KWAME NKRUMAH UNIVERSITY OF SCIENCE AND TECHNOLOGY, KUMASI

SHORTEST PATH ALGORITHM FOR TRANSPORTATION NETWORKS

BY

NATHANIEL FRIMPONG DARQUAH

A Thesis Submitted to the Department of Mathematics,

Kwame Nkrumah University of Science and Technology, Kumasi

in partial fulfillment of the requirement for the degree

Of

MASTER OF SCIENCE

Department of Mathematics

Faculty of Physical Sciences

College of Science

August, 2009

DECLARATION

I hereby declare that this submission is my own work towards the Master of Science degree and that, to the best of my knowledge it contains neither material previously published by another person nor material which has been accepted for award of any other degree of the university

Nathaniel Frimpong Darquah, PG9671306	………………..	….……………….
Student Name & ID	Signature	Date

Certified By

Dr. S. K. Amponsah	…………………….	……………………
Supervisor	Signature	Date

Certified By

Dr. S. K. Amponsah	…………………….	……..……………
Head of Department	Signature	Date

ACKNOWLEDGEMENTS

First and most, I express my sincere thanks to the almighty God for seeing me through my education and again for granting me the wisdom, knowledge and strength throughout this thesis write – up.

There are many people whom I would like to thank for their support and encouragement during the process of writing this thesis.

I wish to thank my supervisor, Dr S. K Amponsah for his numerous insightful comments, encouragement which helped me improve the quality of the thesis and for his motivation and support. He gave me countless advices and opinions that enabled me to finish this thesis.

I also would like to thank Dr M. Astante, Dr J. B Acquah and Mr. I J Arthur for their valuable comments, which improved my thesis significantly.

I owe an enormous debt of gratitude towards others who have helped make this thesis possible: Mr. M Bosu for helping revising the draft of this work and Mr. Steve Badu for his programming contribution.

My special thanks go to my wife Doris and Child Nana Adowa Frimpong Darquah for their patience and encouragement throughout my work.

DEDICATION

Dedication of this work is jointly shared by

1. Wife Mrs. Doris Frimpong Darquah and my Daughter Nana Adowa Frimpong Darquah. God richly bless them.

2. Dr and Mrs. Amponsah for allowing themselves to be used greatly by the God to make me whoever I am now. God richly Bless them.

Abstract

In a metropolis such as Kumasi, the transport network is massive, dynamic, and complicated, and therefore route finding is not an easy task, especially with routes comprising several modes of transport vehicles. This problem is even more important for e-tourism planners and users, security services and moving workforces, who may need to visit an unfamiliar part of the metropolis. To find a route that is the most cost-effective is even harder and time-consuming.

Traffic congestion is becoming a serious environmental threat that must be resolved quickly. Traditionally, travel information systems have been specific to a particular mode of transport. For instance, traffic information (road conditions broadcast) has been directed at drivers. Instead, travel information systems are now being developed, which incorporate route guidance systems to divert drivers away from the congested areas either by change of travel mode or travel route. The mobile travel information system developed at the Kumasi Metropolitan Assembly (KMA) enables progression from a passive mode of interaction between traffic control systems and road-users (one-way flow of information) to an active mode. The integration of data concerning traffic flows and individual journey plans thus makes it possible to perform optimisation of travel. This project focuses on the issue of provision of real-time information about urban travel and assistance with planning travel. KMA traffic-light control system provides real-time information about the traveler distance (time) within certain areas of the city. However, rather than using link travel times at the time of the request, it is more effective to predict the link travel times for the time of travel along the particular links.

The future link travel times depend upon the historical travel time of the link (for the specific time step in the day) as well as the current link travel time. Consequently, the link weights are a combination of real-time data, historical data and static data. The prediction method will be validated in the context of Kumasi urban road network.

Table of Contents

CHAPTER 1

1.0 INTRODUCTION

Traveling is a part of daily life. The majority of people (especially in large cities or developing countries) rely heavily on public transportation instead of their own vehicles. In a metropolis with a complicated transport network, people often do not know how to reach their destination except where they often visit. In addition, people may want to plan for the fastest or the most economical method to their destinations. Such tasks require a sophisticated knowledge about public transport network. Further, we need a multi-modal route finding system, because a transport network comprises many modes of transportation, including railway, bus, mini-bus, and so on, within a large metropolis such as Kumasi. When a user asks for a path from one place to another, the system can generate routes, in multi-modal or single modal mode, according to input criteria, such as cost, time, or transportation mode.

1.1 Dynamic Traffic routing

In recent decades, road transportation systems have become increasingly complex and congested. Traffic congestion is a serious problem that affects people both economically as well as mentally. Moreover, finding an optimal route in an unknown city can be very difficult even with a map. These issues have given rise to the field of Intelligent Transport System (ITS), with the goal of applying and merging advanced technology to make transportation safer and more efficient by reducing traffic accidents, congestion, air pollution and environmental impact (Ahuja,1993). In working towards this goal, traffic routing is required since the traffic conditions change overtime.

Up-to-date, real-time information about traffic conditions can be collected through surveillance systems. However, the utilization of such information to provide efficient services such as real-time en route guidance still lags behind. The objective of this research is to solve the dynamic routing problem, which guides motor vehicles through the urban road network using the quickest path taking into account the traffic conditions on the roads.

1.2 The Role of Geographic Information Systems (GIS) and Location Based Service (LBS)

Geographic Information Systems (GIS) represent a new paradigm for the organization and design of information systems, the essential aspect of which is the use of location as the basis for structuring the information systems. Transportation is inherently geographic and therefore the application of GIS has relevance to transportation due to the spatially distributed nature of transportation related data, and the need for various types of network level analysis, statistical analysis and spatial analysis. GIS possesses a technology with considerable potential for achieving dramatic gains in efficiency and productivity for a multitude of traditional transportation applications.

The impact of GIS technology in the development of transportation information systems is profound. It completely revolutionizes the decision making process in transportation engineering. As a good example, route guidance and congestion management systems can be most suitably developed in a GIS environment. In this application, GIS is used as a powerful tool for identifying and monitoring congestion in urban areas, and planning optimal routes based on minimum time/distance/cost paths. Its graphical display capabilities allow not only visualization of the different routes but also the sequence in which they are built. This allows the user to understand the logic behind the routing design.

With the expansion and proliferation of Location Base Services (LBS) or road map, location awareness and personal location tracking become important attributes of the mobile communication infrastructure and begin to provide invaluable benefits to business, consumer and government sectors. How to establish low-cost, reliable, and high-quality services is the most important challenge in the LBS area. Navigation is perhaps the most well known function of LBS and Geographic Information Systems for Transportation (GIS-T). It is applied in many land-based transportation applications to revolutionize human lives, such as the Intelligent Vehicles Navigation System (IVNS), which is currently a must-have feature especially in the high-end car market

1.3 The Architecture of Navigation Service

Navigation guidance can be discriminated between decentralized and centralized route guidance. In the former, drives derive their own paths using on-board computers, based on either static road maps on paper, or real-time traffic information provided via air waves (radio) network. However, transportation networks have high costs, limited access, and low connection stability making it expensive to deliver detailed traffic information to all map users. Therefore, it may take a long time to find the destination locally or may even be impossible in some cases. On the other hand, navigation services are often used in time-critical circumstances (e.g. 191 Emergency Service) which require near real-time query response and concise route guidance information to facilitate decision making.

Centralized route guidance relies on Traffic Management Centres (TMC) such some FM stations to answer path queries submitted by drivers. In this case, the Client/Server architecture is employed in order to reduce query response time. A centralized GIS server is used to perform the geo-processing task and return query results instead of providing the entire dataset. The service can provide users turn-by-turn navigation instructions about optimal routes to their desired destinations through text or a map display. It can also alert the driver about problems ahead, such as traffic jams or accidents. To deliver query results to mobile clients within a tolerable latency time, it demands an efficient algorithm to retrieve desired navigation information quickly. Thus, it is able to accommodate large numbers of road users. This thesis, discusses the algorithms that are feasible for centralized route guidance.

1.4 Typical Routing Queries

There are various types of routing queries that may be submitted to the centralized GIS server. To answer the queries, many algorithms have been developed to satisfy the conditions and requirements of these queries. The research for generalizing this document is focused on two typical routing queries. The first query deals with finding the optimal route from the current location to a known destination. The other query allows users to locate the closest facility of a certain category (hotel, hospital, gas station, etc.), in terms of travel distance (time), without knowing the destination explicitly.

- **Routing query for known destination**

  For this query, the mobile client or driver has a definite destination in mind and desires to acquire the optimal route leading to the destination. Since the traffic condition changes continually over time, the optimal route will change during travel whenever up-to-date traffic conditions are provided. For example, when we want to drive from the airport to the KMA office, we can plan the entire optimal route prior to departure according to the current condition of the transportation network. However, it may not be the final optimal route due to frequent changes in the traffic conditions. So, we have to modify our route midway and plan a new path from the current location to the destination based on real-time traffic conditions. This case is more complicated than the conventional dynamic concept because both the traffic conditions and the query point (location of the driver) are dynamic. This type of query is also defined as an en route query since it is submitted while the client is moving.

- **Routing query for unknown destination**

  For this query, drivers may inquire about the location of the closest facility, such as the nearest hotel, hospital or gas station, without knowing the destination in advance. In this case, the closest facility is defined in terms of travel distance (time) within the road network as opposed to travel distance. This query can be classified as the Nearest Neighbor problem. Both the closest destination and an associated optimal route need to be found based on travel time within the road network. Similarly, the optimal route also has to be recalculated whenever up-to-date traffic conditions are provided. In extreme circumstances, the closest destination may also change. For example, in an unknown city, we may want to find the location of the closest post office after we check into a hotel. From the query result, we are aware of the position and optimal route to the closest post office. In this case, we expect the navigation service not only to provide the adaptive route leading to it, but also to confirm the validity of the closest post office while traveling. If the traffic conditions do not change significantly, the optimal route may only need to be slightly modified. If the traffic conditions change considerably or there are serious traffic congestions around the anticipated post office destination, this post office may no longer be the closest one in terms of traveling time. A new post office location and optimal route must then be determined dynamically based on the current location and traffic conditions. In this scenario, the query is an en route query. To solve this problem, a dynamic nearest neighbor and route searching algorithm is required.

1.5 Introduction to the shortest path algorithms

The shortest path (SP) algorithms are among fundamental network analysis problems. Since 1957 a considerable progress has been made in the SP algorithms after Minty published his paper (1957). Minty succinctly described the basic SP problem for symmetrical networks (a network is symmetrical if for every pair of nodes the cost of a link between the two nodes is independent of their starting node). To state the problem beyond doubt, he suggested constructing a model of the given network. The model is made of strings, each string of the length proportional to the costs of the modelled link.

Finally, to find the links of the SP one has to pull the source node and the destination node of the journey as far away as possible. The tight strings are the links of the SP. Since 1957 there has been a number of major papers published, the most important were published by Bellman (1959), Dijkstra (1959) and Moore (1959). These articles were formative and most of the traffic research has used their results (for example Clercq (1972) and Cooke and Halsey (1966)). These articles are now included in references by most other publications.

There are a number of review papers. One of the utmost importance has been published by Dreyfus (1969). The review gives a comprehensive summary of the research, which has been carried out up to 1969. The article surveys over ten years of research, discussing the most crucial stages and pointing out the wrong and inefficient solutions. The paper also gives a brief solution of the SP problem for time varying costs of links, which is the basis of this report.

The shortest path algorithms are currently widely used. They are the basis of the network flow problems, tree problems and many related other problems. They determine the smallest cost of travel, of a production cycle, the shortest path in an electric circuit or the most reliable path. In the book by Ahuja at al., (1993) one can realise that the SP problem is an underlying problem of the network optimisation and that it is closely related to network flows or tree building issues.

Internet is a large field where the shortest path algorithms can be applied. The Internet problems involve data packages transmission with the minimal time or by the most reliable path. An example of the SP algorithms in the Internet is given by Cai et al., (1997). This paper proposes three SP algorithms. The devised algorithms are well explained. The article is closely related to the problem. The same algorithms can be used without fundamental changes to the urban traffic issues. The use of the proposed algorithms for public transportation networks will be studied in the section ‘Shortest path and the environment issues’.

Algorithms to be discussed here have a thirty-year old history and solutions to the fundamental problems are well known. The contemporary research is directed toward parallel computing as the method for further lowering of the time complexity bound of the shortest path algorithms. The report is not interested in the parallel approach. The article by Klein and Subramanian (1997) is an example of the shortest path parallel algorithm.

1.6 Introduction and definition of network concepts in connection with traffic issues

1.6.1Types of networks

There are several types of networks of special interest to the project: sparse, planar and road networks. Other types of networks (as grid or dense) are not taken into account.

1.6.2 Sparse networks

Sparse networks are those which have the number of links only a few times bigger than the number of nodes. A network of one hundred (100) nodes and four hundred (400) links would be considered sparse but a network with one hundred (100) nodes and five thousand (5000) links would be classified as dense.

Public transportation networks are sparse. From node approximately four links leave. If a sparse network was presented in a matrix form, then in each row of the matrix about only four places would be used, the rest would be left idle. For matrix network representation there are algorithms, which handle efficiently the sparse networks.

However, it is recommended not to use matrix-based algorithms since the matrix representation of a sparse network is highly inefficient. Instead of the matrix algorithms the tree building algorithms can be used as they store the sparse network information in an efficient way (usually using lists). In this thesis, the Dijkstra algorithm which is a basic tree building algorithm has been used. The matrix in figure 1.3 is an example of the inefficient matrix representation of a sparse network since there are more places unused than used.

1.6.3 Planar networks

There are a number of SP algorithms for planar graphs. Methods characteristic to planar graphs (as separators) lower the computational bound of the SP algorithms.

Since the road network and transport network are mostly planar, application of the algorithms from this group could bring more efficient solutions to our problem. However, not all road networks are planar, there are viaducts and bridges, which can destroy planarity and thus unable, limit or complicate the application of these algorithms. For this reason the methods for the planar graphs will not be considered. The article by Monika R. Henzinger et al., (1997) proposes three new algorithms for planar graphs.

1.6.4 Road networks

In the representation of a road network a link represents a road and a node represents a crossroad. The ratio of the number of links to the number of nodes is approximately 3. (Steenbrink, year 1958), gives an example of a road network with about 2000 nodes and 6000 links). The link costs are always non-negative. The road networks are usually planar and sparse. The number of nodes is big, usually expressed in thousands. Road networks contain loops, which are allowed since they may be only of a non-negative cost (the link costs are only non-negative). The road networks are of a special interest in this thesis.

The characteristic feature of the road networks is their nonnegative link lengths property. Dijkstra year made a good use of nonnegative lengths to design his algorithm. Because of this close relation between Dijkstra, algorithm and road characteristic, it should not be surprising that this report suffers constant ‘Dijkstra’ referring.

The project’s road network of the Kumasi City had about seven hundred and eighteen (718) nodes and one thousand one hundred and eighty – seven (1187) links. The network represents the link connections between nodes and the distances between them.

1.7 A general classification of the algorithms

The SP algorithms are either matrix algorithms or tree building algorithms (tree algorithms are also called labeling algorithms).

1.7.1 Matrix algorithms

Matrix algorithms store the network information in the matrix form and carry out the computations using basic matrix operations (as addition and multiplication of matrices or matrix’s elements). In dense networks is for all pair problems. The disadvantage of the matrix algorithms is the imposed matrix representation. The first disadvantage is the imposed inefficient matrix representation of a sparse network. The more significant disadvantage is that the matrix representation allows one directed link between two nodes (there can be at most two links between two nodes, but they have to be of distinct directions).

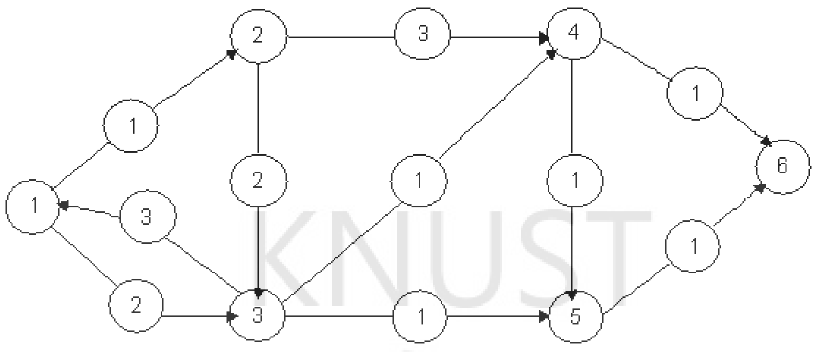


Figure1.1: A sample network that can be represented in a matrix form.

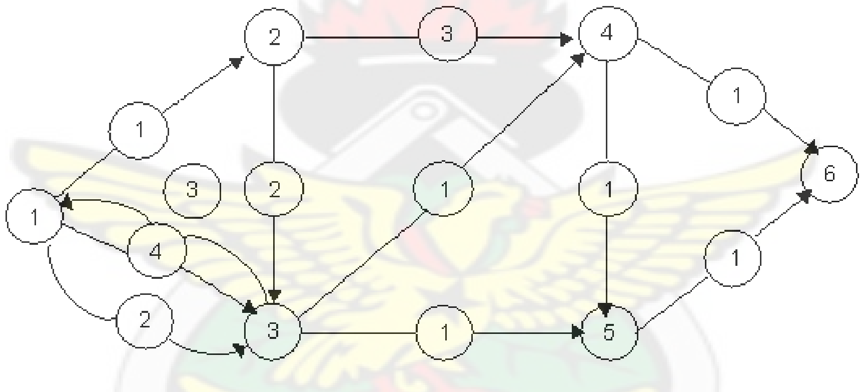


Figure 1.2: A sample network that can be not represented in a matrix form.

$$M = \begin{pmatrix} 0 & 1 & 2 & \infty & \infty & \infty \\ \infty & 0 & 2 & 3 & \infty & \infty \\ 3 & \infty & 0 & 1 & 1 & \infty \\ \infty & \infty & \infty & 0 & 1 & 1 \\ \infty & \infty & \infty & \infty & 0 & 1 \\ \infty & \infty & \infty & \infty & \infty & 0 \end{pmatrix}$$

Figure 1.3: Matrix representation of the network of Figure 1.1

The network from Figure 1.1 is specified by a matrix in Figure 1.3. Not every network can be represented in such a way. If a network has more than one directed link from a single node to some other node, then it cannot be represented in a regular matrix since it can store only one directed link going from a specific link to some other node.

A sample network capable of being represented as a matrix is depicted in Figure 1.1. The network has two links connecting the 1st node to the 3rd node. The link from the 1st node to the 3rd node is ascribed the cost of, which is stored in the M matrix in Figure 1.3. As $a_{13} = 2$. The link which goes in the reverse direction (from the 3rd node to the 1st node) is ascribed the cost of 3, this is stored as $a_{31}=3$.

If there was a need to represent the three links between the 1st and 3rd nodes from the Figure 1.3 then we realise we have run out of places in the matrix and the network cannot be fully represented by a matrix.

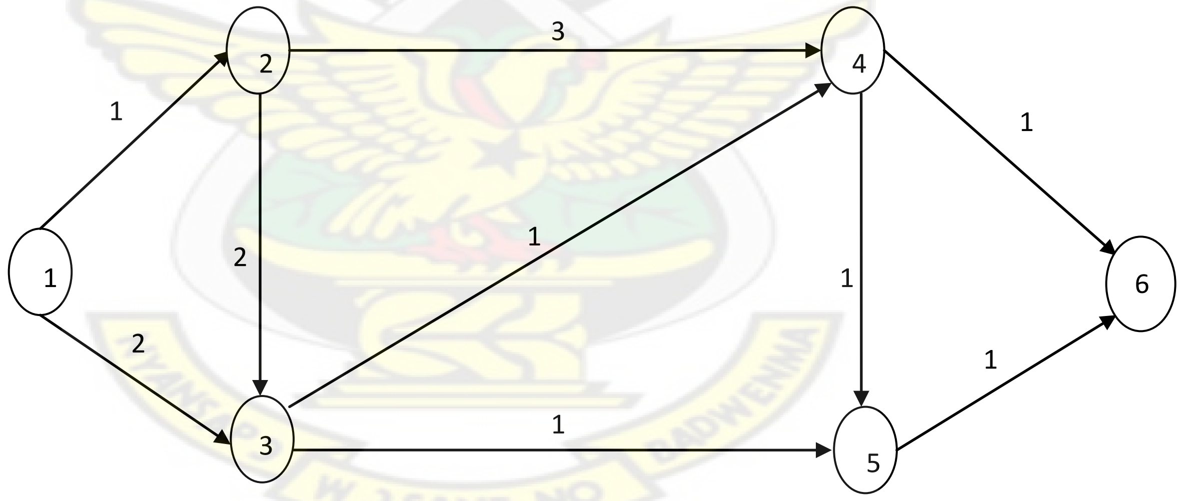
There can be some improvements of the matrix representation envisaged for coping with such an extended network. One improvement is a matrix of lists. An entry in this matrix of lists would not characterise only one directed link from one node to another but a list of directed links from this node to another node. However, this is not classified anymore as the matrix approach to the SP problem because computations of most matrix algorithms would not be performed anymore using basic matrix operations.

1.7.2 The tree building algorithms

The thesis algorithms are tree building algorithms. A tree building algorithm builds a tree with the root in the source node of the trip. Each node of the network can be either a leaf or a fork of the tree. A fork leads to another forks or leaves. There are certain true statements about the tree. The first is that there are *p* leaves then these leaves are *p* nodes of the biggest cost to reach among all nodes. The second says that each fork node (a node that is a fork in the tree) is of the cost smaller than a cost of any leaf node (a node that is a leaf in the tree).

Building such a tree is a dynamic programming task since the result of a node just reached can be used to calculate the cost of the node which can be reached immediately after this node. An example of building a tree for a simple network is presented in the Figure 1.4. To build this tree we use the Dijkstra algorithm.

a) network



b) Tree

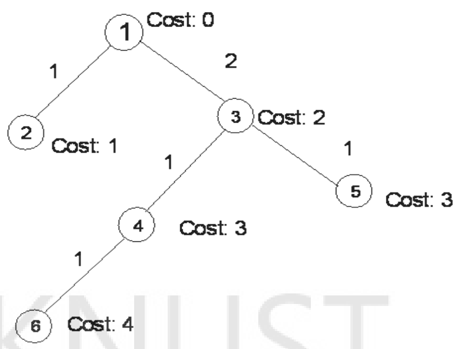


Figure1. 4: A network and its shortest path tree.

The input and the output to the shortest path algorithms

Depending where we are and where we want to go an algorithm can find as many SP's as it is necessary to satisfy us. The SP algorithms can be divided into groups that differ by the given input and the desired output. The groups are: one pair algorithms, one to many, many to one, and all pairs algorithms.

One pair

There are two nodes given: the source node and the destination node. A SP algorithm finds only one SP (if it exists) from the given source node to the given destination node. The tree algorithms are going to build an incomplete tree with the root in the source node. The tree will be complete up to the moment the destination node has been reached. The Dijkstra algorithm (1959) and the Bellman (1958) algorithm are examples are one pair algorithms.

