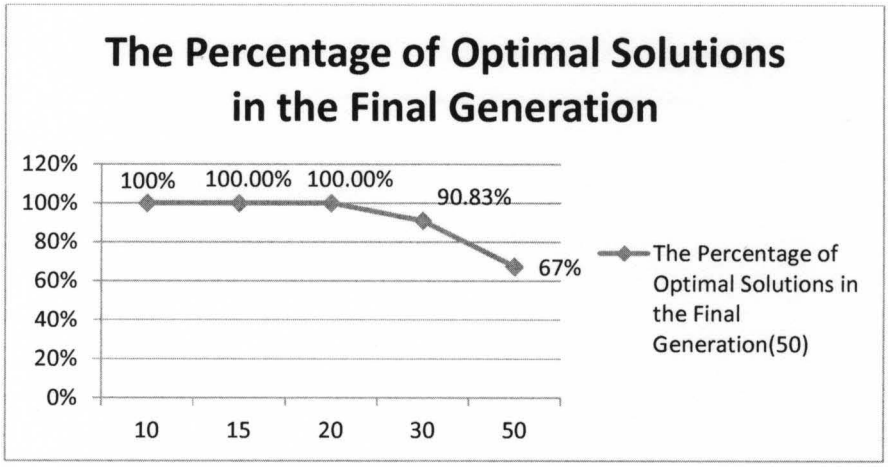

Figure 13 - Quality for Variable Population Size

Table 8

Quality for Variable Population Size = 100

P = 100	Network Size				
	10	15	20	30	50
1	100%	100%	100%	95%	69%
2	100%	100%	100%	100%	78%
3	100%	100%	100%	100%	72%
4	100%	100%	98%	100%	71%
5	100%	100%	100%	100%	83%
6	100%	99%	100%	100%	77%
Average	100%	99.83%	99.67%	99.17%	75%

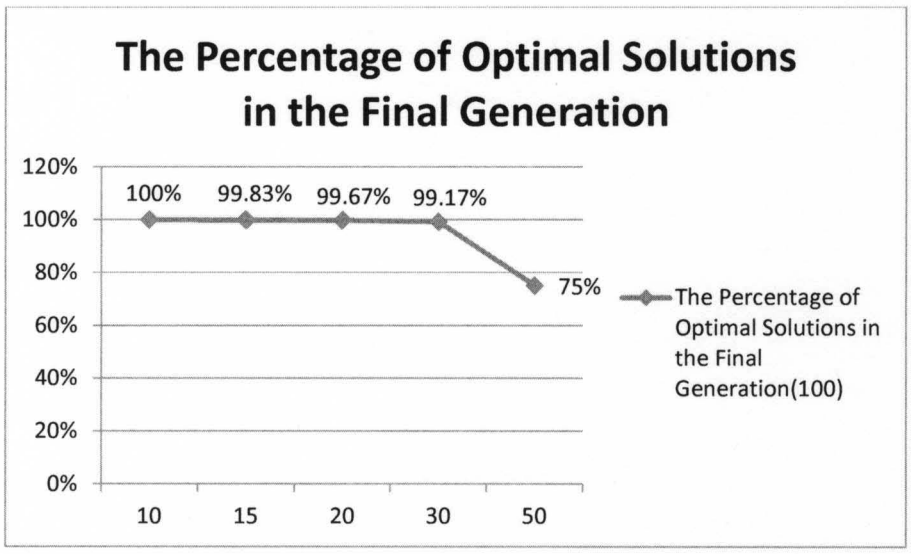

Figure 14 - Quality for Variable Population Size = 100

Table 10 studies how mutation rate affects the convergence of the proposed GA. Particularly, the table gives the average percentage of "best solution" for various mutation rates 2%, 2.5%, 3%, 3.5% and 4%. As can be seen in Table 10 and Figure 15, the convergence decreases from 99.17% to 81% to 66.17% as the mutation rate increases from 2% to 3% to 4%. It suggests that 2% mutation rate is a good choice.