One to many

Only the source node is specified. All shortest paths from this source node to all other nodes will be calculated. If there is a path from the source node to every other node, then there will be (n-1) SP's evaluated (*n* is the number of nodes is the network). A tree building algorithm will create a complete shortest path tree. The Dijkstra algorithm and the Bellman algorithm are also examples of one to many algorithms.

Many – to- One

This problem is given many source nodes and one destination node. To each source node there is time ascribed saying what time the journey starts from this node. The solution to the problem is to find the shortest path from any source node to the destination node that will result in reaching the destination node at the minimal time of arrival (not cost of the journey).

This type of a problem is easy to solve having the Dijkstra algorithm. The solution doesn't differ significantly from the Dijkstra algorithm. Only at the beginning one has to put all the source nodes into the priority queue with appropriate costs.

All pairs

For this algorithm group there is neither a necessity for source node nor for the destination node. An algorithm from this group calculates all the possible $n^2 - n = n(n-1)$ SP's, i.e. the algorithm is to find the shortest path for every pair of nodes. The number of paths is therefore (the paths form one and the same node is 0 and doesn't require calculation). The computations are mostly done on matrices. The Floyd algorithm (1962) is an example from the all pairs algorithm group.

The project makes use of ‘one to many’ algorithm and ‘many to one’ algorithm only. The ‘all pairs’ algorithm is not essential for the project and will not be discussed.

1.8 Remarks

The terms network and graph will be used interchangeably. The terms link cost and link length will also be used interchangeably though the term link cost will be used more generally. The terms starting node and finishing node will refer to the starting node and the finishing node respectively of a link. The source node and the destination node terms are going to denote the starting node (initial node) and the finishing node (terminal node or sink) of the overall shortest path, respectively.

1.9 LIMITATIONS

Limitations of the existing algorithms and previous research

Among several variants of the SP algorithms there is a group of algorithms, which could be applied to solve the present issue, but the solution would not be efficient. An obvious group of algorithms is the one that gives a more general solution than needed and their solution would be redundant. A good example of a group giving a redundant solution is the ‘all pairs’ group of the SP algorithms: only one pair of nodes would be used from the set of all pairs.

Matrix algorithms are not of a good use for sparse networks. Matrix algorithms are memory consuming and for sparse networks time consuming. The implementation of the Dijkstra’s algorithm based on a matrix is inefficient for road networks. Therefore the matrix algorithms are abandoned from this point for the rest of the report.

Bellman (1958) designed an algorithm for networks with also negative link lengths. The concern about negative link length made the Bellman algorithm more general than the Dijkstra algorithm, but it also made it less efficient for the networks without negative links. Since the report deals with road networks (of which the inherent feature is the nonnegative link lengths) only, the study of the Bellman algorithm will not be of much interest for the report.

Another example of an algorithm providing a solution more general than needed is the algorithm devised by Cai et al., (1997). This algorithm is an enhanced Dijkstra algorithm. The algorithm processes a network to whose links two attributes are ascribed: cost of traversing and time of traversing. The algorithm searches for the cheapest path, (the shortest path in terms of the cost) which satisfies an extra condition: the overall time of such a path (the time required to traverse the links of the path) does not exceed a given T. The algorithm can be used to solve the project’s problem by setting every link’s time to 0 and setting T = 0. Such a solution is not efficient thought.

An algorithm by Cooke and Halsey (1966) finds the shortest path from one source to one destination in a network with time dependent link costs. The algorithm is based on the Bellman approach (the paper was also submitted by Bellman). The algorithm works out but unfortunately is inefficient. The description of the algorithm is complicated and not clear. The algorithm was revised by Dreyfus (1969) who pointed out the inefficiency of the algorithm and proposed his solution to the problem.

Clercq (1972) proposed an algorithm for public transport. The algorithm does not use timetables which are the main requirement for the group project; the algorithm proposed by Clercq uses frequencies of bus lines instead. The algorithm is vaguely described without emphasis of main ideas and firm specification of the algorithm. The paper does not report the time complexity and memory requirements.

Remark

This thesis encountered (mostly in the book by Streenbrink (1974) and in the article by Clercq (1972) references to public transport papers written in languages (mostly Dutch), which the author of this report does not speak. Because of this reason many potentially useful sources have not been reached and cannot be discussed.

1.10 OBJECTIVE

The objective of this thesis is to create a formation movement shortest path finding algorithm for

vehicles to implement tactical movement within a large metropolis such as Kumasi and optimization scheme for transportation planning and analysis to provide a major advantage in its ability to take into account a range of different, often unrelated criteria, even if these criteria cannot be directly related to quantitative outcome measures. The approach described is specifically addressed to transportation networks but it is also applicable to other types of physical networks including computer networks. Research into developing an evolutionary approach to transportation network optimisation, by use of a carefully chosen fitness function, is outlined.

1. To make it easy for visitor/tourists to navigate the Adum and to find their destinations far more easily and to help workers.
2. To find the best possible route for users to their destinations in Adum so as to efficiently use their distance.

Determining the "best" route or set of routes for linear utilities such as highways, pipelines, and power transmission lines, through a landscape has been the subject of much research in geographic information systems (GIS) and spatial decision making. Specifying an optimal corridor that connects an origin and destination is analogous to identifying a least-cost-path through a varying space. Extensive research efforts have been executed to solve the problems for many years (Tomlin, 1990; Eastman, 1989; Douglas, 1994; Berry 2004). Tomlin's (1990) Spread algorithm generates an accumulated-cost-surface iteratively and delineates the weighted shortest path from any location to a destination by tracing back along slope lines. Eastman (1989) implemented a similar, but more efficient, push broom algorithm, which is able to produce an accumulated cost surface within three iterations. Many of the existing least-cost-path algorithms in GIS are derived from the Dijkstra's shortest path algorithm and intend to generate a global optimal solution.

1.11 JUSTIFICATION

With the development of geographic information systems (GIS) technology, network and transportation analyses within a GIS environment have become a common practice in many application areas. A key problem in network and transportation analyses is the computation of shortest paths between different locations on a network. Sometimes this computation has to be done in real time. For the sake of illustration, let us have a look at the case of a 911 call requesting an ambulance to rush a patient to a hospital. Today it is possible to determine the fastest route and dispatch an ambulance with the assistance of GIS. Because a link on a real road network in a city tends to possess different levels of congestion during different time periods of a day, and because a patient's location cannot be expected to be known in advance, it is practically impossible to determine the fastest route before a 191 call is received. Hence, the fastest route can only be determined in real time. In some cases the fastest route has to be determined in a few seconds in order to ensure the safety of a patient. Moreover, when large real road networks are involved in an application, the determination of shortest paths on a large network can be computationally very intensive. Because many applications involve real road networks and because the computation of a fastest route (shortest path) requires an answer in real time, a natural question to ask is: Which shortest path algorithm runs fastest on real road networks?

Although considerable empirical studies on the performance of shortest path algorithms have been reported in the literature (Dijkstra 1959; Dial et al,. 1979; Glover et al., 1985; Gallo and Pallottino 1988; Hung and Divoky 1988; Ahuja et al., 1990; Mondou et al., 1991; Cherkassky et al., 1993; Goldberg and Radzik 1993), there is no clear answer as to which algorithm, or a set of algorithms runs fastest on real road networks. In a recent study conducted by Zhan and Noon (1996), a set of three shortest path algorithms that run fastest on real road networks has been identified. These three algorithms are: 1) the graph growth algorithm implemented with two queues, 2) the Dijkstra’s algorithm implemented with approximate buckets, and 3) the Dijkstra’s algorithm implemented with double buckets. Dijkstra's algorithm was then used on the node graph, but modified to run faster using this extra data. However this optimization changes Dijkstra's Algorithm so that it only finds a path, rather than the shortest path.

Other applications of Dijkstra are

- Routing of postal workers
- Routing robots through a warehouse
- Drilling holes on printed circuit board

1.12 METHODOLOGY

The methodology employed included review of relevant literature of the types of Dijkstra algorithm and methods employed in the solution of the Dijkstra algorithm and to develop computer solutions – ArcGIS and VB.net for faster computation of Dijkstra algorithm

CHAPTER 2

LITERATURE REVIEW

Shortest path problems are the most fundamental and the most commonly encountered problems in the study of transportation and communication networks (Syslo et al., 1983). There are many types of shortest path problems. For example, we may be interested in deterring the shortest path (i.e , the most economic path or fastest path or minimum – fuel consumption path) from one specified node in the network to another specified node; or may need to find shortest paths from a specified node to all other nodes.

Arrival time dependent shortest path finding is an important function in the field of traffic information systems or telematics. However, large number of mobile objects on the road network results in a scalability problem for frequently updating and handling their real-time location.

Kyoung-Sook Kim, et al. (2005) proposed a query processing method in MANET (Mobile Ad-hoc Network) environment to find an arrival time dependent shortest path with a consideration of both traffic flow and location in real time. Since their traffic flow method does not need a centralized server, time dependent shortest path query is processed by in-network way. In order to reduce the number of messages to forward and nodes to relay, the control introduce an on-road routing, where messages are forwarded to neighboring nodes on the same or adjacent road segments. This routing method allows the collection of traffic information in real time and the reduction of the number of routing messages.

Experiments show that the number of forwarded messages is reduced in an order of magnitude with our on-road routing method compared to LAR-like method. At best, our method reduces about fifty - seven (57) times less messages.

The Integrated Transport Information System (ITIS) project for the Klang Valley was initiated by the Federal Government in early 2001 and deployed on a design – build basis in 3Q 2002. With the City Hall, Kuala Lumpur as the implementing agency, the project was successfully completed and handed over in June 2005.

Using a spectrum of different technologies and equipment, the ITIS has since been gainfully used by City Hall as well as the police for management of road network operations and particular for management of incidents over a network comprising of over 200kms of roadways. Omar (1994) discussed the technologies used in the ITIS network operations, in particular in the detection and management of incidents, lesson learnt – to – date as well as the roadmap for future operations and ITS related deployment.

Optimisation of forest road network is an important part of logging planning. Matthews (1942) was first to introduce a method for optimisation of road spacing based on minimisation of road and skidding cost. Ghaffarian (2000), found the best road network for a district harvested by skidder. The skidding model developed by stepwise regression model was used to predict the cost skidding per cubic meter for the thirty – nine (39) nodes, which were planned in the district map. The harvesting volume and road cost per each node were computed. The data were entered into network 2000 and the shortest path algorithm; simulated annealing and great deluge algorithms were run to find the best solution to optimise logging cost of the district. The result showed which roads can be eliminated from the existing forest road network.

Due to the reduction of travel time between regions in recent years by the development of transportation networks in Japan, the opportunities for anyone living in both urban and rural areas to meet people and to use urban facilities have increased. However, the various functions of smaller cities will be absorbed into these larger metropolises, since the sphere of urban influence from big cities spreads to greater areas. In these backgrounds, the impacts of developments of expressway networks are analyzed by using the increment in interchangeable population and the changes in trade and recreation areas. Problems rural cities will have to bear in the near future are also discussed. It can be said that one result of this is the sphere of urban influence from big cities will spread to the retailing industry in rural areas in the near future. In order to utilize the expressway improvements effectively in rural cities, new and creative development policies are required which are dissimilar to those of major cities (Hirose, 1994).

Setoguchi et al., (1994) analyzed the influence of network extension and the revised toll on the traffic of urban expressways in Fukuoka, Kita-Kyushu, and Nagoya, and, in estimating the traffic of these urban expressways, based on their maintenance and management, the author of this paper straightened out the relationships between the toll, a factor that determines the conversion amount of traffic, and the value of time to examine what traffic allocation calculations should be at a practical application level.

Hiroshi et al., (1994) proposed to determine the groups of road sections to be simultaneously constructed and the priority between them, considering the disutility of road construction and the priority and simultaneity of construction between road sections. Dynamic programming is utilized for an optimization procedure. The mathematical modelling of the problem and its solution technique are emphasized. An example problem is included and illustrated for showing the applicability of the model. The results indicate that the proposed method is useful for multi-stage determination problem such as in the road network planning.

Talib et al., (1994) described a method which determines a plan for improving a road network taking into consideration the impact of increasing number of trip generation. In this method, the increasing number of trip generation in study area is distributed to other unflourished residential zones, and the groups of road segments to be simultaneously constructed as well as their priority are determined so that the limited budget will be effectively used. The dynamic programming is utilized for the optimization procedure.

In the recent years, with the development of social economy in China, the public urban transportation has greatly changed. In Beijing city, taxi traffic system has become a new kind of public transit means for resident trips. Takeshi et al., (1994) first introduced the development history of taxi traffic system of Beijing city, which includes three stages of taxi service trades from original to now. Through the introduction, the historical reasons that taxi traffic development of Beijing city is increasingly expanding can be known. The second part analyses the interior and exterior circumstances and impact factors of taxi traffic system, and describes the improvement of relative traffic installation and the change of transportation policy of Beijing city. Further, they preliminarily study the developing strategies of Beijing taxi traffic system through the discussion on the change of passenger flow and the estimation of corresponding factors, and comparison with other big cities such as Taipei, Mexico city etc.

The new Young-Jong island international airport (NYIA) and the related hinder land development is expected to be a catalyser, which stimulates even further Korea's economic power and participation of a global market. The basis of the development plan is characterized by following aspects: backup for the Northeast Asian hub due to the globalization trends, urgency of the social overhead capital building, rapid increasing of aviation demand and shortage of the existing facilities. According to this basis, the plan includes international business centre, community development and free trade zone. The main impacts of NYIA plan can be separated into the reinforcement of international competitiveness, the boosting of regional development and the opening of a window on cultural exchange. Also, it is necessary to participate the private sectors and to control different opinions within various government departments (Lee ,1994).

Chikashi et al (1994) described the characteristics of traffic behaviours such as traffic purposes at holidays and week days, selection of transportation modes, walking and selection of parking place to the central area of a local city. Data are obtained from response to questionnaires for people in Miyazaki City. The choice behaviour of parking places is analyzed by using Aggregated Logit Model. The analyses results and answers show that it is necessary to decrease the traffic resistance on walking by such method as pedestrian and vehicular segmentation to keep traffic safe for pedestrian.

Aminu, (2007) put forward the problem of finding shortest paths in traversing some location within the Sokoto Metropolis. In particular, it explores the use of Dijskra's alogrithm in constructing the minnimum spanning tree considering the dual carriage ways in some of the road in Kumasi Metropolis. The results shows that a reduction in the actual distance as compared with ordinary routing. These results indicate , clearly the importance of this type of algorithms in the optimisation of network flows.

Lehr- und Forschungsgebiet Operations Research und Logistik Management RWTH Aachen, Templergraben 64, 52056 Aachen, Germany The elementary shortest path problem with resource constraints (ESPPRC) is a widely used modelling tool in formulating vehicle routing and crew scheduling applications. The ESPPRC consists of finding shortest paths from a source to all other nodes of a network that do not contain any cycles, i.e. duplicate nodes. The ESPPRC occurs as a sub problem of an enclosing problem and is used to implicitly generate the set of all feasible routes or schedules, as in the column generation formulation of the vehicle routing problem with time windows (VRPTW). The ESPPRC problem being NP-hard in the strong sense, classical solution approaches are based on the corresponding non-elementary shortest path problem with resource constraints (SPPRC), which can be solved using a pseudo-polynomial labelling algorithm. While solving the enclosing master problem by branch-and-price, this sub problem relaxation leads to weak lower bounds and sometimes impractically large branch-and-bound trees.

A compromise between solving ESPPRC and SPPRC is to forbid cycles of small lengths. In the SPPRC with k-cycle elimination (SPPRC-k-cyc) only paths with cycles of length at least k + 1 are allowed. The case k = 2 which forbids sequences of the form i to j - i is well known, and has been used successfully to reduce integrality gaps for the VRPTW propose a new definition of the dominance rule among labels for dealing with arbitrary values of k ≥ 2. The numerical experiments on the linear relaxation of some hard VRPTW instances from Solomon's benchmark set show that k-cycle elimination with k ≥ 3 can substantially improve the lower bounds. Using well-known techniques for branching and cutting, the new algorithm has proven to be a key ingredient for getting exact integer solutions of knowingly hard problems from the literature.

Eklund, et al (1994) discussed the implementation of Dijkstra’s classic double bucket algorithm for path finding in connected networks. The work reports on a modification of the algorithm embracing both static and dynamic heuristic components and multiple source nodes. The modified algorithm is applied in 3D Spatial Information System (SIS) for routing emergency service vehicles. The algorithm has been implemented as a suite of modules and integrated into a commercial SIS software environment. Genuine 3D spatial data is used to test the algorithm on the problem of vehicle routing and rerouting under simulated earthquake conditions in the Japanese city of Okayama. Coverage graphs were also produced giving contour lines joining points with identical travel times.

Shortest Path problems are inevitable in road network applications such as city emergency handling and drive guiding system, in situations where the optimal routings have to be found. As the traffic condition among a city changes from time to time and there are usually huge amounts of requests that occur at any moment, needs to quickly find the solution. Therefore, the efficiency of the algorithm is very important. Some approaches take advantage of pre-processing that compute results before demanding. These results are saved in memory and could be used directly when a new request comes up. This can be inapplicable if the devices have limited memory and external storage. Liang (2005) aimed only at investigating the single source shortest path problems and intended to obtain some general conclusions by examining three approaches: Dijkstra's shortest path algorithm, Restricted search algorithm and A* algorithm. To verify the three algorithms, a program was developed under Microsoft Visual Basic .Net environment. The three algorithms were implemented and visually demonstrated. The road network example is a graph data file containing partial transportation data of the Ottawa city.

Since the aftermath of typhoon Herb in 1996, all sort of flood and drought followed in 2002 have claimed lives and countless property, which have imposed serious economic damage on the country. The collection of flood information is the basis for established prevention system. It is anticipated that flood information management system will include flood insurance, flood   warning, damage notification and incorporation with GIS in the future to provide further capabilities.

The use of the ArcGIS and mathematical programming, in accordance to the properties of the disaster, aims pragmatically at a balance between the reliefs of a disaster and the shortest time for conveying the equipments, and to construct the optimal model of the equipment's transportation and mobilisation of the emergency. The system could trace and manage more efficiently, the equipments in urgent need of repair, and reconstruct the state of the recovery.

Humblet (1988) employed a distributed algorithm to compute shortest paths in a network with changing topology. It does not suffer from the routing table looping behaviour associated with the Ford-Bellman t-distributed shortest path algorithm although it uses truly distributed processing. Its time and message complexities are evaluated.

Saunders and Takaoka presented new algorithms for computing shortest paths in a nearly acyclic directed graph G = (V, E). The new algorithms improve on the worst-case running time of previous algorithms. Such algorithms use the concept of a 1-dominator set.

A 1-dominator set divides a graph into a unique collection of acyclic sub graphs, where each acyclic sub graph is dominated by a single associated trigger vertex. The previous time for computing a 1- dominator set is improved from O(mn) to O(m), where $m = |E|$ and $n = |V|$. Efficient shortest path algorithms only spend delete-min operations on trigger vertices, thereby making the computation of shortest paths through non-trigger vertices easier. Under this framework, the time complexity for the all-pairs shortest path (APSP) problem is improved from O(mn + nr log r) to $O(mn + r^2 \log r)$, where r is the number of triggers. Here the second term in the complexity results from delete-min operations in a heap of size r. The time complexity of the APSP problem on the broader class of nearly acyclic graphs, where trigger vertices correspond to any pre computed feedback vertex set, is similarly improved from O(mn + nr^2) to O(mn + r^3). The paper also mentioned that the 1-dominator set concept can be generalised to define a bidirectional 1-dominator set and k-dominator sets.

When you drive to somewhere ‘far away’, you will leave your current location via one of only a few ‘important’ traffic junctions. Starting from this informal observation, develop an algorithmic approach—transit node routing— that allows us to reduce quickest-path queries in road networks to a small number of table lookups. Present two implementations of this idea, one based on a simple grid data structure and one based on highway hierarchies. For the road map of the United States, our best query times improve over the best previously published figures by two orders of magnitude. Our results exhibit various trade-offs between average query time (5 µs to 63 µs), pre-processing time (59min to 1200min), and storage overhead (21 bytes/node to 244 bytes/node) Bast et al., (2006).

On the basis of analyzing the advantages and disadvantages of the shortest path algorithm and the problem solving based on knowledge method, it is clearly showed that neither the algorithm, which provides the precise solution nor the common method, which is totally suitable to people’s usual finding activities and based on the common sense, can provide us with a satisfactory solution. However, they can be complementary to each other, and this has made the combined use of the two to become a necessity.

Rong, WENG Min, DU QingYun, and CAI ZhongLiang put forward the combination use of knowledge and algorithm for way-finding. In this combined method, the knowledge is used for retrieving the case and isolating the searching area while algorithm is used for finding out the best solution in the isolated areas. The study shows that although the new approach cannot always ensure a most accurate solution, it not only prunes off a lot of search space but also produces routes that meet people's preference of travelling on familiar and major roads.

Yu et al., (1995) proposed a hierarchical algorithm for approximating all pairs of shortest paths in a large scale network. The algorithm begins by extracting a high level sub network of relatively long links (and their associated nodes) where routing decisions are most crucial. This high level network partitions the shorter links and their nodes into a set of lower level sub networks. By fixing gateway nodes within the high level network for entering and exiting these sub networks, a computational savings is achieved at the expense of optimality. They explore the magnitude of this tradeoff between computational savings and associated error both analytically and empirically with a case study of the South-eastern Michigan traffic network. An order of magnitude drop in computational times was achieved with an on – line route guidance simulation at the expense of a five percent (5%)   increase in expected trip times.

A lot of the related work on shortest paths in stochastic networks has focused on the notion of shortest paths in expectation, e.g., (Bertsekas and Tsitsiklis 1991). Other models have added costs on the edges in addition to travel times (Chabini 2002), (Miller-Hooks and Mahmassani 2000) where the costs depend on the realized travel times and in this way can capture a measure of uncertainty.

Finding the path of smallest expected length trivially reduces to deterministic shortest path problems and does not take into account risk in predicting the optimal route. Since most real world applications care about a tradeoff between risk and expectation, we consider nonlinear objectives that capture more information about the edge distributions. Closest to this model, Loui (Loui, 1983) considered a decision analytic framework for optimal paths under uncertainty, however, he only studied monotone increasing cost functions and his algorithm has running time O (nn) in the worst case. Mirchandani and Soroush (Mirchandani and Soroush, 1985) extended his work to a quadratic cost function of the path length, however their algorithm is also an exhaustive search over all potentially optimal paths, and thus exponential in the worst case.

Another branch of the stochastic shortest path literature has focused on adaptive algorithms (Fan, Kalaba and Moore 2000), (Gao and Chabini, 2002), (Boyan and Mitzenmacher, 2001), which compute the optimal next edge in light of lengths or travel times already realized en route to the current node. Another direction has been to give approximations and heuristics for expected shortest paths in stochastic networks with nonstationary (time-varying) edge length distributions (Miller-Hooks and Mahmassani, 2000), (Fu and Rilett, 1998), (Hall, 1986), to list a few. In this proposal, we only consider stationary edge length distributions that do not change with time; time-varying distributions will be the subject of future work.

Delava et al., (2008) show that *a*ccomplishing an effective routing of emergency vehicle will minimize its response time and will thus improve the response performance. Traffic congestion is a critical problem in urban area that influences the travel time of vehicles. The aim of this study is developing a spatial decision support system (SDSS) for emergency vehicle routing. The proposed system is based on integration of geospatial information system (GIS) and real-time traffic conditions. In this system dynamic shortest path is used for emergency vehicle routing. This study investigates the dynamic shortest path algorithms and offers an applicable solution for emergency routing. The shortest path applied is based on the Dijkstra algorithm in which specific rules have been used to intelligently update the proposed path during driving. Results of this study, illustrate that dynamic vehicle routing is an efficient solution for the reduction of travel time in emergency routing. Finally, it is shown that using GIS in emergency routing offers a powerful capability for network analysis, visualization and management of urban traffic network. Spatial analysis capabilities of GIS are used to find the shortest or fastest route through a network. These capabilities of GIS for analyzing spatial networks enable them to be used as decision support systems (DSSs) for dispatching and routing of emergency vehicles.

In agent based traffic simulations which use systematic relaxation to reach a steady state of the

Scenario, the performance of the routing algorithm used for finding a path from a start node to an end node in the network is crucial for the overall performance. For example, a systematic relaxation process for a large scale scenario with about 7.5 million inhabitants (roughly the population of Switzerland) performing approximately three trips per day on average requires about 2.25 million route calculations, assuming that 10% of the trips are adapted per iteration. Expecting about 100 iterations to reach a stable state, 225 million routes have to be delivered in total.

Nicolas Lefebvre and Michael Balmer (2008) focus on routing algorithms and acceleration methods for point-to-point shortest path computations in directed graphs that are time-dependent, i.e. link weights vary during time. The work is done using MATSim-T (Multi-Agent Traffic Simulation Toolkit) which is used for large-scale agent-based traffic simulations. The algorithms under investigation are both variations of Dijkstra's algorithm and the A*-algorithm. Extensive performance tests are conducted on different traffic networks of Switzerland. The fastest algorithm is the A* algorithm with an enhanced heuristic estimate: While it is up to 400 times faster than Dijkstra's original algorithm on short routes, the speed compared to Dijkstra's diminishes with the length of the route to be calculated.

CHAPTER 3

TRANSPORTATION NETWORKS ANALYSIS

3.1 Background of Graph Theory

In this chapter, some fundamental concepts of graph theory are introduced and will be referred to in subsequent discussions.

Definition of a Graph

In mathematics and computer science, graph theory deals with the properties of graphs. Informally, a graph is a set of objects, known as nodes or vertices, connected by links, known as edges or arcs, which can be undirected or directed (assigned a direction). It is often depicted as a set of points (nodes, vertices) joined by links (the edges). Precisely, a graph is a pair, $G = (V; E)$, of sets satisfying $E \in [V]$; thus, the elements of E are 2-element subsets of V. The elements of V are the nodes (or vertices) of the graph G, the elements of E are its links (or edges). In this case, E is a subset of the cross product $V * V$ which is denoted by $E \in [V]$. To avoid notational ambiguities, we shall always assume that $V \cap E = \varnothing$.

A connected graph is a non-empty graph G with paths from all nodes to all other nodes in the graph. The order of a graph G is determined by the number of nodes. Graphs are finite or infinite according to their order. In this thesis, the graphs are all finite and connected. Furthermore, a graph having a weight, or number, associated with each link is called a weighted graph, denoted by G = (V; E; W). An example of a weighted graph is shown in Figure 3.1

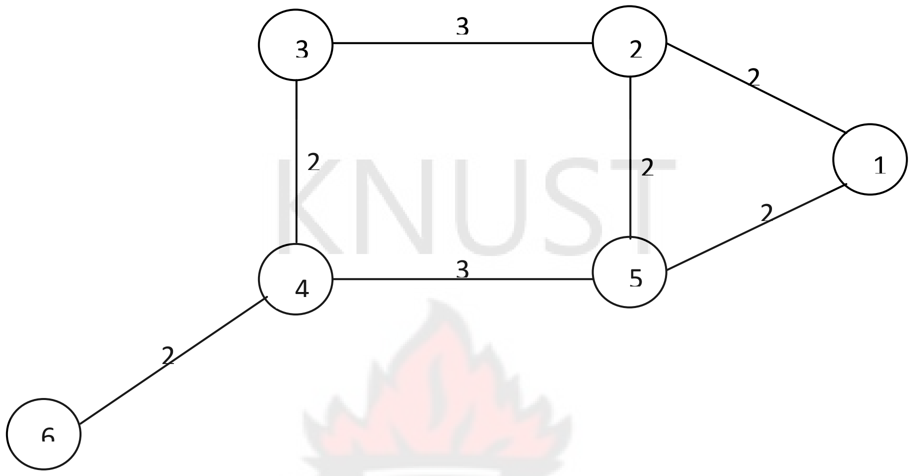


Figure 3.1: A diagram of a weighted graph with 6 nodes and 7 links.

Degree of a Vertex (Node)

A node v is incident with a link e if $v \in e$; then e is a link at v. The two nodes incident with a link are its end nodes. The set of neighbors of a node v in G is denoted by $N(v)$.

The degree $d(v)$ of a node v is the number $|E(v)|$ of links incident on v. This is equal to the number of neighbors of v. A node of degree 0 is isolated. The number $\delta(G) = \min\{d(v) \mid v \in V\}$ is the minimum degree of G, while the number $\Delta(G) = \max\{d(v) \mid v \in V\}$ is the maximum degree.

The average degree of G is given by the number

$$d(G) = \frac{1}{|V|}\sum_{v\in V} d(v)$$

(3.1)

Clearly,

$$\delta(G) \le d(G) \le \Delta(G)$$

(3.2)

The average degree globally quantifies what is measured locally by the node degrees: the number of links of *G* per node. Sometimes it is convenient to express this ratio directly, as $\varepsilon(G) = |E|/|V|$. The quantities *d* and ε are intimately related. Indeed, if we sum up all of the node degrees in *G*, we count every link exactly twice: once from each of its ends. Thus,

$$|E| = \frac{1}{2}\sum_{v \in V} d(v) = \frac{1}{2} d(G).|V|$$

(3.3)

and therefore

$$\varepsilon(G) = \frac{1}{2} d(G)$$

(3.4)

Graphs with a number of links that are roughly quadratic in their order are usually called dense graphs. Graphs with a number of links that are approximately linear in their order are called sparse graphs. Obviously, the average degree *d* (*G*) for a dense graph will be much greater than that of a sparse graph.

In a graph, a path, from a source node *s* to a destination node *d*, is defined as a sequence of nodes $(v_0, v_1, v_2, ..., v_k)$ where $s = v_0$, $d = v_k$, and the links $(v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k)$ are present in E. The cardinality of a path is determined by the number of links. The cost of a path is the sum of the link costs that make up the path.

An optimal path from node *u* to node *v* is the path with minimum cost, denoted by *(u, v)*. The cost can take many forms including travel time, travel distance, or total toll. In my research, the cost or weight of a path stands for the travel time which is needed to go through the path.

3.2 Network Data Models

Graph algorithms need efficient access to the graph nodes and links that are stored in the computer's memory. In typical graph implementations, nodes are implemented as structures or objects and the set of links establish relationships (connections) between the nodes. There are several ways to represent links, each with advantages and disadvantages. The data structure used depends on both the graph structure and the algorithm used for manipulating the graph. Theoretically, one can distinguish between list and matrix structures but in concrete applications the best structure is often a combination of both. Among these data structures, graphs are commonly represented using the incidence matrix, adjacency matrix and adjacency list.

3.2.1 Incidence Matrix

The incidence matrix of an undirected graph is a (0, 1)-matrix, which has a row for each link and a column for each node. In this case, (*v*, *e*) = 1 if, and only if, node *v* is incident upon link e and (*v*, *e*) = 0 otherwise. For a directed graph, the incidence matrix can be represented as (*v*, *e*) = 1 or -1, according to whether the link leaves node *v* or it enters node *v*. The resulting incidence matrix for the undirected graph is as in shown Figure 3.2.

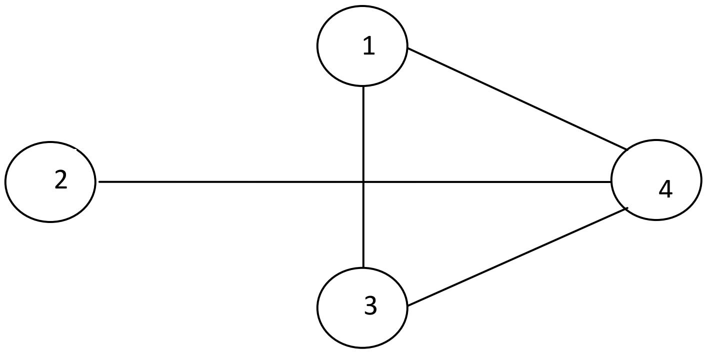


Figure 3.2: An Undirected Graph

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

The matrix above shows for the undirected graph in figure 3.2

3.2.2 Adjacency Matrix

The adjacency matrix of a graph is an *n* by *n* matrix stored as a two-dimensional array with rows and columns labeled by graph nodes. A 1 or 0 is placed in position (*u*, *v*) according to whether *u* and *v* are adjacent or not. Node *u* and *v* are defined as adjacent if they are joined by a link. For a simple graph with no self-loops, the adjacency matrix must have 0s in the diagonal. For an undirected graph, the adjacency matrix is symmetric. Following is the adjacency matrix for Figure 3.2.

$$\begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

The adjacency matrix the undirected graph in figure 3.2

3.2.3 Adjacency List

The adjacency list is another form of graph representation in computer science. This structure consists of a list of all nodes in a given graph. Furthermore, each node in the list is linked to its own list containing the names of all nodes that are adjacent to it.

In addition, the distances to those nodes are also stored. The adjacency list for Figure 3.3 can be described by Figure 3.4.

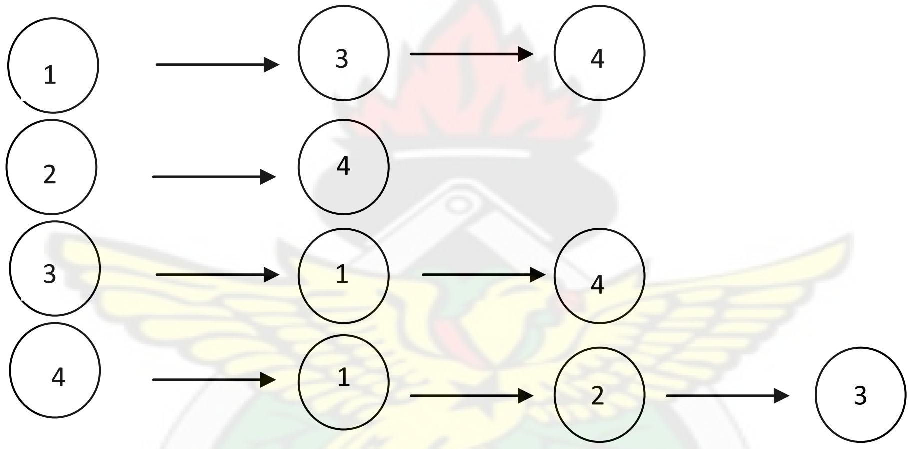


Figure 3.3: An Adjacency List

The above adjacency list is easy to follow and clearly illustrates the adjacent nature of the four nodes. It is most often used when the graph contains a small to moderate number of links.

3.2.4 Transportation Network Data Model

A transportation network is a type of directed, weighted graph. The use of GIS for transportation applications is widespread and a fundamental requirement for most transportation GIS is a structured road network.

In developing a transportation network model, the street system is represented by a series of nodes and links with associated weights. This representation is an attempt to quantify the street system for use in a mathematical model. Inherent in the modeling effort is a simplification of the actual street system. The network nodes represent the intersections within the street system and the network links represent the streets. The weights represent travel time between the nodes.

As a specialized type of graph, a transportation network has characteristics that differ from the general graph. A suitable data structure is required to represent the transportation network. Comparing the three data structures, an adjacency list representation of the graph occupies less space because it does not require space to represent links which are not present. The space complexity of an adjacency list is $O(|E|+|V|)$, where $|E|$ and $|V|$ are the number of links and nodes respectively. In contrast, incidence matrix and adjacency matrix representations contain too many 0s which are useless and redundant in storage. The space complexity of incidence matrices and adjacency matrices are $O(|E| \times |V|)$ and $O(|V|^2)$ respectively. In the following discussion, we shall take a more detailed look at the three data models in terms of storage space and suitable operations. Using a naive linked list implementation on a 32-bit computer, an adjacency list for an undirected graph requires approximately $16 \times (|E| + |V|)$ bytes of storage space.

On the other hand, because each entry in the adjacency matrix requires only one bit, they can be represented in a very compact way, occupying only $|V|^2/8$ bytes of contiguous space. First, we assume that the adjacency list occupies more memory space than that of an adjacency matrix. Then

$$16\text{x}(|E|+|V| \geq |V|^2 / 8$$

Based on equation (3.1.2) in section 3.1, we have,

$$16\text{x}(\frac{1}{2}d(G)\text{x}|V|+|V|) \geq |V|^2 / 8$$

where *d* (*G*) is the average degree of *G*.

$$d(G) \geq \frac{|V|-128}{64}$$

(3.5)

This means that the adjacency list representation occupies more space when equation (3.5) holds.

In reality, most transportation networks are large scale sparse graphs with many nodes but relatively few links as compared with the maximum number possible (|*V*| × (|*V*| −1) for maximum). That is, there are no more than 5 links (Δ (*G*) ≈ 5) connected to each node. In most cases there are usually 2, 3 or 4 (δ (*G*) = 2) links, although the maximum links is |*V*|-1 for each node. Also, road networks often have regular network structures and a normal layout, especially for well planned modern cities. Again most transportation networks are near connected graphs, in which any pair of points is traversable through a route.

Assuming the average degree of a road network is 5, equation 3.5 holds only if $|V| \leq 448$. However, most road networks contains thousands of nodes where $|V| >> 448$. As a result, equation 3.2 cannot hold. Thus, the adjacency list representation occupies less storage space than that of an adjacency matrix. For example, consider a road network containing 10000 nodes. If an adjacency matrix is employed to store the network, at least 10 megabytes of memory space is required. It will most likely take more computational power and time to manipulate such a large array, and then it is impossible to conduct routing searches in some mobile data terminals, such as smart phones and Personal Digital Assistance (PDAs).

The comparison between the adjacency matrix and incidence matrix can give the same result. Assuming an adjacency matrix occupies more storage space than that of an incidence matrix,

then

$$|V|^2 \geq |E| \times |V|$$

From equation 3.2, we obtain,

$$d(G) \leq 2 \qquad \textbf{(3.6)}$$

This means that the adjacency matrix representation occupies more space if and only if equation 3.6 holds. Since the minimum degree of transportation network is 2 ($\delta(G) = 2$), then equation 3.6 is invalid. As a result, the adjacency matrix occupies less storage space than that of the incidence matrix. Since the adjacency matrix cannot compete with the adjacency list in terms of storage space (i.e., requires more space), it follows that the incidence matrix will also not be able to compete.

Other than the space tradeoff, the different data structures also facilitate different operations. It is easy to find all nodes adjacent to a given node in an adjacency list representation by simply reading its adjacency list. With an adjacency matrix, we must scan over an entire row, taking $O(|V|)$ time, since all $|V|$ entries in row v of the matrix must be examined in order to see which links exist. This is inefficient for sparse graphs since the number of outgoing links j may be much less than $|V|$.

Although the adjacency matrix is inefficient for sparse graphs, it does have an advantage when checking for the existence of a link $u \rightarrow v$, since this can be completed in $O(1)$ time by simply looking up the array entry $[u; v]$. In contrast, the same operation using an adjacency list data structure requires $O(j)$ time since each of the j links in the node list for u must be examined to see if the target is node v.

However, the main operation in a route search is to find the successors of a given node and the main concern is to determine all of its adjacent nodes. The adjacency list is more feasible for this operation.

The above discussions demonstrate that the adjacency list is most suitable for representing a transportation network since it not only reduces the storage space in the main memory, but it also facilitates the routing computation.

Summary

Since transportation networks are a specialized type of graph, some fundamental knowledge of graph theory is required. Some basic concepts, such as the definition of a graph, degree of a graph, and the definition of a path, were introduced at the beginning of this chapter. In the discussion of the degree of a graph, the dense graph and sparse graph have been defined and used in data model discussion.

In the data model discussion, three types of data models for graph representation were given: the incidence matrix, adjacency matrix and adjacency list. The discussion includes a description of each model, an analysis of the space complexity, storage space requirements and an examination of suitable operations for each model. Based on the discussion, an adjacency list is regarded as the best representation of the transportation network considering its own characteristics. My research, will utilize an adjacency list to construct topology of the experimental road network in order to implement my routing computations.

3.3 Shortest Path Problems

The computation of shortest paths has been extensively researched since it is a fundamental issue in the analysis of transportation networks. There are many factors associated with shortest path algorithms. First, there is the type of graph on which an algorithm works - directed or undirected, real-valued or integer link costs, and possibly-negative or non-negative link-costs. Furthermore, there is the family of graphs on which an algorithm works - acyclic, planar, and connected. All of the shortest path algorithms presented in this thesis assume directed graphs with non-negative real-valued link costs.

3.3.1 Classification of Shortest Path (SP) Problems

Even though different researchers tend to group the types of shortest path problems in slightly different ways, one can discern, in general, between shortest paths that are calculated as one-to-one, one-to-all, or all-to-all.

Given a graph, one may need to find the shortest paths from a single starting node *v* to all other nodes in the graph. This is known as the single-source shortest path problem. As a result, all of the shortest paths from *v* to all other nodes form a shortest path tree covering every node in the graph. Another problem is to find all of the shortest paths between all pairs of nodes in the graph. This is known as the all-pairs shortest path problem. One way to solve the all-pairs shortest path problem is by solving the single-source shortest path problem from all possible source nodes in the graph. Dijkstra's algorithm is an efficient approach to solving the single-source shortest path problem on positively weighted directed graphs with real-valued link costs. Many of today's shortest path algorithms are based on Dijkstra's approach.

There is also the relatively simple single-pair shortest path problem, where the shortest path between a starting node and a destination node must be determined. In the worst case, this kind of problem is as difficult to solve as single-source.

3.4 Analysis of the Searching Strategy

3.4.1 Breadth-First Search

A Breadth-First search (BFS) is a method that traverses a graph touching all of the nodes reachable from a given source node. BFS starts at the source node, which is at level 0. In the first stage, it visits all of the nodes at level 1. In the second stage, it visits all of the nodes at level 2 that are adjacent to the nodes of level 1, and so on.

The BFS exhaustively searches the entire graph without considering the goal until it may finds it or terminates when every node has been visited. The BFS regards every link as having the same length and labels each node with a distance that is given in terms of the number of links from the start node. All child nodes obtained by expanding a node are added to a FIFO queue (First in, First out). In typical implementations, a container (e.g. linked list or queue) called "open" is used to store any nodes that have not yet been examined by the search algorithm. Once the nodes have been examined, they are placed in another container that is called "closed". A breadth-first search is described in Figure 3.4.

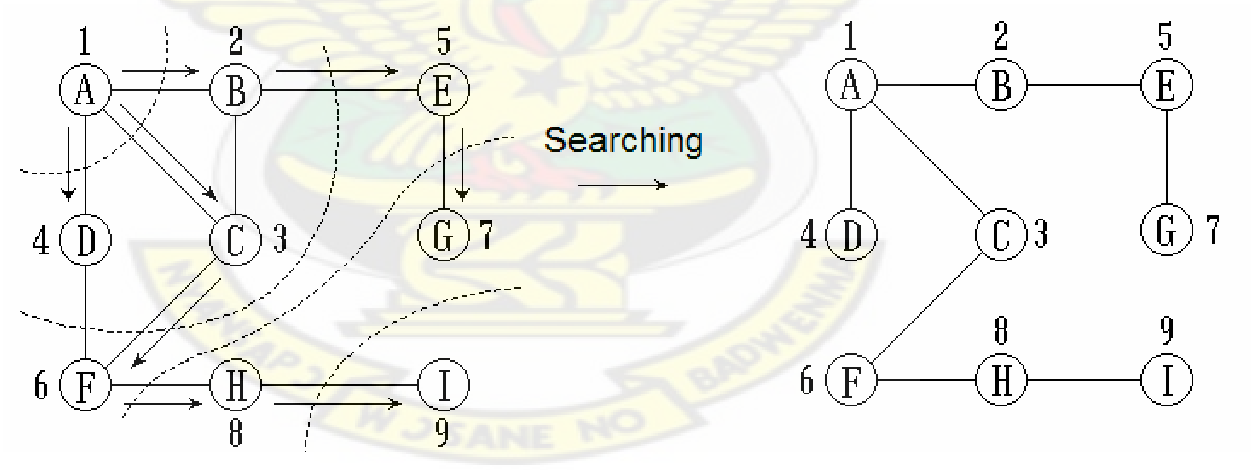


Figure 3.4. : Breadth-first Search

3.4.2 Depth-First Search

Depth-First Search (DFS) starts at a start node *S* in G, which then becomes the current node. The algorithm then traverses the graph by any link (*u*, *v*) incident to the current node *u*. If the link (*u*, *v*) leads to an already visited node *v*, then the search backtracks to the current node *u*. If, on the other hand, link (*u*, *v*) leads to an unvisited node *v*, the algorithm moves to *v* and *v* then becomes the current node. That is, it will pick the next adjacent unvisited node until it reaches a node that has no unvisited adjacent nodes. The search proceeds in this manner until it reaches a dead-end. At this point, the search starts backtracking and the process terminates when backtracking leads back to the start node. Figure 3.5 shows a DFS applied to an undirected graph, with the nodes labeled in the order they were explored.