Table 9

Quality for Variable Mutation Rate

Example	Mutation Rate				
	2%	2.50%	3%	3.50%	4%
1	95%	78%	72%	74%	67%
2	100%	93%	75%	67%	59%
3	100%	93%	87%	72%	74%
4	100%	89%	80%	73%	61%
5	100%	88%	93%	85%	67%
6	100%	86%	79%	70%	69%
Average	99.17%	88%	81%	73.50%	66.17%

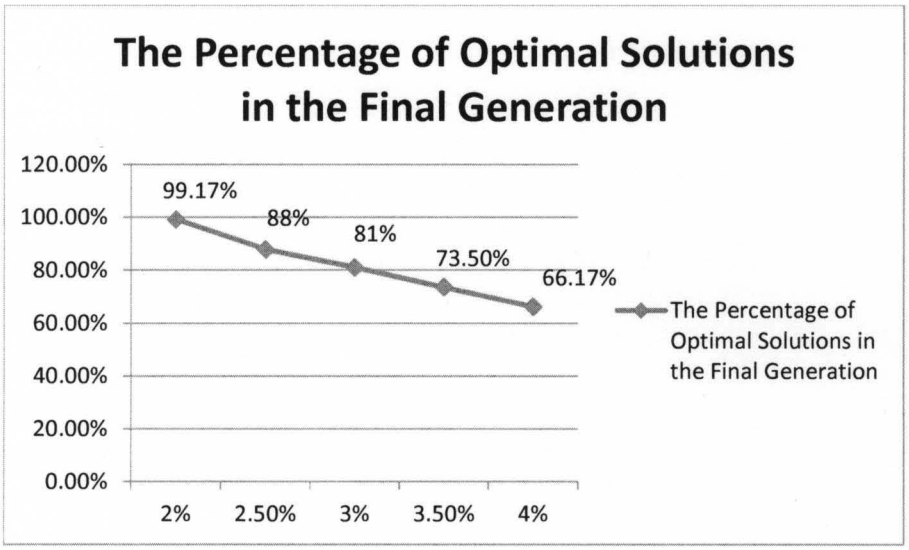

Figure 15 - Quality for Variable Mutation Rate

On the other hand, Table 11 and Figure 16 show that crossover rate does not have significant impact on the convergence of the proposed GA. We recommend to use crossover rate of 70% when implementing the proposed GA.

Table 10

Quality for Variable Crossover Rate

Example	Crossover Rate			
	30%	40%	50%	70%
1	96%	100%	99%	95%
2	98%	100%	100%	100%
3	100%	100%	100%	100%
4	98%	100%	100%	100%
5	91%	92%	98%	100%
6	97%	100%	97%	100%
Average	96.67%	98.67%	99.00%	99.17%