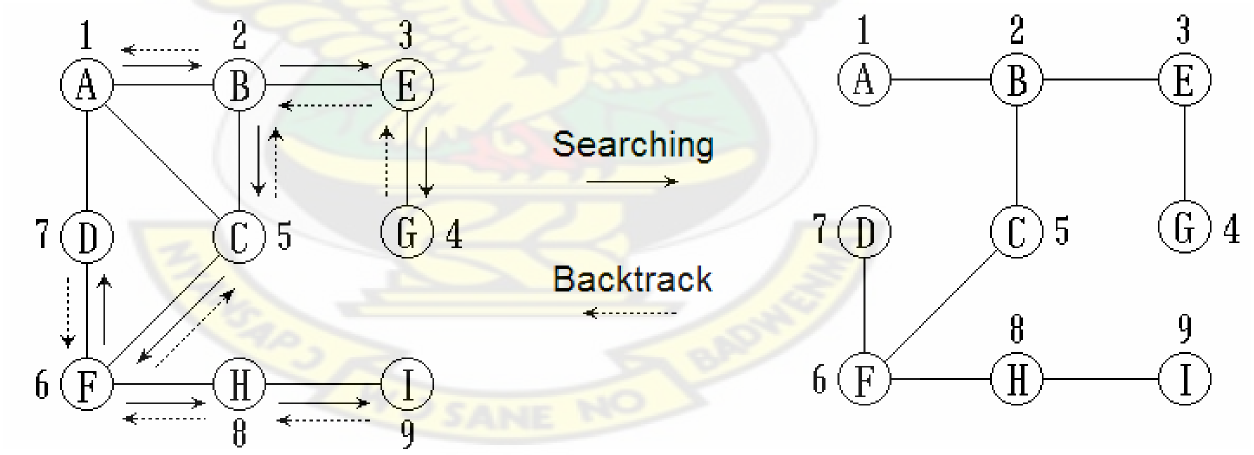


Figure 3.5: Depth-first Search

3.4.3 Best-First Search

The Breadth-First search is able to find a solution without getting trapped in dead-ends, while the depth-first algorithm finds a solution without computing all of the nodes. The

Best-First search allows us to switch between paths thus gaining the benefit of both approaches. It is a combination of DFS and BFS, which optimizes the search at each step by ordering all current adjacent nodes according to their priority as determined by a heuristic evaluation function. The search then expands the most promising node, which has the highest priority. If the current node generates adjacent nodes that are less promising, it is possible to choose another at the same level. In effect, the search changes from depth to breadth. The heuristic evaluation function predicts how close the end of the current path is to a solution. Those paths that the function determines to be close to a solution are given priority and are extended first.

A priority queue is typically used to order the paths for efficient selection of the best candidate for extension.

In summary, since the DFS and BFS exhaustively traverse the entire graph until they find the goal, they are categorized as uninformed searches. In contrast, the Best-First search utilizes a heuristic to reduce the search space and is able to find the goal more efficiently and is categorized as informed search.

3.5 Classical Shortest Path Algorithms for Static Networks

Path finding is applicable to many kinds of networks, such as roads, utilities, water, electricity, telecommunications and computer networks, the total number of algorithms that have been developed over the years is immense, depending only on the type of network involved. Labeling algorithms are the most popular and efficient algorithms for solving the SP problem. These algorithms utilize a label for each node that corresponds to the tentative shortest path length p_k to that node. The algorithm proceeds in such a way that these labels are updated until the shortest path is found.

Labeling algorithms can be divided into two sets: the label setting (LS) algorithms and label correcting (LC) algorithms. For each number of iteration, the LS algorithm permanently sets the label of a node as the actual shortest path from itself to the start node, thus increasing the shortest path vector by one component at each step. The LC algorithm does not permanently set any labels.

All of the components of the shortest path vector are obtained simultaneously; a label is set to an estimate of the shortest path from a given at each iteration. Once the algorithm terminates, a predecessor label is stored for each node, which represents the previous node in the shortest path to the current node. As a result, it only determines the path set, $P_{k=} \{p_1,\ldots, p_k\}$, in the last step of the algorithm. Backtracking is then used to construct the shortest paths to each node. Typical label setting algorithms include Dijkstra's algorithm and the A* algorithm. The Floyd-Warshall algorithm is an example of a label correcting algorithms.

3.5.1 Dijkstra's Algorithm

Dijkstra's algorithm, named after its inventor, has been influential in path computation research. It works by visiting nodes in the network starting with the object's start node and then iteratively examining the closest not-yet-examined node. It adds its successors to the set of nodes to be examined and thus divides the graph into two sets: *S*, the nodes whose shortest path to the start node is known and *S'*, the nodes whose shortest path to the start node is unknown. Initially, *S'* contains all of the nodes. Nodes are then moved from *S’* to *S* after examination and thus the node set, S, “grows”. At each step of the algorithm, the next node added to *S* is determined by a priority queue. The queue contains the nodes *S’*, prioritized by their distance label, which is the cost of the current shortest path to the start node. This distance is also known as the start distance. The node, *u*, at the top of the priority queue is then examined, added to *S*, and its out- links are relaxed. If the distance label of *u* plus the cost of the out- link *(u, v)* is less than the distance label for *v*, the estimated distance for node *v* is updated with this value. The algorithm then loops back and processes the next node at the top of the priority queue. The algorithm terminates when the goal is reached or the priority queue is empty. Dijkstra's algorithm can solve single source SP problems by computing the one-to-all shortest path trees from a source node to all other nodes. The pseudo-code of Dijkstra's algorithm is described below.

```
Function Dijkstra ($G$, $start$)
    1) $d[start] = 0$
    2)      $S = \varnothing$
    3)          $S' = V \in G$
    4)              while $S' \neq \varnothing$
    5)                  do $u = \mathrm{Min}(S')$
    6)                      $S = S \cup \{u\}$
    7)                      for each link $(u, v)$ outgoing from $u$
    8)                  do if $d[v] > d[u] + w(u, v)$ // Relax $(u, v)$
```

```
9)                          then $d[v] = d[u] + w(u, v)$
10)                 Previous[$v$] = $u$
```

3.5.2 A* algorithm

It is not feasible to use Dijkstra's algorithm to compute the shortest path from a single start node to a single destination since this algorithm does not apply any heuristics. It searches by expanding out equally in every direction and exploring a too large and unnecessary search area before the goal is found. Dijkstra's algorithm is a version of a BFS and although this algorithm is guaranteed to find the optimal path., it is not extensively applied due to its relatively high computing cost. This has led to the development of heuristic searches. In terms of heuristic searches, the A* algorithm is widely regarded as the most efficient method.

The A* algorithm is a heuristic variant of Dijkstra's algorithm, which applies the principle of artificial intelligence. Like Dijkstra's algorithm, the search space is divided into two sets: *S*, the nodes whose shortest path to the start node is known and *S'*, the nodes whose shortest path to the start node is unknown. It differs from Dijkstra's algorithm in that it does not only consider the distance between the examined node and the start node, but it also considers the distance between the examined node and the goal node.

In the A* algorithm, g (n) is called the start distance, which represents the cost of the path from the start node to any node n, and h(n) is estimated as the goal distance, which represents the heuristic estimated cost from node n to the goal. Because the path is not yet complete, we cannot actually know this value, and h (n) has to be “guessed”. This is where the heuristic method is applied.

In general, a search algorithm is called admissible if it is guaranteed to always find the shortest path from a start node to a goal node. If the heuristic employed by the A* algorithm never overestimates the cost, or distance, to the goal, it can be shown that the A* algorithm is admissible. The heuristic is called an admissible heuristic since it makes the A* search admissible.

If the heuristic estimate is given as zero, this algorithm will perform the same as Dijkstra's algorithm. Although it is often impractical to compute, the best possible heuristic is the actual minimal distance to the goal. An example of a practical admissible heuristic is the straight-line distance from the examined node to the goal in order to estimate how close it is to the goal.

The A* algorithm estimates two distances $g(n)$ and $h(n)$ in the search, ranks each node with the equation: $f(n) = g(n) + h(n)$, and always expands the node n that has the lowest f(n).Therefore, A* avoids considering directions with non-favorable results and the search direction can efficiently lead to the goal. In this way, the computation time is reduced. Thus, the A* algorithm is faster than Dijkstra's algorithm for finding the shortest path between single pair nodes. The algorithm is an example of a best-first search.

3.5.2 Comparison of Algorithms Based on Distance (Time) Complexity

The efficiency of a search algorithm is a critical issue in route planning since it relates to the practicality and effectiveness of the search algorithm. Since a time consuming search algorithm is inapplicable in real world applications, it is necessary to conduct a complexity analysis for different algorithms.

The complexity analysis involves two aspects: time and space complexity. Algorithm requirements for time and space are often contradictory with a saving on space often being the result of an increase in processing time, and vice versa. However, advances in computer hardware have made it possible to provide sufficient memory in most computational environments and the main concern is now the time complexity of the algorithm.

In shortest path computation, there are two essential operations: one is the additive computation which gives the start distance of the current node based on previous nodes and the link weight between them; the other is the comparison operation which gives a possible shorter path to the start node. We assume the time cost for these two operations is equivalent. The time complexity is measured by the frequency of the most used operations in the above algorithms.

Observing the pseudo-code of Dijkstra's algorithm in section 3.5.1, the main loop from steps 5 to 10 takes the most computational time. In step 5, the algorithm finds the node with a minimum start distance. It requires $|V|$ times comparison at first time, $|V| - 1$

times at second time and so on. Therefore the time complexity of the node search is $|V|+(|V|-1)+\ldots+1=O(|V|_2)$. In steps 8 to 10, the algorithm examines all links that are connected to the current node for the additive and comparison operations. From the view of the entire search, it will examine all of the links in the network, which takes $|E|$ time. Therefore the final time complexity of Dijkstra's algorithm is $O(|V|_2+|E|)=O(|V|_2)$.

For the A* algorithm, its time complexity is calculated in a different way since it only computes the shortest path between a single pair of nodes. If the average degree of a network is denoted as *d*, and the search depth (i.e., the levels traversed in searching the tree until the goal is found) is denoted as *h*, then the time complexity of the A* algorithm is $O(d^h)$. The time complexity comparison between these two algorithms is shown in Table 3.1.

	Dijkstra's Algorithm	A^* Algorithm
Time Complexity	$O(\|V\|^2)$	$O(d^h)$

Table 3.1 Time Complexity Comparison between Classical Algorithms

In this section, I suggest that the shortest path from the current location to a known destination is a typical query for navigation services. Based on the above time complexity comparison, A* is an efficient algorithm to solve the SP problem, because *d* and *h* are much smaller than $|V|$. Thus, the distance (time) complexity of the Dijkstra algorithms is far greater than A* in that they involve redundant computation for solving the single pair SP problem. Since they are more applicable to other shortest path problems, they may be employed in other discussed later in the thesis.

Although A* can answer the first type of query proposed in section 1.4, it is not the optimal solution as it is a static approach. In a dynamic environment, A* has to recompute the shortest path from scratch every time there is a change in traffic conditions. From this point of view, it must be improved in order to be adaptable to a dynamic environment.

3.6 Dynamic Traffic Routing

3.6.1 Dynamic Transportation Network

Time is an essential part of today’s world. While long distance travel time seems to be getting shorter each year, daily commuters have to spend more and more time just getting to their offices. A major reason for this situation is traffic congestion, which results from high traffic flow, incidents, events or road construction. Traffic congestion is perhaps the most conspicuous problem in the transportation network and has become a crucial issue that needs immediate attention.

In the past, when drivers encountered traffic congestion, they had to queue up and wait until the congestion cleared. Analysts were content with just studying the queuing times and predicting waiting times, without making any attempt to actually solve the problem. Current countermeasures for traffic congestion are oriented toward a "local" optimum, i.e., a point-to-point diversion by using sign boards to divert traffic flow around the point of congestion. The emergence of LBS gives a new paradigm for applying GIS to transportation issues. As a key component, navigation services are regarded as the most promising solution for solving this problem.

In transportation network representations, the weight of the links can be assigned as the cost of travel time, along the links. Changes in traffic conditions are considered as changes in link-weights, where the congestion occurs. Since traffic conditions always change over time, the centralized navigation service has to monitor the traffic fluctuations over a day-long interval and detect any congestion upstream in order to allow drivers to take preventive action. By using dynamic shortest path algorithms, navigation services can also help mobile clients to plan an alternative optimal route to their destination based on the updated traffic conditions. In this sense, the solution provided by the navigation service is closer to a "global" optimum. This feature also encourages the possibility of deploying these algorithms in real-time traffic routing software.

3.6.2 Related Research for Dynamic Traffic Routing

Recent developments in LBS reflect a propensity for increased use of dynamic algorithms for routing. Most of these algorithms have already been applied successfully for routing in computer networks. As well, these algorithms can be applied to transportation network management, especially in the context of the centralized architecture of navigation services, where traffic flow would exhibit a behavior close to that of “packets” in computer networks.

Motivated by theoretical as well as practical applications, many studies have examined the dynamic maintenance of shortest paths in networks with positive link weights, aiming at bridging the gap between theoretical algorithm results and their implementation and practical evaluation.

In dynamic transportation networks, weight changes can be classified as either deterministic or stochastic time-dependent. In the deterministic time-dependent shortest path (TDSP) problem, the link-weight functions are deterministically dependent on arrival times at the tail node of the link, i.e., with a probability of one. In the stochastic TDSP problem, the link-weight is a time-dependent random variable and is modeled using probability density functions and time-dependency. Here, link weights take on time-dependent values based on finite probability values. Cooke and Halsey first proposed a TDSP algorithm in 1958. The algorithm they suggested is a modified form of Bellman's label correcting the shortest path algorithm. Hall worked on the stochastic TDSP problem and showed that one cannot simply set each link-weight random variable to its expected value at each time interval and solve an equivalent TDSP problem. Frank derived a closed form solution for the probability distribution function of the minimum path travel time through a stochastic time-variant network. There were also a number of other works addressing similar problems. All of these are based on the model of a time-dependent network where link length or link travel time is dependent on the time interval.

All of the research discussed above attempts to use probabilistic and statistical approaches to determine the random change of link-weights and then derive the most promising shortest path. To simplify the dynamic shortest path (DSP) problem, my thesis research assumes that the link-weight changes are collected and updated by a centralized navigation service. Based on the given link-weights for each time interval, my research focuses on the DSP algorithm itself. The DSP algorithm utilizes current traffic conditions to dynamically maintain the optimal path en route.

With a single weight change, usually only a small portion of the graph is affected. For this reason, it is sensible to avoid computing the shortest path from scratch, but only to update the portion of the graph that is affected by the link-weight change.

Incremental search methods are used to solve dynamic shortest path problems, where shortest paths have to be determined repeatedly as the topology of a graph or its link costs change. A number of incremental search methods have been suggested in the literature for algorithm, which differ in their assumptions: whether they solve single-source or all-pairs shortest path problems; which performance measure they use, when they update the shortest paths; which kinds of graph topology and link costs they apply to; and how the graph topology and link costs are allowed to change over time. An algorithm is referred to as fully-dynamic if both the weight increment and decrement are supported and semi-dynamic if only the weight increment (or decrement) is supported.

Among the algorithms proposed for the DSP problem, the algorithm of Ramalingam and Reps (RR for short, also referred to as the DynamicSWSF-FX algorithm) seems to be the most used. It is a fully-dynamic DSP algorithm which updates the shortest paths incrementally.

In their work on algorithms for the DSP problem. Proposed a fully dynamic algorithm, which is a specialization of the RR algorithm for updating a shortest path tree. It is a modification of their previous work on a semi-dynamic incremental algorithm.

This chapter shows that the RR algorithm is an efficient approach for solving the DSP problem. One of its main advantages is that the algorithm performs efficiently in most situations. First of all, it updates a shortest path graph instead of a shortest path tree, although it can be easily specialized for updating a tree. Even and Shiloach proposed a semi-dynamic incremental algorithm that works in cascades, which can be computationally expensive for large link-weight increments. RR has good performance independent of weight increments. For updating a shortest path tree, Demetrescu's semi-dynamic incremental algorithm performs well only if most of the affected nodes have no alternative shortest paths. However, the RR algorithm performs well even when there are alternative paths available. Even the algorithm of Frigioni et al.,(1996) which is theoretically better than RR, was usually outperformed by RR in computational testing.

Many theoretical studies of DSP algorithms have been carried out but few experimental results are known. Frigioni et al., (1998) compared the RR algorithm with the algorithm proposed by Frigioni et al., for updating a single-source shortest path graph. They concluded that the RR algorithm is usually better in practice, with respect to running times, but their algorithm has a better worst case time complexity.

3.6.3 Incremental Approach – RR Algorithm

In dynamic transportation networks, only portions of links change their weight between each update. The start distances for some nodes stay the same as before and thus do not need to be recomputed. This suggests that a complete re-computation of the optimal route can be wasteful since some of the previous search results can be reused. Incremental search methods, such as the RR algorithm, reuse information from previous searches to find shortest paths for series of similar path-planning problems potentially faster than is possible by solving each path-planning problem from scratch.

The problem with reusing previous search results is how to determine which start distances are affected by the cost update operation and need to get recomputed.

Assume S denotes the finite set of nodes of the graph and *succ* $(s) \subseteq S$ denotes the set of successors of node $s \in S$. Similarly, *pred* $(s) \subseteq S$ denotes the set of predecessors of node $s \in S$. In this case, $0 < w(s, s') \leq \infty$ denotes the cost of moving from node s to node $s' \in$ *succ* (s) and $g(s)$ denotes the start distance of node $s \in S$, that is, the cost of a shortest path from s to its corresponding start node.

There are two estimates held by the RR algorithm in its lifetime. The first one is the *g(s)* of node *s* which directly corresponds to the start distance in Dijkstra's algorithm.
It can be carried forward and reused from search to search. The second is another estimate of the start distances, namely the *rhs*-value which is a one-step look-ahead

value based on the *g*-value and thus is potentially better informed than the *g*-value. Its name comes from the RR algorithm where it is the value of the right-hand side (rhs) of the grammar rules. It always satisfies the following relationship:

$$rhs = \begin{cases} 0 & if s = Sstart \\ Min s' \in pred(s)(g(s) + w(s, s')) & otherwise \end{cases} \quad \textbf{(3.6.1)}$$

A new concept needs to be defined, called local consistency. A node is locally consistent if its *g*-value equals its *rhs*-value. This concept is important because a local consistency check can be used to avoid node re-expansion. Moreover, the *g*-values of all nodes are equal to their start distances if all nodes are locally consistent. Whenever link costs are updated, the *g*-value of the affected nodes will be changed. The nodes become locally inconsistent. The RR algorithm maintains a priority queue that always exactly contains the locally inconsistent nodes. These are the nodes whose *g*-value potentially needs to be updated in order to make them locally consistent. In this way, the shortest path tree can be adjusted dynamically.

Summary

In this chapter, the shortest path problem is well discussed. The chapter started with the classification of the shortest path problem, which divided the shortest paths into one-to-one, one-to-all, or all-to-all.

Commonly used search strategies, such as the breadth-first, depth-first and best-first searches, were then introduced. Based on the search strategy analysis, two classical shortest path algorithms are described as typical solutions to the shortest path problems defined by the classification. They are Dijkstra's and the A* algorithms, which are devised for static environments. Although the time complexity comparison demonstrates that the A* algorithm is most suitable for calculating the shortest path between single pair nodes due to its static property. The algorithm is inefficient in dynamic transportation networks.

To satisfy the requirement of applications for real-world traffic networks, the dynamic shortest path (DSP) problem is addressed. Firstly, the scenario of the dynamic traffic network is provided to illustrate the past and present solutions in the real-world and demonstrate the importance of DSP research. Secondly, some related research on the time-dependent shortest path (TDSP) problem is briefly introduced in order to identify the research area in this thesis, which assumes the link-weight changes have been given. Based on this assumption, some previous algorithms are explored. Among them, the RR algorithm is shown to be the efficient approach in most dynamic environments. It plays a major role in my solution to the DSP problem. Nevertheless, all of the dynamic approaches discussed in this chapter are still not capable of answering the first query type proposed at the beginning of this thesis, i.e., trying to find the adaptive route from the current location to a known destination. These algorithms can only calculate the dynamic shortest path between fixed start and goal nodes for different time intervals. This means that they are not able to deal with changes in the position of the start node as a mobile user moves along the initial optimal path and makes an en route query for a new shortest path in accordance with traffic condition changes.

CHAPTER 4

NEW IDEAS

4.1 Shortest path and the environment issues

Suppose there is a need to find a path, which implies the smallest usage of fuel. This case is similar as the money cost of the travel, but it differs since the bus money cost (the money for a bus ticket, for example) is higher than (and not linear to) the fuel used. The shortest path in terms of used fuel needs to be evaluated from the environment point of view.

The simplest approach to the problem is to ascribe a cost to every link costs that express the impact on the environment. A higher cost will be attached to a car link, and a smaller cost will be attached to the bus link. The cost of a link should be dependent on the length of s link. According to the criteria of cost, the algorithm searches for the shortest path and at the same time computes the time cost.

The time cost of a shortest path generated with the help of such an algorithm will not be optimised. We can conceive the case where there is the shortest path found in terms of the lowest fuel cost, but the time cost is not acceptable. This may happen is we waited a long time to save not a significant amount of fuel.

A number of constraining criteria for such a shortest route finding can be given. First, we can fix a certain amount of time which can be taken at most for waiting at a bus stop.

Among the links that fulfill this condition, the link of the smallest fuel cost is chosen. Another constraint can be that the overall travel time cannot be greater than a fixed amount T. In the article by Cai at el., (1997) one can find three algorithms for the internet data packages routing among, which one algorithm is very well suited to our needs. The algorithm searches a specific type of networks. Each link in the network has two numbers ascribed: cost and time. For us the cost can be the fuel cost and time is the time cost.

The proposed algorithm is going to find the shortest route according to fuel cost and with the overall time cost not exceeding a specific amount of time T. The main ideas of the project are involved in the adaptation of the algorithm proposed by Dreyfus (1969) for bus networks which is based on the Dijkstra's algorithm.

The main ideas have been used to adopt the algorithm described by Dreyfus (1969) to the public transportation networks, to describe it mathematically.

4.2 Introduction to the bus routing algorithm description

The algorithm can be used for any public transportation network based on timetables. In any public transports means that the project takes into account, the root for the public transport which must based on timetables in which the driver reports the bus routing algorithm.

4.3 The bus routing algorithm

To understand the problem clearly, it is useful to visualise a traveler who wants to get from one bus stop to another in a city using buses only. The input data to the algorithm consists of a description of the bus transportation network (timetables, description of connections between bus stops), the bus stop where the journey begins (the source node) and the bus stop at which the journey ends (the destination node). The objective is to find the shortest path between the two specified nodes, namely the path that requires the minimal amount of time.

The algorithm presented here was designed to solve the problem described above. The new algorithm had to be designed in order to meet the special needs of bus transportation networks such as timetables and the possibility of waiting at bus stops. The main difference between the standard shortest path problem and this one is that links vary with distance (time) and it is allowed to wait at the nodes as long as it is necessary to obtain the minimal time cost. The problem can be classified as the shortest path problem with time dependent costs of links and the allowance of waiting at nodes. A time cost of every link may differ in any desired way.

The solution to the problem is based on Dijkstra's algorithm, which is the best known algorithm for directed networks with nonnegative link costs. There are several principles (as the use of a priority queue or the use of buckets) underlying an efficient implementation of the Dijkstra algorithm which can also be applied to implement the bus algorithm (the article by Cherkassy et al is a good paper discussing principles for Dijkstra algorithm implementation).

The Dijkstra algorithm has to be modified because of two problems.

The first modification deals with a problem of fixed times at which a bus leaves the bus stop. This new attribute of a link is going to be named the departure time of a link.

Dijkstra's algorithm is not concerned with a departure time of a link; the algorithm was designed to work with links, which can be used at any time. In our problem the main constraint is that links cannot be used at any time, the time at which a bus link can be used is fixed according to a timetable.

The second problem is the actual cost of a bus connection between two nodes. In our case the cost of a link is not the criterion to judge the optimality of the link choice anymore. In the present problem the actual criterion is the sum of the waiting time and the link cost, or, in other words, the time of arrival at the finishing node of a link. In effect, we are also concerned with the waiting times at nodes. The need to wait at bus stops is a consequence of the departure time attribute (if a link cannot be used right now then it is necessary to wait for the departure).

Suppose there are buses leaving a specific bus stop at 1, 2, …, 10 time units and arriving at the other specific bus stop after the time cost. The time costs of the buses differ considerably since the buses may be of different companies and they may take different routes. Having the data, the task is to find the cheapest connection between two nodes. The task then is to find the link that has the minimum sum of the time cost of a link and the waiting time (that is necessary to wait for this link).

We can phrase the solution to the problem in this way: the sought link is the link of which the arrival time is minimal. This formulation of the solution, i.e. finding the minimal arrival time, is going to be used as opposed to the summation of a waiting time and the time cost (which is the same but complicates the coming formulas).

4.4 The model of a bus transportation network

We need to represent a bus transportation network to model time tables. Therefore we are going to put a link into the directed graph to represent a connection between two bus stops. We need to associate a bus departure time from a bus stop, this departure time represent one entry in a timetable. We are also going to need the time of arrival time to the next bus stop (this can be taken from the bus table of the next node). Each entry in a timetable will be represented by one link and therefore we need to operate on sets, not on matrices.

Definitions

We introduce new definitions and concepts to model the bus network conveniently and to describe the algorithm precisely.

Definition of the network

We are given a directed graph $G (V, E, d, a)$, where V is the set of nodes (Vertices) and E is the set of links (Edges). We denote the number of element of the set V of nodes as $n = |V|$ and the number of links of the set of E as m = |E|. We will refer to a node of a graph by '*xj*'*and* to a link by '*e*'. Function $d : E \langle R*\{\infty\}$ (the infinity can be ascribed to a link, therefore it is included) associates with every link a departure time (a real nonnegative number) from the staring node and the function $a : E \langle R*\{\infty\}$ ascribes the time of arrival (a real nonnegative number) at the finishing node. The departure time $d(xj)$ of the xj link (from the staring node) is the time at which the link may be used, it cannot be in use at any other time. The arrival time $a(x_j)$ of the x_j link is the time at which we get to the finishing node of the link.

If it is necessary to know the time cost of the x_j link (time required to traverse the link), we can calculate it in this way: $c(x_j) = a(x_j) - d(x_j)$. We can describe uniquely a link of the graph only by four numbers $(x_i, x_i+1, d(x_i), a(x_i))$ where *xi* is a starting node of the link, $x_i+1$is the finishing node of the link, $d(x_i)$ is the departure time and $a(x_i)$ is the arrival time.

Referring to a specific link only by a pair (x_i, x_i+1) can be ambiguous since there can be two or more links between the x_i *and* x_i+1 nodes, links may differ in the departure time or the arrival time.

The $\mathcal{P}$ Symbol

The symbol X represents a powerset of a X set (i.e. the set of all subsets of the X set).

Definition of the *slbtn* function

s*lbtn: V×V* ɑ$\mathcal{P}$*E*

Set of Links Between Two Nodes

The *slbtn* function takes a pair of nodes (x_i, x_i+1) and returns the set which comprises every link that starts at *xi* node and finishes at the *xi+1* node.

Definition of the *slcsn* function

slcsn: V ɑ $\mathcal{P}$*E*

Set of Links of Common Starting Node

The function returns all the links that have the node given as the argument as the starting node.

Definition of the arrival time at a specific node

T (x_i) is the time of arrival at the node xi. Let *t0* be the time at which the journey starts. We define $T(x_i)$ recursively.

$$T(s) = t_o$$

$$T(x_{i+1}) = \min_{\{e \in \boldsymbol{slbtn}(x_i, x_{i+1}) \,|\, d(e) \geq T(x_i)\}} a(\mathbf{e})$$

Note that $T(x_i)$ is not the cost of the shortest path. It represents the time at which the node is reached. The cost of getting to a node let's say x_{17} may be only ten minutes, but $T(x_{17}) = 17{:}10$ if we started the journey at 17:00.

The path

Let $P = (s = x_1, x_2, \ldots, x = x_r)$ be a path form the *s* node to the *x* node. The s node is the source node of the path and the x node is the destination node of the path.

The shortest path

Let $P = (s = x_1, x_2, \ldots, x = x_r)$ be the shortest path from the source node s to the destination node *x*. The nodes of the shortest path are such that they result in the minimal arrival time to the x_r node. The time of arrival is given by $T(x_r)$.

The subsequent nodes of the shortest path are found by the use of $T(x_i)$ function. To find x_{r-1} we have to find the link which led to x_r with minimal arrival time. Having the link, we have the starting node of the link. This starting node is the x_{r-1} node of the shortest path. This method has to be repeated until the source node is reached.

4.5 Step description of the algorithm

The aim of the algorithm is to minimise *T(x)* (*x* is the destination node). To minimise it we first have to minimise *T(x*$_i$*)* for nodes *x*$_i$ which are at the shortest path from s to x.

Before we get to the algorithm description, there are some definitions to be introduced.

We classify all the nodes of the graph into three sets, every node can be a member of only one of the following sets:

- SPN: the Set of Permanent Nodes is the set of nodes which have been completely processed; the time of reaching these nodes has been computed and will not change.
- SSN: the Set of Scanned Nodes is the set of nodes which have been reached, but have not been completely processed; the time cost of getting to them is known but may change.
- SNRN: the Set of Not Reached Nodes is the set of nodes which have not been reached at all.

The step description

Step 0

The source node *s* is initialised as *scanned* (*s* ∈ *SSN*) and every other node *xi* $\bar{s}$ of the graph is initialised as *not reached* (*xi* ∈*SNRN*). Furthermore, the arrival time of the source node is set to *t0* (*t0* is the time at which the journey starts), i.e. *T(s)* = *t0*, and the arrival time of every other node *xi* $\bar{s}$ of the graph is set to infinity, i.e. *T (xi)* = ∞.

Step k

We process only one node during this step. We choose the node to be processed from the SSN (Set of Scanned Nodes). If the SSN is empty, this means there is no path between the source node and the destination nodes and the algorithm quits. If the SSN is not empty, we choose a x_i node from the SSN which has minimal $T(x_i)$. If there is more than one node with the minimal $T(x_i)$ then we choose one of them arbitrarily. In formula:

$$X_i \in SSN \bullet T(x_i) = \min_{x_i \in SSN} T(x_j)$$

Once the x_i node is chosen, we proceed to process it. First the node x_i is excluded from the SSN and becomes a member of SPN (Set of Permanent Nodes). At this stage it is certain that the arrival time $T(x_i)$ is minimal (it may only get larger since taking another link will increase the cost; link costs are always positive) and this is the reason for moving the x_i node to SPN.

Next the links leaving the x_i node are handled. From the set E (the set of links of the graph) every link which has the x_i node as a starting one is selected. The retrieved set is further constrained to links of the departure time greater or equal to $T(x_i)$ (only buses that will arrive to a bus stop can be taken, not those which have left).

CHAPTER 5

5.0 DATA COLLECTION AND ANALYSIS

This thesis offers an application solution for dynamic routing of vehicles in Adum. It proposes a routing system that users employs historical traffic data to model recurring congestion and compute initial shortest path. As unpredicted (nonrecurring) congestion occurs and is reported from some FM station or traffic control center, the system analyses the real time data to determine if the planned route needs to be change (modified). It can changed the planned route as a function of the current position, destination location, and real time traffic condition

The proposed routing system has been composed of three subsystems including ArcGIS Network Analyst (for the digitized map), Dijkstra's algorithm and VB.Net for the software development.

The routing optimization problem in traffic management has been already explored with a number of algorithms. Routing algorithms use a standard of measurement called a metric (i.e. path length) to determine the optimal route or path to a specified destination. Optimal routes are determined by comparing metrics, and these metrics can differ depending on the design of the routing algorithm used (Parker, 2001).

Different kinds of algorithms have been proposed finding the optimal routes, such as:

- Simulated Annealing is a related global optimization technique which traverses the search space by generating neighboring solutions of the current solution (Kirkpatrick et al., 1983).

- Tabu Search is similar to Simulated Annealing, in that both traverse the solution space by testing mutations of an individual solution. While simulated annealing generates only one mutated solution, tabu search generates many mutated solutions and moves to the solution with the lowest fitness of those generated (Glover et al., 1997).

- Genetic algorithms (Holland, 1975) use biological methods such as reproduction, crossover, and mutation to quickly search for solutions to complex problems. Genetic algorithm begins with a random set of possible solutions. In each step, a fixed number of the better current solutions are saved and they are used to the next step to generate new solutions using genetic operators.

- The ant colony optimization algorithm which has been used to produce near-optimal solutions to the travelling salesman problem. They have an advantage over simulated annealing and genetic algorithm approaches when the graph may change dynamically (Dorigo et al., 1999).

- Dijkstra's algorithm, used by Network Analyst, is a greedy algorithm that solves the single-source shortest path problem for a directed graph with nonnegative edge weights (Dijkstra, 1959).

However, Network Analyst is still relatively new software, so there is not much published material concerning its application traffic management.

Miller (2005) compares the RouteSmart 4.40, the ArcLogistics Route and the ArcMap Network Analyst extension on the ability of either software package to create routes usable by the Drivers in Adum, efficient manner for the city of Kumasi in Ashanti Region

Dijkstra's Algorithm, introduced in 1959 provides one the most efficient algorithms for solving the shortest-path problem. In a network, it is frequently desired to find the shortest path between two nodes. The weights attached to the edges can be used to represent quantities such as distances, costs or times. In general, if we wish to find the minimum distance from one given node of a network, called the source node or start node, to all the nodes of the network, Dijkstra's algorithm is one of the most efficient techniques to implement. In general, the distance along a path is the sum of the weights of that path. The minimum distance from node ***a*** to ***b*** is the minimum of the distance of any path from node ***a*** to ***b***.

5.1 NETWORK ANALYST

ArcGIS Network Analyst is a powerful extension that provides network-based spatial analysis including routing, travel directions, closest facility, and service area analysis. ArcGIS Network Analyst enables users to dynamically model realistic network conditions, including turn restrictions, speed limits, height restrictions, and traffic conditions at different times of the day (ESRI 2006). The users with Network Analyst extension are able to:

- Find efficient travel routes,
- Determine which facility or vehicle is closest,
- Generate travel directions, and
- Find a service area around a site.

In the current work, using Network Analyst, an optimum route for the routine find in particular area is generated in the area under study. Network Analyst uses the Dijkstra's Algorithm (Dijkstra 1959) in order to solve the Routing Problem and it can be generated based on two criteria (Lakshumi et al 2006):

1. Distance criteria: The route is generated taking only into consideration the location of the waste large items. The volume of traffic in the roads is not considered in this case.

2. Time criteria: The total travel time in each road segment should be considered as the: Total travel time in the route = runtime of the vehicle + distance time. The runtime of the vehicle is calculated by considering the length of the road and the speed of the vehicle in each road.

The Network Analyst extension allows the user to perform ”Find Best Route”, which solves a network problem by finding the least cost impedance path on the network from one stop to one or more stops. Network modeling gives the opportunity to the user to include the rules relating to the objects, arcs and events in association with solving transportation problems (Stewart 2004).

Dijkstra’s Algorithm

```
function Dijkstra (Graph, source):
       for each vertex v in Graph:              // Initializations
           dist[v]:= infinity                  // Unknown distance
function from
                                                  source to v
           previous[v] := undefined             // Previous node in
optimal path
                                                   from source
       dist[source] := 0                        // Distance from source to
                                                    source
       Q := the set of all nodes in Graph
        // All nodes in the graph are unoptimized - thus are in Q
       while Q is not empty:                    // The main loop
           u := vertex in Q with smallest dist[]
           if dist[u] = infinity:
                break                            // all remaining vertices
are
                                                   inaccessible
            remove u from Q
            for each neighbor v of u:            // where v has not yet
been
                                                          removed from Q.
               alt := dist[u] + dist_between(u, v)
               if alt < dist[v]:                 // Relax (u,v,a)
                   dist[v] := alt
                   previous[v] := u
        return previous[]
```

5.2 THE PATH FINDING ALGORITHM

Network Analyst software determines the best route by using an algorithm which finds the shortest path, developed by Edgar Dijkstra (1959). Dijkstra’s algorithm is the simplest path finding algorithm, even though these days a lot of other algorithms have been developed. Dijkstra’s algorithm reduces the amount of computational time and power needed to find the optimal path. The algorithm strikes a balance by calculating a path which is close to the optimal path that is computationally manageable (Olivera, 2002). The algorithm breaks the network into nodes (where lines join, start or end) and the paths between such nodes are represented by lines.

In addition, each line has an associated cost representing the cost (length) of each line in order to reach a node. There are many possible paths between the origin and destination, but the path calculated depends on which nodes are visited and in which order. The idea is that, each time the node, to be visited next, is selected after a sequence of comparative iterations, during which, each candidate-node is compared with others in terms of cost (Stewart, 2004).

The following comprehensible example, which is an application of the algorithm on a case of 6 nodes connected by directed lines with assigned costs, explains the number of steps between each of the iteration of the algorithm (Figure 5.1). The shortest path from node 1 to the other nodes can be found by tracing back predecessors (bold arrows), while the path's cost is noted above the node.

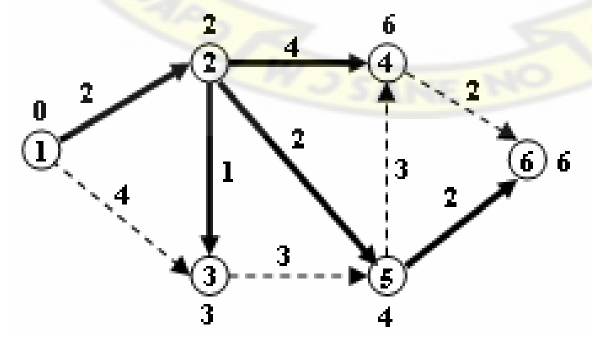


Figure 5.1: An example of Dijkstra's algorithm (Orlin 2003).

Each node is processed exactly once according to an order that is being specified below. Node 1 (i.e. origin node) is processed first. A record of the nodes that were processed is kept; call it Queue (Table 1). So initially Queue = {1}. When node k is processed the following task is performed: If the path's cost from the origin node to j could be improved including the vertex (k,j) in the path then, an update follows both of Distance[j] with the new cost and Predecessors[j] with k, where j is any of the unprocessed nodes and Distance[] is the path's cost from the origin node to j. The next node to be processed is the one with the minimum Distance [], in other words is the nearest to the origin node among all the nodes that are yet to be processed. The shortest route is found by tracing back predecessors.

		Distance						Predecessors				
Queue	Next node	1	2	3	4	5	6	2	3	4	5	6
1	2	-	2	4	∞	∞	∞					
1,2	3	-	-	3	6	4	∞		2	2	2	
1,2,3	4	-	-	-	6	4	∞					
1,2,3,5	5	-	-	-	6	-	6					5
1,2,3,5,4	6	-	-	-	-	-	6					
1,2,3,5,4,6	-	-	-	-	-	-	-					

Table 5.1: A record, called Queue, with all processed nodes

Network Analyst can be very useful in a variety of sections (ESRI 2006) in our daily life, such as in:

- Business, scheduling deliveries and installations while including time window restrictions, or calculating drive time to determine customer base, taking into account rush hour versus midday traffic volumes.
- Education, generating school bus routes honoring curb approach and no U-turn rules.
- Environmental Health, determining effective routes for county health inspectors.
- Public Safety, routing emergency response crews to incidents, or calculating drive time for first responder planning.
- Public Works, determining the optimal route for point-to-point pickups of massive trash items or routing of repair crews.
- Retail, finding the closest store based on a customer's location including the ability to return the closest ranked by distance.
- Transportation, calculating accessibility for mass transit systems by using a complex network dataset.

5.3 CASE STUDY

A digital road network in small area of Kumasi (Adum), capital of Ashanti Region, was used within the GIS map at a scale of about 1:2000. The road network was represented as connections of the nodes and links. Geometric networks are built in the ArcGIS model to construct and maintain topological connectivity for the road data in order to allow the path finding analysis to be possible. To plan the initial shortest path, use historical data of average traffic volume at surface streets or freeway segments within the area under study. The segment lengths have been extracted using ESRI's ArcGIS software. The average volume of each link in the network has been from obtained from KMA Traffic Unit. Summation of the travel distance (times) for all the segment of a particular path between origin and destination provides the total distance (time), which is minimized by the shortest path algorithm. The routing macro uses Dijkstra's routing algorithm.

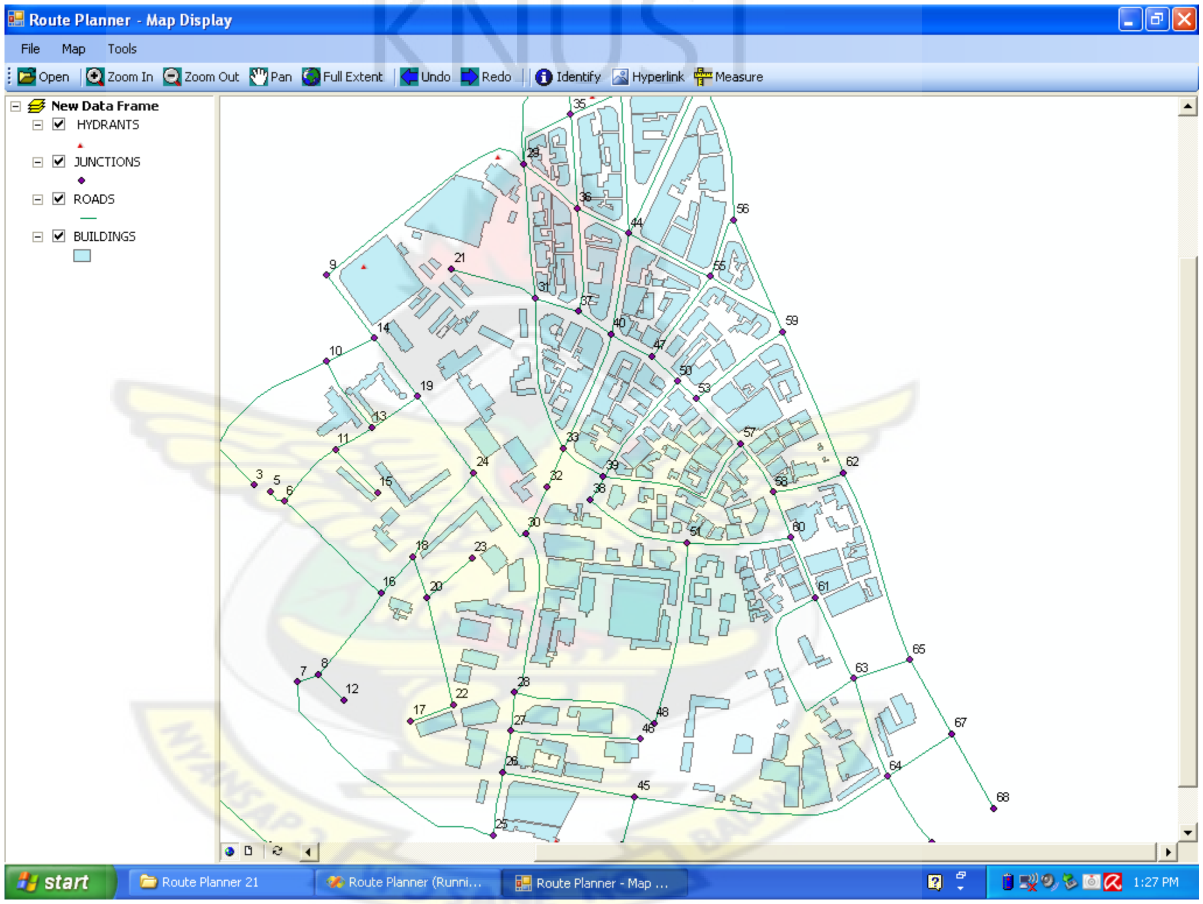


Figure 5. 2: Shows the map of Adum.

5.4 PROPOSED SOLUTION

This thesis describes a study of planning vehicle routes for the shortest path in a district of Adum using Network Analyst - a user-friendly extension of ArcGIS and Visual Basis

Dot Net with Dijkstra's algorithm, which provides efficient routing solutions in a simple and straightforward manner. In order to simulate the situation in ArcGIS, all the relevant information was acquired from KMA. More precisely, when creating a network routing solution, specific spatial data are needed for the accurate completion of the network. For example, a complete road network, where all the roads within the network are connected, is significant because it allows connection throughout the system.

MODEL ASSUMPTIONS

- Traffic congestion not considered
- Calculation based on road distance
- State of the road not considered

Adum map was taken from the Town and Country Planning Department of KMA.

Digitized by the Geodetic Department (KNUST) to convert the map into a road network

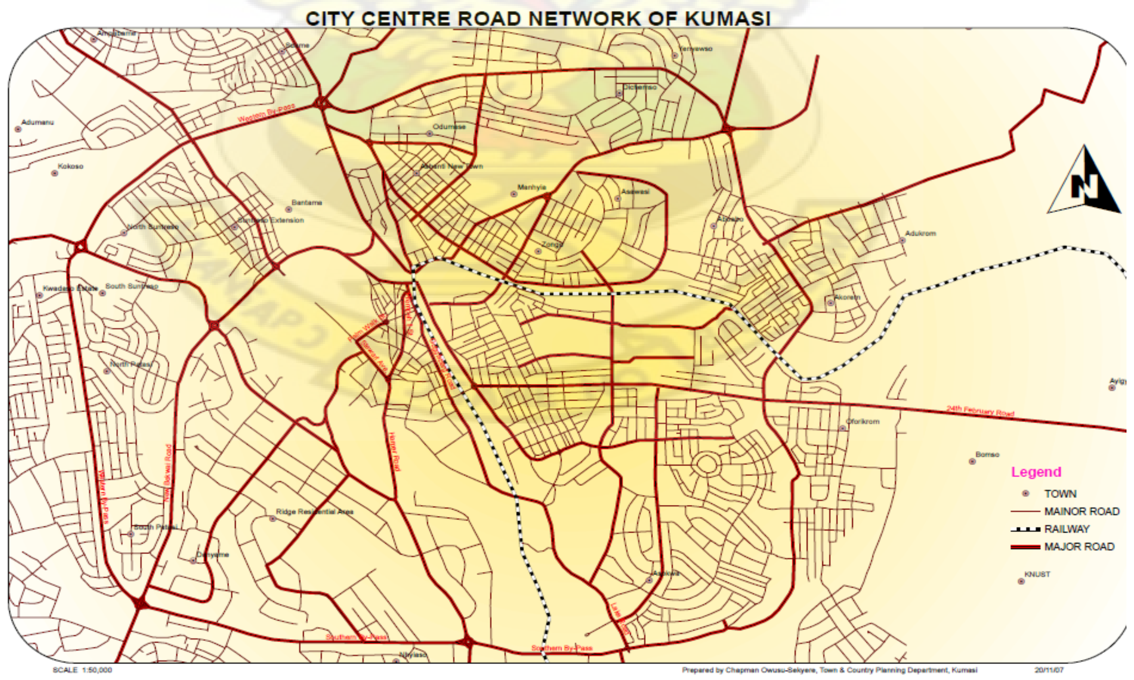


Figure 5. 3: Shows the City Centre Road Network of Kumasi .