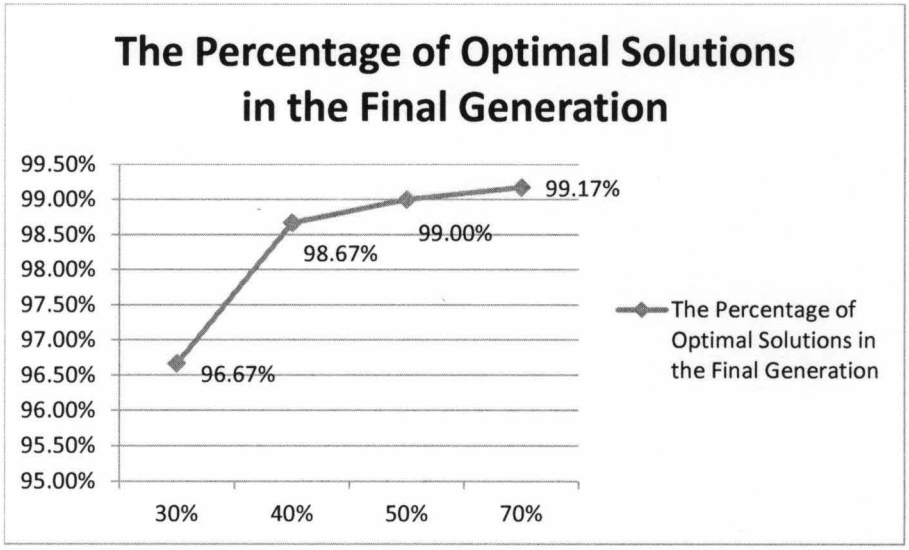

Figure 16 - Quality for Variable Crossover Rate

Finally, we study how the elitism rate affects the quality. Table 12 shows that the average percentage of "best solution" is about 74% when the elitism rate is 5%. When we increase the elitism rate from 5% to 8% and 10%, the average percentage increases to 100%.

Table 11

Quality for Variable Elitism Rate

Example	Elitism Rate		
	5%	8%	10%
1	75%	100%	100%
2	71%	100%	100%
3	69%	100%	100%
4	49%	100%	100%
5	80%	100%	100%
6	100%	100%	100%
Average	74.00%	100.00%	100.00%

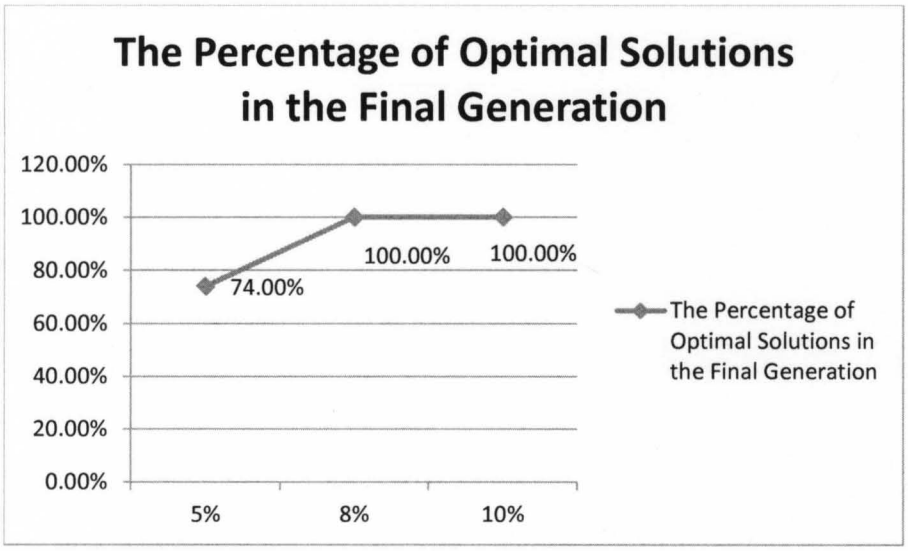

Figure 17 - Quality for Variable Elitism Rate

CHAPTER VI

CONCLUSIONS AND FUTURE RESEARCH

We study a sensor location problem that minimizes the total number of sensors to employ at road intersections in a transportation network so that the traffic flows on the entire network are uniquely deduced. The problem is highly combinatorial in nature, we propose a network flow based approach for its solution. Adopting the ideas of "hidden network" and "incremental flow" in Rubin (2010), we develop a genetic algorithm to solve large-scale sensor location problems.

Our numerical results suggest the following conclusions. First, the network size, i.e., the number of nodes in the network, has the most significant impact on the solution time of the proposed GA. Emprically, we observe that network size of 50 requires approximately 30 minutes of CPU time.

Second, the arc density has some limited effect on the solution time. On one hand, the computational time increases when the arc density changes from 10% to 50% due to the increase of the network size. On the other hand, as the number of arcs

increases, the number of arcs adjacent to monitored nodes also increases. The latter implies that we would have more knowledge on the network flows, which may help us to deduce the flows on the rest of the network. Consequently, the sensor location problem becomes easier and requires less CPU time.

Third, the terminal node density has little impact on the solution of the GA. It is our speculation that the distribution of the terminal nodes would have more effect on the CPU time than does the number of terminal nodes.