EXTRACT MAP OF ADUM NETWORK

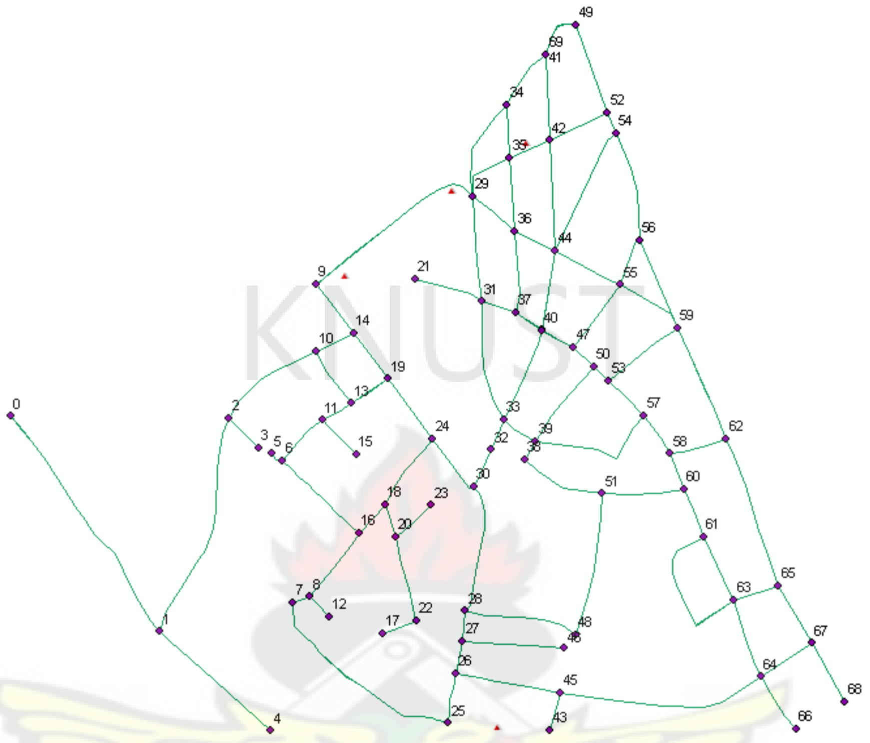


Figure 5.4: Shows the Extract Map of Adum Network

SOFTWARE DEVELOPMENT

The proposed routing system has been of three subsystems including:

- ArcGIS Network Analyst
- Dijkstra's Algorithm
- VB.Net

For the software development

FEATURES OF THE INTERFACE

Step one: The first interface of the program.

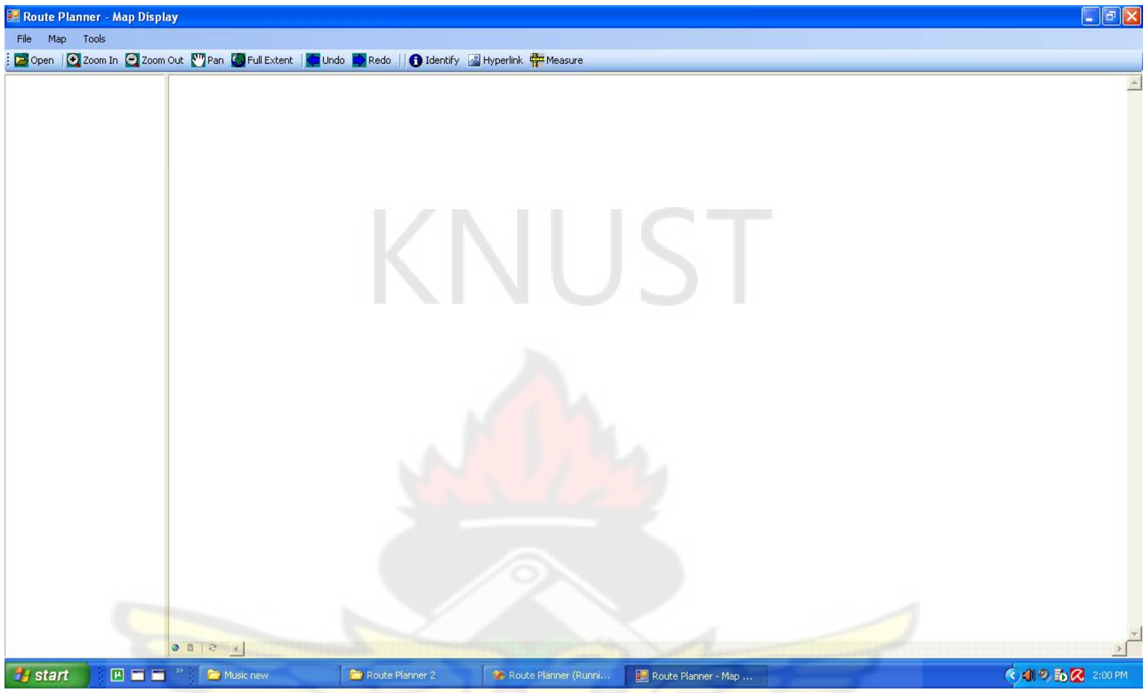


Figure 5.5: Shows the first interface of the program

Step two: The user open to select to select the map needed.

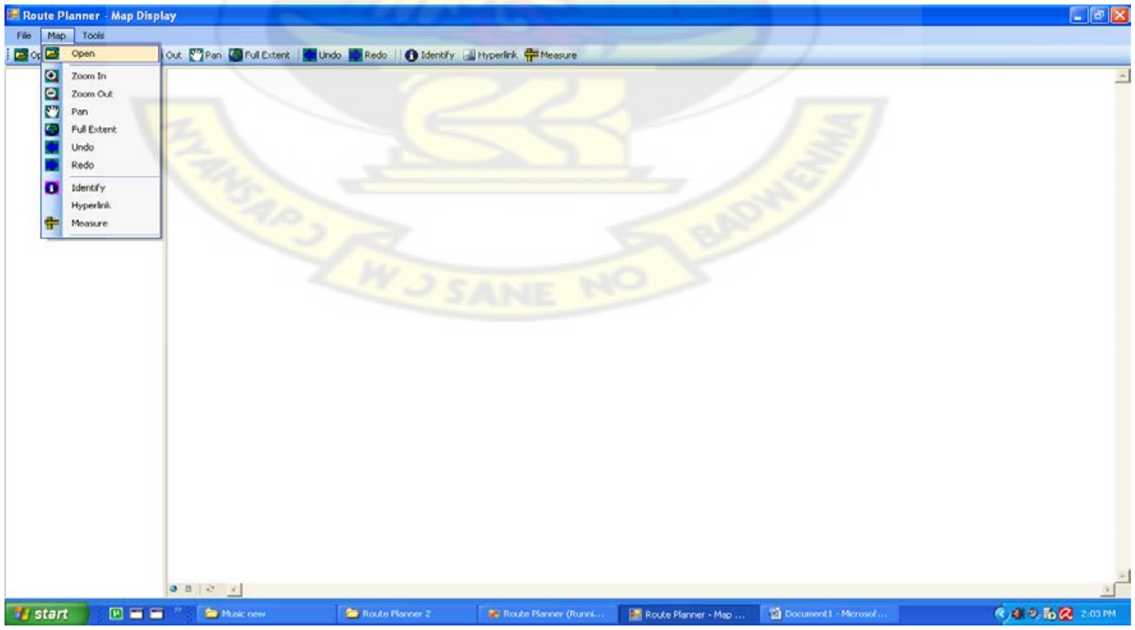


Figure 5.6: Shows the how users select a map

Step Three: The user selects the needed map from the dialogue box to be open.

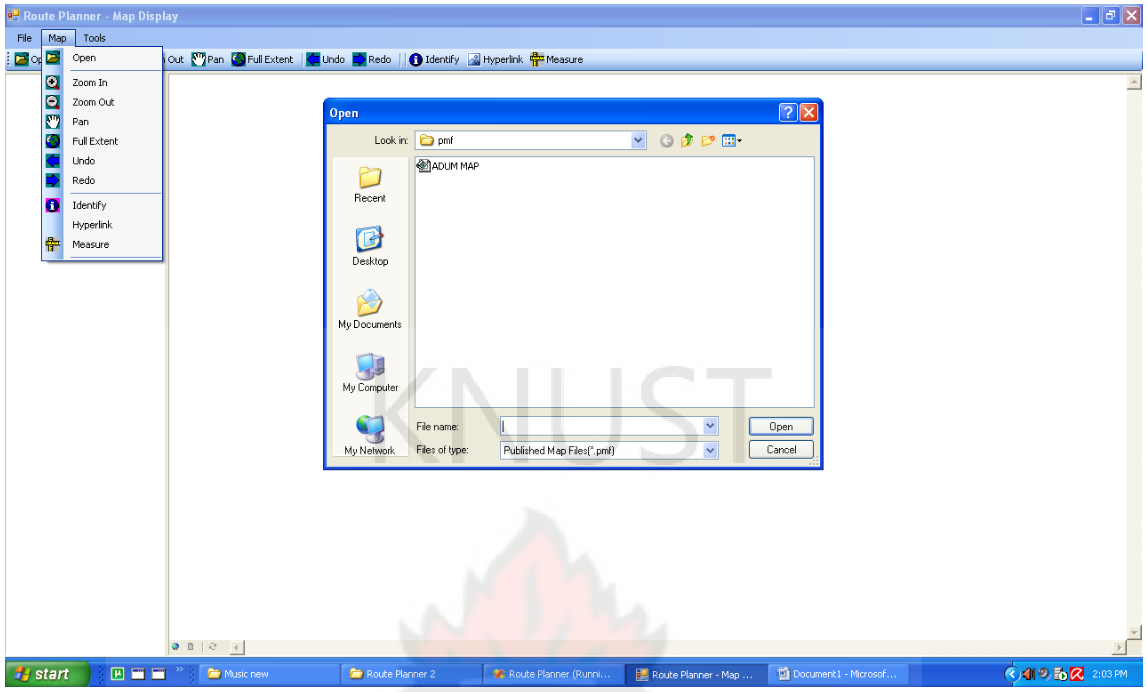


Figure 5.7: Shows the how maps are the display and selected.

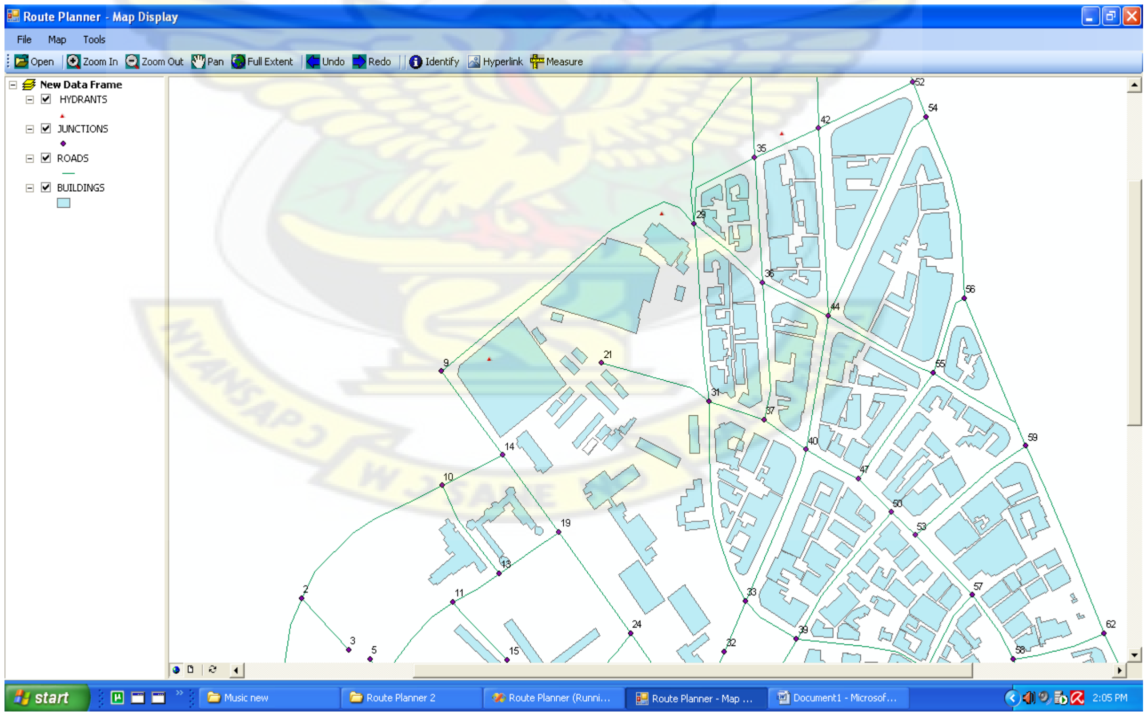


Figure 5.8: Shows the how selected map been display

Step Four: The user uses the tool menu to select the Shortest Path Navigator where the user selects the Source Street and the Destination. The flash bottom flashes the selected street and the Flicker also flicks the selected street. The Go button uses to calculate the shortest distance from the Source Street to the Destination Street on the map, and then display the distance on the blank space. The Flash Features shows the shortest path on the map.

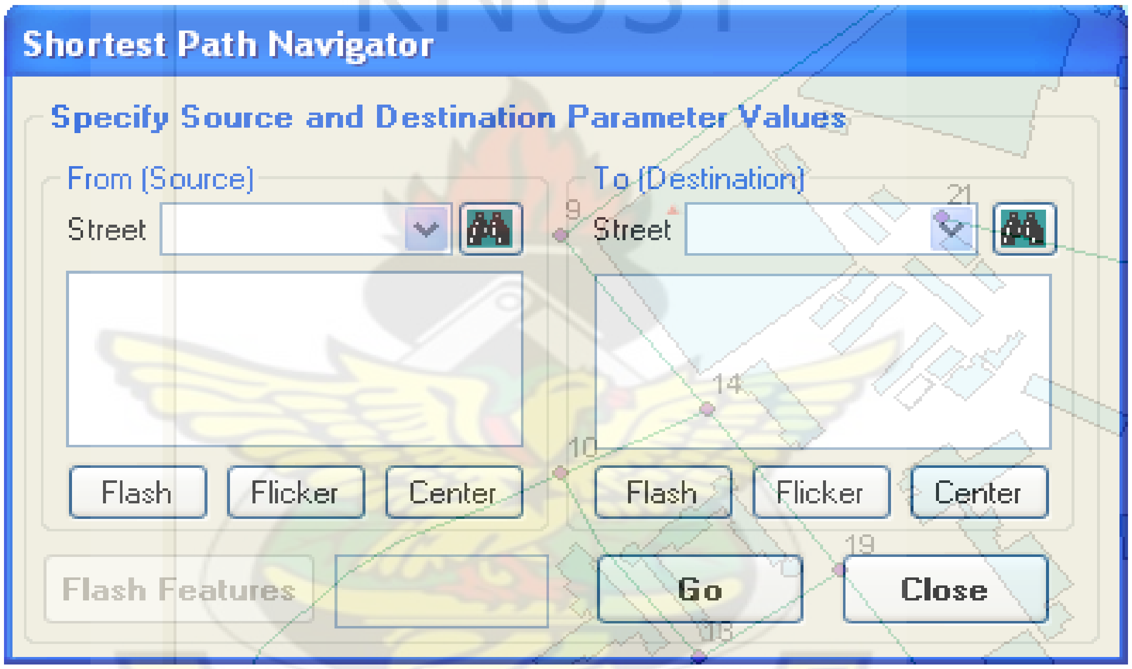


Figure 5.9: Shows the how user selects the Source Street and the Destination

Step Five : How the streets are been selected.

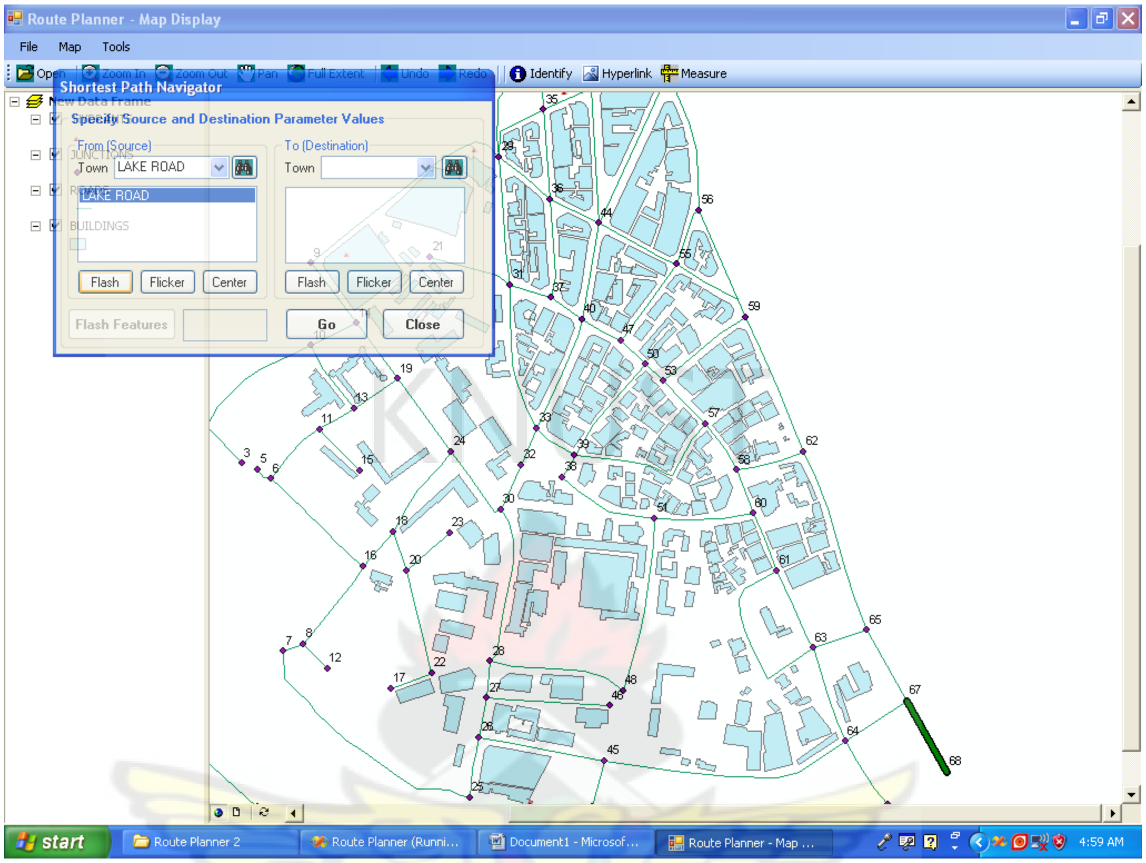


Figure 5.10: Shows the how user selects the Source Street.

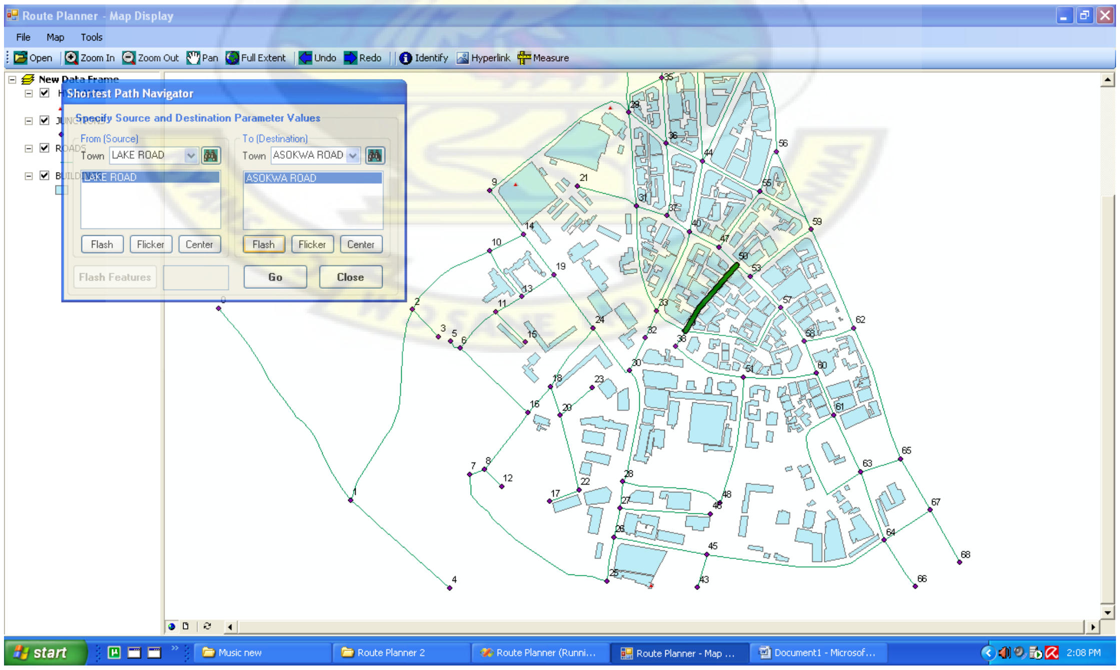


Figure 5.9: Shows the how user selects the Destination Street.

5.5 DISCUSSION AND FUTURE WORK

Computation of shortest paths is a famous area of research in Computer Science, Operations Research and GIS. There is a great number of ways to calculate shortest paths depending on the type of network and problem specification. Network Analyst is not only capable of reproducing a satisfying number of scenarios, but also it has the ability to be easily adapted to new conditions.

CODES FOR THE SHORTEST PATH USING VB.NET

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms;
using ESRI.ArcGIS.PublisherControls;
using GpsToolsNET;

namespace ShortestPathLibrary
{
    public class MapLibrary
    {
        public static int ProgressValue = 0;

        public MapLibrary()
        {
        }

        public static bool LoadMap(AxArcReaderControl arcReaderControl)
        {
            try
            {
                OpenFileDialog ofd = new OpenFileDialog();
                ofd.Filter = "Published Map Files(*.pmf)|*.pmf";
                if (ofd.ShowDialog() == DialogResult.OK)
                {
                    return LoadMap(arcReaderControl, ofd.FileName);
                }
                else
                {
                    throw new Exception("");
                }
            }
            catch (Exception ex) { return false; }
        }

        public static bool LoadMap(AxArcReaderControl arcReaderControl,
string mapPath)
        {
            try
            {
                if (arcReaderControl.CheckDocument(mapPath))
                {
                    arcReaderControl.LoadDocument(mapPath);
                    return true;
                }
                else
                {
                    throw new Exception("");
                }
            }
            catch (Exception ex) { return false; }
        }

        public static void CurrentARTool(AxArcReaderControl
arcReaderControl, MapTool mapTool)
        {
            try
            {
                if (mapTool == MapTool.FullExtent)
                {
                    if (arcReaderControl.CurrentViewType ==
esriARViewType.esriARViewTypeMap)
arcReaderControl.ARPageLayout.FocusARMap.ZoomToFullExtent();
                    else if (arcReaderControl.CurrentViewType ==
esriARViewType.esriARViewTypePageLayout)
arcReaderControl.ARPageLayout.ZoomToWholePage();
                }
                if (mapTool == MapTool.MapHyperlink)
                {
                    if (arcReaderControl.CurrentViewType ==
esriARViewType.esriARViewTypeMap) arcReaderControl.CurrentARTool =
esriARTool.esriARToolMapHyperlink;
                }
                if (mapTool == MapTool.MapIdentify)
                {
                    if (arcReaderControl.CurrentViewType ==
esriARViewType.esriARViewTypeMap) arcReaderControl.CurrentARTool =
esriARTool.esriARToolMapIdentify;
                }
                if (mapTool == MapTool.MapIdentifyUsingLayer)
                {
                    if (arcReaderControl.CurrentViewType ==
esriARViewType.esriARViewTypeMap) arcReaderControl.CurrentARTool =
esriARTool.esriARToolMapIdentifyUsingLayer;
                }
                if (mapTool == MapTool.MapMeasure)
                {
                    if (arcReaderControl.CurrentViewType ==
esriARViewType.esriARViewTypeMap) arcReaderControl.CurrentARTool =
esriARTool.esriARToolMapMeasure;
                }
                if (mapTool == MapTool.MapSwipe)
                {
                    if (arcReaderControl.CurrentViewType ==
esriARViewType.esriARViewTypeMap) arcReaderControl.CurrentARTool =
esriARTool.esriARToolMapSwipe;
```

```
                }
                if (mapTool == MapTool.MapZoomInOut)
                {
                    if (arcReaderControl.CurrentViewType ==
esriARViewType.esriARViewTypeMap) arcReaderControl.CurrentARTool =
esriARTool.esriARToolMapZoomInOut;
                }
                if (mapTool == MapTool.Pan)
                {
                    if (arcReaderControl.CurrentViewType ==
esriARViewType.esriARViewTypeMap) arcReaderControl.CurrentARTool =
esriARTool.esriARToolMapPan;
                    else if (arcReaderControl.CurrentViewType ==
esriARViewType.esriARViewTypePageLayout) arcReaderControl.CurrentARTool
= esriARTool.esriARToolLayoutPan;
                }
                if (mapTool == MapTool.RedoExtent)
                {
                    if (arcReaderControl.CurrentViewType ==
esriARViewType.esriARViewTypeMap)
arcReaderControl.ARPageLayout.FocusARMap.RedoExtent();
                    else if (arcReaderControl.CurrentViewType ==
esriARViewType.esriARViewTypePageLayout)
arcReaderControl.ARPageLayout.RedoExtent();
                }
                if (mapTool == MapTool.UndoExtent)
                {
                    if (arcReaderControl.CurrentViewType ==
esriARViewType.esriARViewTypeMap)
arcReaderControl.ARPageLayout.FocusARMap.UndoExtent();
                    else if (arcReaderControl.CurrentViewType ==
esriARViewType.esriARViewTypePageLayout)
arcReaderControl.ARPageLayout.UndoExtent();
                }
                else if (mapTool == MapTool.ZoomIn)
                {
                    if (arcReaderControl.CurrentViewType ==
esriARViewType.esriARViewTypeMap) arcReaderControl.CurrentARTool =
esriARTool.esriARToolMapZoomIn;
                    else if (arcReaderControl.CurrentViewType ==
esriARViewType.esriARViewTypePageLayout) arcReaderControl.CurrentARTool
= esriARTool.esriARToolLayoutZoomIn;
                }
                else if (mapTool == MapTool.ZoomOut)
                {
                    if (arcReaderControl.CurrentViewType ==
esriARViewType.esriARViewTypeMap) arcReaderControl.CurrentARTool =
esriARTool.esriARToolMapZoomOut;
                    else if (arcReaderControl.CurrentViewType ==
esriARViewType.esriARViewTypePageLayout) arcReaderControl.CurrentARTool
= esriARTool.esriARToolLayoutZoomOut;
                }
            }
            catch (Exception ex) { }
        }
        public static MapField GetMapField(ARLayer layer, string 
fieldName)
        {
            try
            {
                if (layer == null) throw new Exception("");
                MapField mapField = new MapField();
                ArcReaderSearchDef searchDef = new 
ArcReaderSearchDefClass();
                ARFeatureCursor featureCursor = 
layer.SearchARFeatures(searchDef);
                ARFeature feature = featureCursor.NextARFeature();
                for (int i = 0; i < feature.FieldCount; i++)
                {
                    if 
(feature.get_FieldName(i).Trim().Equals(fieldName.Trim().ToUpper(), 
StringComparison.CurrentCultureIgnoreCase))
                    {
                        mapField.FieldIndex = i;
                        mapField.FieldName = feature.get_FieldName(i);
                        mapField.FieldType = feature.get_FieldType(i);
                        break;
                    }
                }
                return mapField;
            }
            catch (Exception ex) { return null; }
        }

        public static MapField GetMapField(ARLayer layer, int 
fieldIndex)
        {
            try
            {
                if (layer == null) throw new Exception("");
                MapField mapField = new MapField();
                ArcReaderSearchDef searchDef = new 
ArcReaderSearchDefClass();
                ARFeatureCursor featureCursor = 
layer.SearchARFeatures(searchDef);
                ARFeature feature = featureCursor.NextARFeature();
                mapField.FieldIndex = fieldIndex;
                mapField.FieldName = feature.get_FieldName(fieldIndex);
                mapField.FieldType = feature.get_FieldType(fieldIndex);
                return mapField;
            }
            catch (Exception ex) { return null; }
        }

        public static bool GetShortestPath(NetworkAnalystInitialiser 
naInit)
        {
            try
            {
                if (naInit.InitialiserStatus.Status)
                {
                    ShortestPathNavigator spNavigator = new
ShortestPathNavigator(naInit);
                    if (spNavigator.InitialiserStatus.Status)
                    {
                        spNavigator.Show(naInit.OwnerForm);
                    }
                    else
                    {
                        throw new
Exception(spNavigator.InitialiserStatus.Message);
                    }
                }
                else
                {
                    throw new
Exception(naInit.InitialiserStatus.Message);
                }
                return true;
            }
            catch (Exception ex) { return false; }
        }

        public static void FlashFeatures(ARFeature[] features, int
milliSecondsTimeout)
        {
            foreach (ARFeature feature in features)
            {
                feature.Flash();
                System.Threading.Thread.Sleep(milliSecondsTimeout);
            }
        }

        public static void FlashAndHighlightFeatures(ARFeature[]
features, int milliSecondsTimeout)
        {
            foreach (ARFeature feature in features)
            {
                feature.Flash();
                feature.Highlight(true, 15000);
                System.Threading.Thread.Sleep(milliSecondsTimeout);
            }
            foreach (ARFeature feature in features)
            {
                feature.Highlight(false, 100000);
            }
        }

        public static void FlashAndSelectFeatures(AxArcReaderControl
arcReaderControl, ARFeature[] features, int milliSecondsTimeout)
        {
            ARFeatureSet featureSet;
            foreach (ARFeature feature in features)
            {
                feature.Flash();
                System.Threading.Thread.Sleep(milliSecondsTimeout);
            }
        }
        public static void FlashFeatures(ARFeatureSet features, int
milliSecondsTimeout)
        {
            for (int i = 0; i < features.ARFeatureCount; i++)
            {
                features.get_ARFeature(i).Flash();
                System.Threading.Thread.Sleep(milliSecondsTimeout);
            }
        }

        public static MapShortestPath
GetShortestPath(NetworkAnalystInitialiser naInit, string sourceNode,
string destinationNode)
        {
            try
            {
                if (!naInit.InitialiserStatus.Status)
                {
                    System.Windows.Forms.MessageBox.Show("Initialiser
status is not set");
                    return null;
                }
                MapDijkstra md = new MapDijkstra(naInit, sourceNode,
destinationNode, null, null);
                if (md.Run())
                {
                    MapShortestPath msp = new
MapShortestPath(md.ShortestPath, md.PathCost);
                    return msp;
                }
                else
                {
                    throw new Exception("There was error solving for
shortest path");
                }
            }
            catch (Exception ex) { return null; }
        }
        public static MapShortestPath
GetShortestPath(NetworkAnalystInitialiser naInit, ARFeature
sourceFeature, ARFeature destinationFeature)
        {
            try
            {
                if (!naInit.InitialiserStatus.Status)
                {
                    System.Windows.Forms.MessageBox.Show("Initialiser
status is not set");
                    return null;
                }
                MapDijkstra md = new MapDijkstra(naInit,
sourceFeature.get_ValueAsString(naInit.FromNodeIndex),
destinationFeature.get_ValueAsString(naInit.FromNodeIndex),
sourceFeature, destinationFeature);
                if (md.Run())
                {
                    MapShortestPath msp = new
MapShortestPath(md.ShortestPath, md.PathCost);
                    return msp;
                }
                else
                {
                    throw new Exception("There was error solving for
shortest path");
                }
            }
            catch (Exception ex) { return null; }
        }
        public static MapShortestPath
GetShortestPath(NetworkAnalystInitialiser naInit, ARFeature
sourceFeature, string destinationNode)
        {
            try
            {
                if (!naInit.InitialiserStatus.Status)
                {
                    System.Windows.Forms.MessageBox.Show("Initialiser
status is not set");
                    return null;
                }
                MapDijkstra md = new MapDijkstra(naInit,
sourceFeature.get_ValueAsString(naInit.FromNodeIndex), destinationNode,
sourceFeature, null);
                if (md.Run())
                {
                    MapShortestPath msp = new
MapShortestPath(md.ShortestPath, md.PathCost);
                    return msp;
                }
                else
                {
                    throw new Exception("There was error solving for
shortest path");
                }
            }
            catch (Exception ex) { return null; }
        }
        public static MapShortestPath
GetShortestPath(NetworkAnalystInitialiser naInit, string sourceNode,
ARFeature destinationFeature)
        {
            try
            {
                if (!naInit.InitialiserStatus.Status)
                {
                    System.Windows.Forms.MessageBox.Show("Initialiser
status is not set");
                    return null;
                }
                MapDijkstra md = new MapDijkstra(naInit, sourceNode,
destinationFeature.get_ValueAsString(naInit.FromNodeIndex), null,
destinationFeature);
                if (md.Run())
                {
                    MapShortestPath msp = new
MapShortestPath(md.ShortestPath, md.PathCost);
                    return msp;
                }
                else
                {
                    throw new Exception("There was error solving for
shortest path");
                }
            }
            catch (Exception ex) { return null; }
        }
    }

    public class MapDijkstra
    {
        private struct Node
        {
            public string Label;
            public double Distance;
            public bool Visited;
            public Node(string l, double d, bool v)
            {
                Label = l;
                Distance = d;
                Visited = v;
            }
        }
        public ARFeatureSet Edges;
        private NetworkAnalystInitialiser naInit;
        public ArrayList Nodes;
        private ArrayList[] pathNodes;
        string sourceNodeLabel, destinationNodeLabel;
        ARFeature sourceFeature = null, destinationFeature = null;
        public System.Collections.ArrayList PathNodes = new
System.Collections.ArrayList();
        public ARFeature[] ShortestPath;
        public double PathCost = 0;
        public MapDijkstra(NetworkAnalystInitialiser naInit, string
sourceNodeLabel, string destinationNodeLabel, ARFeature sourceFeature,
ARFeature destinationFeature)
        {
            this.naInit = naInit;
            this.sourceNodeLabel = sourceNodeLabel;
            this.destinationNodeLabel = destinationNodeLabel;
            this.sourceFeature = sourceFeature;
            this.destinationFeature = destinationFeature;
            ArcReaderSearchDef searchDef = new
ArcReaderSearchDefClass();
            Edges = naInit.ARRouteLayer.QueryARFeatures(searchDef);
        }
        private bool UnvisitedNodeExists()
        {
            try
            {
                foreach (Node node in Nodes) if (!node.Visited) return
true;
                return false;
            }
            catch (Exception ex) { return false; }
        }
        private Node GetMinimumNode(Node node, int currentNodeIndex)
        {
            Node minNode = new Node(); minNode.Distance = -1;
            double length;
            Node tNode;

            for (int nodeIndex = 0; nodeIndex < Nodes.Count;
nodeIndex++)
            {
                tNode = (Node)Nodes[nodeIndex];
                if (!tNode.Label.Equals(node.Label) && !tNode.Visited)
                {
                    length = GetEdgeLength(node.Label, tNode.Label);
                    if (length > 0)
                    {
                        if (tNode.Distance < 0)
                        {
                            tNode.Distance = node.Distance + length;
                            pathNodes[currentNodeIndex].TrimToSize();
                            pathNodes[nodeIndex] = new
System.Collections.ArrayList();
                            foreach (int ni in
pathNodes[currentNodeIndex]) pathNodes[nodeIndex].Add(ni);
pathNodes[nodeIndex].TrimToSize();
                        }
                        else if ((node.Distance + length) <
tNode.Distance)
                        {
                            tNode.Distance = node.Distance + length;
                            pathNodes[currentNodeIndex].TrimToSize();
                            pathNodes[nodeIndex] = new
System.Collections.ArrayList();
                            foreach (int ni in
pathNodes[currentNodeIndex]) pathNodes[nodeIndex].Add(ni);
pathNodes[nodeIndex].TrimToSize();
                        }
                    }
                    Nodes[nodeIndex] = tNode;
                }
            }
            foreach (Node nd in Nodes)
            {
                if (!nd.Visited && nd.Distance > 0)
                {
                    if (minNode.Distance < 0)
                    {
                        minNode = nd;
                    }
                    else if (minNode.Distance > nd.Distance)
                    {
                        minNode = nd;
                    }
                }
            }
            return minNode;
        }
        private double GetEdgeLength(string start, string stop)
        {
            double length = 0;
            ARFeature tf;
            for (int i = 0; i < Edges.ARFeatureCount; i++)
            {
                tf = Edges.get_ARFeature(i);
                if
((tf.get_ValueAsString(naInit.FromNodeIndex).Equals(start) &&
tf.get_ValueAsString(naInit.ToNodeIndex).Equals(stop)) ||
(tf.get_ValueAsString(naInit.FromNodeIndex).Equals(stop) &&
tf.get_ValueAsString(naInit.ToNodeIndex).Equals(start)))
                {
                    length =
Convert.ToDouble(tf.get_Value(naInit.ShapeLengthIndex));
                    break;
                }
            }
            return length;
        }
        private ARFeature GetEdge(string start, string stop)
        {
            ARFeature tempFeature = null;
            for (int i = 0; i < Edges.ARFeatureCount; i++)
            {
                tempFeature = Edges.get_ARFeature(i);
                if
((tempFeature.get_ValueAsString(naInit.FromNodeIndex).Equals(start) &&
tempFeature.get_ValueAsString(naInit.ToNodeIndex).Equals(stop)) ||
(tempFeature.get_ValueAsString(naInit.FromNodeIndex).Equals(stop) &&
tempFeature.get_ValueAsString(naInit.ToNodeIndex).Equals(start)))
                {
                    break;
                }
            }
            return tempFeature;
        }
        private int SetNodes()
        {
            int sourceNodeIndex = -1;
            ARFeature tempFeature;
            Nodes = new ArrayList();
            for (int i = 0; i < Edges.ARFeatureCount; i++)
            {
                tempFeature = Edges.get_ARFeature(i);
                Node startNode = new
Node(tempFeature.get_ValueAsString(naInit.FromNodeIndex).Trim(), -1,
false);
                Node endNode = new
Node(tempFeature.get_ValueAsString(naInit.ToNodeIndex).Trim(), -1,
false);
                if (Nodes.IndexOf(startNode) < 0)
                {
                    if (startNode.Label.Equals(sourceNodeLabel) &&
sourceNodeIndex < 0)
                    {
                        startNode.Distance = 0;
                        sourceNodeIndex = Nodes.Count;
                        Nodes.Add(startNode);
                    }
                    else if (!endNode.Label.Equals(sourceNodeLabel))
                    {
                        Nodes.Add(startNode);
                    }
                }
                if (Nodes.IndexOf(endNode) < 0)
                {
                    if (endNode.Label.Equals(sourceNodeLabel) &&
sourceNodeIndex < 0)
                    {
                        endNode.Distance = 0;
                        sourceNodeIndex = Nodes.Count;
                        Nodes.Add(endNode);
                    }
                    else if (!endNode.Label.Equals(sourceNodeLabel))
                    {
                        Nodes.Add(endNode);
                    }
                }
            }
            Nodes.TrimToSize();
            pathNodes = new System.Collections.ArrayList[Nodes.Count];
            return sourceNodeIndex;
        }
        public bool Run()
        {
            try
            {
                if (sourceNodeLabel.Equals(destinationNodeLabel)) throw
new Exception("Source and destination are similar");
                int nodeIndex = SetNodes();
                if (nodeIndex < 0) throw new Exception("Nodes could not
be set");

                Node currentNode = (Node)Nodes[nodeIndex];
                pathNodes[nodeIndex] = new
System.Collections.ArrayList();
                while (UnvisitedNodeExists())
                {
                    nodeIndex = Nodes.IndexOf(currentNode);
                    currentNode.Visited = true;
                    pathNodes[nodeIndex].Add(nodeIndex);
                    Nodes[nodeIndex] = currentNode;
                    if (currentNode.Label.Equals(destinationNodeLabel))
break;
                    currentNode = GetMinimumNode(currentNode,
nodeIndex);
                }
                //Make sure source and destination features are part of
the collection
                ARFeature tempFeature;
                if (destinationFeature != null)
                {
                    tempFeature =
GetEdge(((Node)Nodes[(int)pathNodes[nodeIndex][pathNodes[nodeIndex].Cou
nt - 2]]).Label,
((Node)Nodes[(int)pathNodes[nodeIndex][pathNodes[nodeIndex].Count -
1]]).Label);
                    if
(destinationFeature.get_ValueAsString(naInit.ObjectIDIndex).Equals(temp
Feature.get_ValueAsString(naInit.ObjectIDIndex)))
                    {
                        destinationFeature = null;
                    }
                }
                if (sourceFeature != null)
                {
                    tempFeature =
GetEdge(((Node)Nodes[(int)pathNodes[nodeIndex][0]]).Label,
((Node)Nodes[(int)pathNodes[nodeIndex][1]]).Label);
                    if
(sourceFeature.get_ValueAsString(naInit.ObjectIDIndex).Equals(tempFeatu
re.get_ValueAsString(naInit.ObjectIDIndex)))
                    {
                        sourceFeature = null;
                    }
                }
                //Fill Path Nodes arraylist
                if (sourceFeature != null && destinationFeature !=
null)
                {
                    ShortestPath = new
ARFeature[pathNodes[nodeIndex].Count - 1 + 2];
                    ShortestPath[0] = sourceFeature;
                    ShortestPath[ShortestPath.Length - 1] =
destinationFeature;
                    for (int i = 0; i < pathNodes[nodeIndex].Count - 1;
i++)
                    {
                        ShortestPath[i + 1] =
GetEdge(((Node)Nodes[(int)pathNodes[nodeIndex][i]]).Label,
((Node)Nodes[(int)pathNodes[nodeIndex][i + 1]]).Label);
                    }
                }
                else if (sourceFeature != null)
                {
                    ShortestPath = new
ARFeature[pathNodes[nodeIndex].Count - 1 + 1];
                    ShortestPath[0] = sourceFeature;
                    for (int i = 0; i < pathNodes[nodeIndex].Count - 1;
i++)
                    {
                        ShortestPath[i + 1] =
GetEdge(((Node)Nodes[(int)pathNodes[nodeIndex][i]]).Label,
((Node)Nodes[(int)pathNodes[nodeIndex][i + 1]]).Label);
```

```
                    }
                }
                else if (destinationFeature != null)
                {
                    ShortestPath = new
ARFeature[pathNodes[nodeIndex].Count - 1 + 1];
                    ShortestPath[ShortestPath.Length - 1] =
destinationFeature;
                    for (int i = 0; i < pathNodes[nodeIndex].Count - 1;
i++)
                    {
                        ShortestPath[i] =
GetEdge(((Node)Nodes[(int)pathNodes[nodeIndex][i]]).Label,
((Node)Nodes[(int)pathNodes[nodeIndex][i + 1]]).Label);
                    }
                }
                else
                {
                    ShortestPath = new
ARFeature[pathNodes[nodeIndex].Count - 1];
                    for (int i = 0; i < pathNodes[nodeIndex].Count - 1;
i++)
                    {
                        ShortestPath[i] =
GetEdge(((Node)Nodes[(int)pathNodes[nodeIndex][i]]).Label,
((Node)Nodes[(int)pathNodes[nodeIndex][i + 1]]).Label);
                    }
                }
                //Calculate Path Cost
                PathCost = 0;
                foreach (ARFeature feature in ShortestPath) PathCost +=
Convert.ToDouble(feature.get_Value(naInit.ShapeLengthIndex));
                return true;
            }
            catch (Exception ex) { return false; }
        }
    }

    public class NetworkAnalystInitialiser
    {
        private Form ownerForm;
        private AxArcReaderControl arControl;
        private string routeLayerName;
        private ArrayList mapLayers;
        private ARLayer arRouteLayer;

        private ArrayList arLayers;
        private ArrayList routeLinkers = null;
        private string objectIDField, fromNodeField, toNodeField,
shapeLengthField, linkFieldName;
        private int objectIDIndex, fromNodeIndex, toNodeIndex,
shapeLengthIndex, linkFieldIndex;
        private bool linkFieldNumeric;
        private ResultObject initStatus = new ResultObject();

        public NetworkAnalystInitialiser(Form ownerForm,
AxArcReaderControl arControl, string routeLayerName)
```

```
        {
            this.ownerForm = ownerForm;
            this.arControl = arControl;
            this.routeLayerName = routeLayerName;
            mapLayers = new ArrayList();
        }
        public NetworkAnalystInitialiser(Form ownerForm,
AxArcReaderControl arControl, string routeLayerName, string
nodeLayerName, string objectIDField, string fromNodeField, string
toNodeField, string shapeLengthField)
        {
            this.ownerForm = ownerForm;
            this.arControl = arControl;
            this.routeLayerName = routeLayerName;
            this.objectIDField = objectIDField;
            this.fromNodeField = fromNodeField;
            this.toNodeField = toNodeField;
            this.shapeLengthField = shapeLengthField;
            mapLayers = new ArrayList();
            Initialise();
        }

        public Form OwnerForm
        {
            get { return ownerForm; }
            set { ownerForm = value; }
        }
        public AxArcReaderControl ARControl
        {
            get { return arControl; }
            set { arControl = value; }
        }
        public string RouteLayer { get { return routeLayerName; } }

        public ARLayer ARRouteLayer { get { return arRouteLayer; } }
        public ArrayList MapLayers { get { return mapLayers; } }

        public ArrayList ARLayers
        {
            get { return arLayers; }
        }
        public ArrayList RouteLinkers
        {
            get { return routeLinkers; }
        }
        public string ObjectIDField
        {
            get { return objectIDField; }
            set { objectIDField = value; }
        }
        public string FromNodeField
        {
            get { return fromNodeField; }
            set { fromNodeField = value; }
        }
        public string ToNodeField
        {
```