Finally, in terms of recommendations of the GA parameters, we see a tradeoff between the solution quality and the CPU time. Our numerical results suggest that for small to medium size networks with 30 or fewer nodes, population size of 50 and generation size of 50 work well. For larger networks with 50 or more nodes, we recommend to use population size of 100 and generation size of 50. Furthermore, in all cases, we suggest to use crossover rate of 70%, mutation rate of 2% and elitism rate of 10%.

Overall, the proposed based genetic algorithm is efficient at solving the sensor location problem. On average, for networks with 20 nodes, it can be solved in less than half minute, with 30 nodes in less than 3 minutes, and with 50 nodes in less than 30 minutes. For networks with 30 or fewer nodes,

more than 90% of the final solutions are the "best" incumbent solution found by the solver. For networks with 50 nodes, the solution quality drops to as low as 70%.

Several improvements can be made in future research. First, in Section 4.4 we propose a "distance-based" method for additional screening of any feasible solution that will make sure sensors are spreaded out from each other. However, this has yet been implemented in the GA. We would like to investigate how this screening process helps to expedite the algorithm. Second, a neighborhood search method is also proposed in Section 4.4 to convert an infeasible solution to a feasible solution by adding sensors at proper locations. This has not been implemented in the current GA either. We plan to investigate the benefit of such neighborhood search in future works.

Third, our current approach checkes the feasibility of a candidate solution through a conjecture in Rubin (2010). We would like to prove it theoretically. Fourth, we also like to investigate a network-flow based approach to determine if there exists non-zero "incremental flow" in the hidden network. In particular, we would like to check if one can push a unit flow between any two selected terminal nodes in the hidden network without violating the requirement of positive turning factors. This may be related to path identification methods

in network flows. Finally, another related problem that is relevant in large (telecommunication) networks is to place a given number of sensors appropriately on the network so that the information gained is maximized.

REFERENCES

Bianco L., Confessore G., & Reverberi P..(2001). A Network Based Model for Traffic Sensor Location with Implications on O/D Matrix Estimates. Transportation Science, 35, 1, 50-60.

Bianco L., Confessore G., & Gentili M..(2006). Combinatorial Aspects of the Sensor Location Problem. Annals of Operation Research 144, 1, 201-234.

Bianco L., Cerulli R., & Gentili M.. New Resolution Approaches for the Sensor Location Problem.

Carter B., Ragade R.. A Probabilistic Model for the Deployment of Sensors. Louisville, KY: University of Louisville.

Confessore G., Paolo D., & Monica G..(2005). Experimental Evaluation of Approximation and Heuristic Algorithms for the Dominating Paths Problem. Computers & Operation Research, 32, 9, 2383-2405.

CPLEX (2010). ILOG CPLEX Optimization, Inc., Incline Village, NV.

Morrison R.D.(2008). Characteristics of Optimal Solutions to the Sensor Location Problem. Master's thesis. Harvey Mudd College.

Rubin P.A.. (2010). Private communiation.

Schrank D., Lomax T.. (2007). Urban Mobility Report. Texas Transportation Institute, September 2007. http://mobility.tamu.edu.

Yang H., Iida Y., & Sasaki T.. (1991). An Analysis of the Reliability of an Origin/Destination Trip Matrix Estimated from Traffic Counts. Transportation Research, 25(B), 351-363.

Yang H., Zhou J..(1998). Optimal Traffic Counting Location For Origin-Destination Matrix Estimation. Transportation Research, 32, 2, 109-126.

CURRICULUM VITAE

Name: Di Zhang

Address: Department of Industrial Engineering
J.B. Speed School of Engineering
University of Louisville
Louisville,KY 40292

Education: Bachelor of Engineering, Electric Engineering
Beijing Hang University
2005-2009

Work Experience:
Teaching Assistant
IndustrialEngineering,University of
Louisville, 2010-2011

Professional Societies and Memberships:
Institute of Industrial Engineers