```
    get { return toNodeField; }
    set { toNodeField = value; }
}
public string ShapeLengthField
{
    get { return shapeLengthField; }
    set { shapeLengthField = value; }
}
public string LinkFieldName
{
    get { return linkFieldName; }
    set { linkFieldName = value; }
}
public int ObjectIDIndex
{
    get
    {
        if (initStatus.Status) return objectIDIndex;
        else throw new Exception("Object ID Index Not Set");
    }
}
public int FromNodeIndex
{
    get
    {
        if (initStatus.Status) return fromNodeIndex;
        else throw new Exception("From Node Index Not Set");
    }
}
public int ToNodeIndex
{
    get
    {
        if (initStatus.Status) return toNodeIndex;
        else throw new Exception("To Node Index Not Set");
    }
}
public int ShapeLengthIndex
{
    get
    {
        if (initStatus.Status) return shapeLengthIndex;
        else throw new Exception("Shape Length Index Not Set");
    }
}
public int LinkFieldIndex
{
    get
    {
        if (initStatus.Status) return linkFieldIndex;
        else throw new Exception("Link Field Index Not Set");
    }
}
public bool LinkFieldNumeric
{
    get { return linkFieldNumeric; }
}
```

```
        public ResultObject InitialiserStatus
        {
            get { return initStatus; }
        }
        public MapLayer GetMapLayer(string layerName)
        {
            if (initStatus.Status)
            {
                foreach (MapLayer mapLayer in mapLayers)
                {
                    if
(mapLayer.LayerName.Trim().Equals(layerName.Trim(),
StringComparison.CurrentCultureIgnoreCase))
                    {
                        return mapLayer;
                    }
                }
            }
            return null;
        }
        public ARLayer GetARLayer(string layerName)
        {
            if (initStatus.Status)
            {
                foreach (ARLayer arLayer in arLayers)
                {
                    if (arLayer.Name.Trim().Equals(layerName.Trim(),
StringComparison.CurrentCultureIgnoreCase))
                    {
                        return arLayer;
                    }
                }
            }
            return null;
        }
        public bool AddRouteLinker(RouteLinker routeLinker)
        {
            try
            {
                if (routeLinkers == null) routeLinkers = new
ArrayList();
                routeLinkers.Add(routeLinker);
                routeLinkers.TrimToSize();
                return true;
            }
            catch (Exception ex) { return false; }
        }
        public bool AddMapLayer(MapLayer mapLayer)
        {
            try
            {
                mapLayers.Add(mapLayer);
                return true;
            }
            catch (Exception ex) { return false; }
        }
        private bool AddARLayer(ARLayer arLayer)
        {
            try
            {
                if (arLayer.Searchable)
                {
                    if (arLayer.IsGroupLayer)
                    {
                        for (int i = 0; i < arLayer.ARLayerCount; i++)
                        {
                            AddARLayer(arLayer.get_ChildARLayer(i));
                        }
                    }
                    else
                    {
                        if (GetMapLayer(arLayer.Name.Trim()) != null)
arLayers.Add(arLayer);
                        if
(arLayer.Name.Trim().Equals(routeLayerName.Trim(),
StringComparison.CurrentCultureIgnoreCase)) { arRouteLayer = arLayer; }
                    }
                }
                return true;
            }
            catch (Exception ex) { return false; }
        }
        public bool Initialise()
        {
            initStatus.Status = true;
            initStatus.Message = "Initialiser correctly set";
            try
            {
                if (this.arControl == null)
                {
                    initStatus.Status = false;
                    initStatus.Message = "ArcReaderControl not set";
                }
                if (this.routeLayerName.Trim().Length == 0)
                {
                    initStatus.Status = false;
                    initStatus.Message = "Route layer not set";
                }
                arLayers = new ArrayList();
                arRouteLayer = null;
                for (int i = 0; i <
arControl.ARPageLayout.FocusARMap.ARLayerCount; i++)
                {
                    if
(!AddARLayer(arControl.ARPageLayout.FocusARMap.get_ARLayer(i)))
                    {
                        initStatus.Status = false;
                        initStatus.Message = "Layer could not be
added";
                    }
                }
                arLayers.TrimToSize();
                objectIDIndex = fromNodeIndex = toNodeIndex =
shapeLengthIndex = linkFieldIndex = -1;
```

```
                ArcReaderSearchDef searchDef = new
ArcReaderSearchDefClass();
                ARFeatureCursor featureCursor =
arRouteLayer.SearchARFeatures(searchDef);
                ARFeature feature = featureCursor.NextARFeature();
                for (int i = 0; i < feature.FieldCount; i++)
                {
                    if
(feature.get_FieldName(i).Trim().Equals(objectIDField.Trim(),
StringComparison.CurrentCultureIgnoreCase)) objectIDIndex = i;
                    if
(feature.get_FieldName(i).Trim().Equals(fromNodeField.Trim(),
StringComparison.CurrentCultureIgnoreCase)) fromNodeIndex = i;
                    if
(feature.get_FieldName(i).Trim().Equals(toNodeField.Trim(),
StringComparison.CurrentCultureIgnoreCase)) toNodeIndex = i;
                    if
(feature.get_FieldName(i).Trim().Equals(shapeLengthField.Trim(),
StringComparison.CurrentCultureIgnoreCase)) shapeLengthIndex = i;
                    if
(feature.get_FieldName(i).Trim().Equals(linkFieldName.Trim(),
StringComparison.CurrentCultureIgnoreCase)) { linkFieldIndex = i;
linkFieldNumeric = (feature.get_FieldType(i) ==
esriARFieldType.esriARFieldTypeInteger || feature.get_FieldType(i) ==
esriARFieldType.esriARFieldTypeSmallInteger) ? true : false; }
                }
                if (objectIDIndex == -1 || fromNodeIndex == -1 ||
toNodeIndex == -1 || shapeLengthIndex == -1 || linkFieldIndex == -1 )
initStatus.Status = false;
            }
            catch (Exception ex)
            {
                initStatus.Status = false;
                initStatus.Message = ex.Message;
            }
            return initStatus.Status;
        }
    }

    public class MapShortestPath
    {
        private ARFeature[] features;
        private double pathCost;
        public MapShortestPath(ARFeature[] features, double pathCost)
        {
            this.pathCost = pathCost;
            this.features = features;
        }
        public ARFeature[] Features
        {
            get { return features; }
        }
        public double PathCost
        {
            get { return pathCost; }
        }
    }
    public class ResultObject
    {
        private bool status = false;
        private string message = "";
        public ResultObject()
        {
        }
        public bool Status { get { return status; } set { status =
value; } }
        public string Message { get { return message; } set { message =
value; } }
    }

    public class MapLayer
    {
        private string layerName;
        private ArrayList searchableFields;
        public MapLayer(string layerName)
        {
            this.layerName = layerName;
            searchableFields = new ArrayList();
        }
        public string LayerName
        {
            get { return layerName; }
            set { layerName = value; }
        }
        public ArrayList SearchableFields
        {
            get { return searchableFields; }
        }
        public bool AddSearchableField(string fieldName)
        {
            try
            {
                searchableFields.Add(fieldName);
                searchableFields.TrimToSize();
                return true;
            }
            catch (Exception ex) { return false; }
        }
    }

    public class MapField
    {
        private int fieldIndex;
        private string fieldName;
        private bool isNumeric;
        private esriARFieldType fieldType;

        public MapField()
        {
        }

        public int FieldIndex
        {
            get { return fieldIndex; }
            set { fieldIndex = value; }
        }
        public string FieldName
        {
            get { return fieldName; }
            set { fieldName = value; }
        }
        public bool IsNumeric
        {
            get { return isNumeric; }
        }
        public esriARFieldType FieldType
        {
            get { return fieldType; }
            set
            {
                fieldType = value;
                if (fieldType == esriARFieldType.esriARFieldTypeDouble
|| fieldType == esriARFieldType.esriARFieldTypeInteger || fieldType ==
esriARFieldType.esriARFieldTypeOID || fieldType ==
esriARFieldType.esriARFieldTypeSingle || fieldType ==
esriARFieldType.esriARFieldTypeSmallInteger)
                {
                    isNumeric = true;
                }
                else
                {
                    isNumeric = false;
                }
            }
        }
    }

    public class RouteLinker
    {
        private string layerName;
        private string linkField;
        public RouteLinker()
        {
        }
        public RouteLinker(string layerName, string linkField)
        {
            this.layerName = layerName;
            this.linkField = linkField;
        }
        public string LayerName
        {
            get { return layerName; }
            set { layerName = value; }
        }
        public string LinkField
        {
            get { return linkField; }
            set { linkField = value; }
        }
    }
    public enum MapTool
    {
        Pan,
        ZoomIn,
        ZoomOut,
        MapHyperlink,
        MapIdentify,
        MapIdentifyUsingLayer,
        MapMeasure,
        MapSwipe,
        MapZoomInOut,
        FullExtent,
        UndoExtent,
        RedoExtent
    }

    public enum ShortestPathParameterType
    {
        FeatureToFeature, None
    }
}
```

CHAPTER 6

CONCLUSIONS AND RECOMMENDATIONS

This study addresses the problem of determining dynamic shortest path in traffic networks, where arc travel times vary over time. This study proposes a dynamic routing system which is based on the integration of GIS and real-time traffic conditions. It uses GIS for improving the visualization of the urban network map and analysis of car routing. GIS is used as a powerful functionality for planning optimal routes based on particular map travel time information.

The results of this study illustrate that dynamic routing of emergency vehicle compared with static solution is much more efficient. This efficiency will be most important when unwanted incidents take place in roads and serious traffic congestion occur. In this study, the initial planned route is saved since when exist at any distance. The routing system analysis real distance data receives only portion of the planned path traffic data and vehicle location to determine if the direction may be a changed. This improves the computational planned route need to be modified.

- This study addresses the problem of determining shortest path in traffic networks, in Kumasi Metropolis
- The study proposes a routing system which is based on the integration of ArcGIS and road distance.

- It uses ArcGIS and VB.NET coding to obtain user friendly interface which allows the visualization of the Adum road map and traversal of shortest route between two selected junctions.

The updated route is send via a dynamic routing system for all vehicles in urban (Adum) road communication system to vehicle driver to change his network has some special considerations which are the route. This process continues until the mission of subject of our future work.

6.1 Future work

Sometimes the given algorithms may produce output that is of no use even though it has been correctly generated. For example, there can be a path that will require a car and one bus only to reach the destination after 30 minutes. However, the algorithm may advise you to take a car and three times to take a bus which will take 25 minutes, 5 minutes less than the previous path. From the point of view of the defined conditions the second path is better, but a more reasonable path is the first one, though 5 minutes shorter. The first path is actually better because it is less cumbersome (it is easier to take one bus instead of three), more reliable (three buses cause more risk than only one since each bus can break down, changing buses is risky as opposed to sitting in the bus) and is cheaper. This example proves the need to introduce different conditions for solving the shortest route problem.

The future research can go into two directions. First, well known algorithms can be adapted into the public transport needs. For example, the algorithm for finding second shortest path, third etc, paths for buses can be developed. More can be proposed: finding the shortest path going through specific nodes, through specific number of nodes or by the most reliable path.

The other direction is more interesting: development of new algorithms for traffic issues and not just adaptation of existing algorithms. So far there has not been devised (as far as the author of this report knows) an algorithm for many public transport means: a train, an underground, buses and a car. There would not be anything interesting in this except that the buses and metro would be considered in parallel. A user could point out that the path should be build up in accordance with the following criteria:

- The allowed types of changes (for example to change a bus to a train may be disallowed).
- Transportation, calculating accessibility for mass transit systems by using a complex network map.
- Public works, determining the optimal route for point- to – point pickups of trash items.
- Public Safety, routing emergency response crew to incidents, or calculating drive distance for first responder planning.

More than that, user can specify exactly how many changes he wants between different types of transportation. For example the user can say that only one change between car and bus is allowed but that changing between buses and an underground vehicle can be done as many times as necessary. Also, the number of changes can be named as *at* most or exactly.

Therefore saying at most 3 changes of vehicle can ban choosing the best route with only one change. But still, this is should also be possible to find the path with exact number of changes. The flexibility of conditions seems to be very big.

REFERENCES

Ahuja, R. K., Magnanti, T. L., Orlin, J. B., (1993).
Network Flows: Theory, Algorithms and Applications, Prentice Hall, Englewood Cliffs, NJ

Bellman, R., (1958) On a Routing Problem, Quart. Appl. Math. 16, 87-90

Cherkassky, B. V., Goldberg, A. V., Radzik, T., 1996, Shortest path algorithms: Theory and experimental evaluation, Mathematical Programming 73, 129-174

Cai, X., Klocks, T., Wong, C.K., (1997) Time-Varying Shortest Path Problems with Constraints, Networks 29, 1997, 141-149

Husdal, J. (2000). Fastest Path Problems in Dynamic Transportation Networks, http://www.husdal.com/mscgis/research.htm, last accessed November 22, 2005.

Vonderohe, A. P., Travis, L., Smith, R. L. and Tasai, V. (1993). NCHRP Report 359, Adoption of Geographic Information System for Transportation, Transport Research Board, National Research Council, Washington, DC.

OpenGIS - A Request for Technology - In Support of an Open Location Services (OpenLSTM) Testbed, 2000.

Skiena, S. (1990). Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica. Reading, MA: Addison-Wesley, pages. 135-136.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs Numerische Mathematik 1, pages. 269-271.

Hart, P. E., Nilsson, N. J., Raphael, B. (1972). Correction to "A Formal Basis for the Heuristic Determination of Minimum Cost Paths", SIGART Newsletter, 37, pages. 28-29.

Arrival Time Dependent Shortest Path by On – Road Routing in Mobile Ad – Hoc Network (2005)

Optimisation of an Existing Forest Road Network Using Network 2000

(Akio KONDO, Yoshitaka AOYAMA & Yoshinobu HIROSE (1994)) The Impact of the
Development of High Mobility Transportation Networks on Rural Cities, Related Problems and
Countermeasures.

Tadaomi SETOGUCHI (1994), Observation on the Characteristics of Converted Traffic, Viewed from the Available Forms of Urban Express ways.

Hiroshi TANOUE, Takashi CHISHAKI, Masaru KIYOTA & Chikashi DEGUCHI (1994)
A Method for Determining the Groups of Road Sections to Be Simultaneously Constructed and
the Priority between Them by Considering the Priority and Simultaneity of Road Construction

CHISHAKI, Guoquan Li & Wen-Chih Huang (1994) The Developing Trend of Taxi Traffic in Beijing Metropolitan Region Zhongying Dong, Takeshi

Young-Hwan Lee (1994) A Study on the Development Plan of the New Yong-Jong Island International Airport

Chikashi DEGUCHI, Hiroshi YOKOTA, Masaru KIYOTA & Tetsunobu Yoshitake (1994)
Survey and Analysis of the Characteristics of Traffic Access and Parking at the Central Area of a Local City

Aminu A Ibrahim (2007) Graph Algorithms and shortest path problem: A case of Dijkstra's
Algorithm and the Dual Carriage Ways in Sokoto Metropolis

Stefan Irnich (2002) The Shortest Path Problem with k-Cycle Elimination (k _ 3): Improving
Branch and Price Algorithms for Vehicle Routing and Scheduling

Peter W. Eklund, Steve Kirkby1, Simon Pollitt (2001) A Dynamic Multi-source Dijkstra's Algorithm for Vehicle Routing

Liang Dai (2000) Fast Shortest Path Algorithm for Road Network and Implementation

Application of multi – objective shortest path and Allocation Analysis of Flood Prevention.

Pierre A. Humblet (1988), An Adaptive Distributed Dijkstra Shortest Path Algorithm

Shane Saunders and Tadao Takaoka (2006) Efficient Algorithms for Solving Shortest Paths on Nearly Acyclic Directed Graphs

Networks Holger Bast, Stefan Funke, Domagoj Matijevic, Peter Sanders and Dominik Schultes
(2007). In Transit to Constant Time Shortest-Path Queries in Road

QU Rong , WENG Min , DU QingYun ,and CAI ZhongLiang (2005) Combining Algorithm with Knowledge For Way-Finding

Approximating Shortest Paths in Larger – scale Networks with an Application to Intelligent Transportation Systems Yu – Li Chot, H. Edwin Romeijn and Robert L. Smith

Path Planning Under Uncertainty: Complexity and Algorithms Evdokia Nikolova

A GIS-based Dynamic Shortest Path Determination in Emergency Vehicles

Fast shortest path computation in time-dependent traffic networks

Cooke, K. L. and Halsey, E. (1966). "The Shortest Route Through a Network with Time-Dependent Inter-nodal Transit Times", *Journal of Mathematical Analysis and Applications* 14, pages. 493-498.

Bellman, R. (1958). "On a routing problem", Quarterly of Applied Mathematics 16, page. 87-90.

Hall, R. W. (1986). "The Fastest Path through a Network with Random Time-Dependent Travel Times", Transportation Science 20, pages. 182-188.

Frank, H. (1969). “Shortest Paths in Probabilistic Graph”, Operations Research, 17(4), pages. 583-599.

Ramalingam, G. and Reps, T. (1996), "On the computational complexity of dynamic graph problems", Theoretical Computer Science 158 (1–2), pages. 233–277.

Frigioni, D., Marchetti-Spaccamela, A., Nanni, U. (1998). "Semidynamic algorithms for maintaining single source shortest path trees", Algorithmica 22 (3), pages. 250–274.

Ramalingam, G. and Reps, T. (1996). "An incremental algorithm for a generalization of the shortest-path problem", Journal of Algorithms 21, pages. 267–305.

Buriol, L.S., Resende, M.G.C., Ribeiro, C.C., Thorup, M. (2003). A hybrid genetic algorithm for the weight setting problem in ospf/is-is routing. Networks, under review.

Fortz, B. and Thorup, M. (2000). Increasing internet capacity using local search,Technical report, AT&T Labs, Research, 180 Park Avenue, Florham Park, NJ 07932 USA.

Frigioni, D., Marchetti-Spaccamela, A., Nanni, U. (1998). Semi-dynamic algorithms for maintaining singlesource shortest path trees. *Algorithmica*, 22(3), pages. 250–274.

Terrovitis, M., Bakiras, S., Papadias, D., Mouratidis, K. (2005). "Constrained Shortest Path Computation", SSTD 2005, LNCS 3633, pages. 181–199.

Ausiello, G., Italiano, G., Marchetti-Spaccamela, A., Nanni, U. (1991). "Incremental algorithms for minimal length paths", Journal of Algorithms 12 (4), pages. 615–638.
Even, S. and Shiloach, Y. (1981). "An on-line edge deletion problem", Journal of the ACM 28 (1).

Feuerstein, E. and Marchetti-Spaccamela, A. (1993). "Dynamic algorithms for shortest paths in planar graphs", Theoretical Computer Science 116 (2), pages. 359–371.

Franciosa, P, Frigioni, D., Giaccio, R. (2001). "Semi-dynamic breadth-first search in digraphs", Theoretical Computer Science 250 (1–2), pages. 201–217.

Frigioni, D., Marchetti-Spaccamela, A., Nanni, U. (1996). "Fully dynamic output bounded single source shortest path problem", in: Proceedings of the Symposium on Discrete Algorithms, pages. 212–221.

Cooke, K. L., Halsey, E., The shortest route through a network with time-dependent intermodal transit times, Journal of Mathematical Analysis and Applications 14, 1966, 493-498

Dijkstra, E. W., A Note on Two Problems in Connexion with Graphs, Numerishe Mathematic 1, 1959, 269-271

Dreyfus, S. E., An Appraisal of Some Shortest-Path Algorithms, Operations Research 17, 1969, 395-412

Klein, P.N., Subramanian, S., A Randomized Parallel Algorithm for Single-Source Shortest Paths, Journal of Algorithms, Vol. 25, No. 2, Nov 1997, pp. 205-220

Henzinger, M. R., P. Klein, P., Rao, S., Sairam Subramanian, Faster Shortest-Path Algorithms for Planar Graphs, Journal of Computer and System Science 55, 1997,3-23

Minty, G., A comment on the shortest route problem. Operations Research. 5, 724, 1957

Moore, E.F., The shortest path through a maze. Proceeding of an International Symposium on the theory of Switching, Part II, April 2-5, 1957, The Annals of the Computation Laboratory of

Harvard University 30, Harvard University Press, and Cambridge Mass.
Steenbrink, P. A., Optimisation of transport networks, John Wiley & Sons Ltd, 1974

Dijkstra, E.W., 1959. “A note on two problems in connexion with graphs”. Numerische Mathematik, 1, 269-271.

Dorigo, M., Di Caro, G., Gambardella, L.M., 1999. “Ant Algorithms for Discrete Optimization”. Artificial Life, 5(2), 137–172.

ESRI, GIS and Mapping Software Support Group, 2006. “ArcGIS Network Analyst: Routing, Closest Facility, and Service Area Analysis”. http://www.esri.com/networkanalyst (Accessed on

February 10, 2007).

GEOLORE (2003): Development of a Geographical Information System for the organization and
implementation of an integrated waste management at a local and/or regional level.

Glover, F., Laguna, M., 1997. “Tabu Search”. Kluwer, Norwell, MA.

Holland, J.H., 1975. “Adaptation in Natural and Artificial Systems”, University of Michigan Press.

Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P., 1983. “Optimization by Simulated Annealing, Science”. 220 (4598), 671-680.

Olivera, F., 2002. “Map Analysis with Networks”. http://ceprofs.tamu.edu/folivera/GISCE/
Spring2002/Presentations/NetworksIntro.ppt (Accessed on February 10, 2007).

Orlin, J., 2003. “Dijkstra’s Algorithm Animation”. MIT OpenCourseWare, Network Optimization, Spring 2003.
http://ocw.mit.edu/NR/rdonlyres/Sloan-School-of-Management/15-082JNetwork-OptimizationSpring2003/FC13EFA1-0FE2-4BFB-B019
8939606EDDCC/0/dijkstrasalgorithm.pdf (Accessed on February 10, 2007).

Parker M., 2001. “Planning Land Information Technology Research Project: Efficent Recycling Collection Routing in Pictou County”.
www.cogs.ns.ca/planning/projects/plt20014/images/research.pdf (Accessed on February 10, 2007).

Rice, M., 2006. “ArcGIS Desktop - Extension – Network Analyst forum”.

http://forums.esri.com/Thread.asp?c=93&f=1944&t=187632&mc=21#msgid565389 (Accessed on February 10,2007).

Stewart, L.A., 2004. “The Application of Route Network Analysis to Commercial Forestry Transportation”. http://gis.esri.com/library/userconf/proc05/papers/pap1309.pdf (Accessed on February 10, 2007).