UCGE Reports

Number 20238

Department of Geomatics Engineering

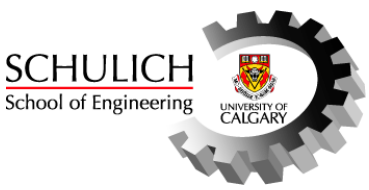
Incremental Routing Algorithms For Dynamic Transportation Networks

(URL: http://www.geomatics.ucalgary.ca/research/publications/GradTheses.html)

by

Qiang Wu

January 2006



UNIVERSITY OF CALGARY

Incremental Routing Algorithms For Dynamic Transportation Networks

by

Qiang Wu

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

GEOMATICS ENGINEERING DEPARTMENT

CALGARY, ALBERTA, CANADA

January, 2006

Abstract

Over the last decade, the rapidly growing population of mobile data terminals, integrating GIS and GPS, has given rise to a new type of real-time spatiotemporal service called Location-Based Service (LBS). Routing is an important function of LBS and Geographic Information Systems for Transportation (GIS-T). It is used in many land-based transportation applications, such as the Intelligent Vehicles Navigation System (IVNS).

There are two common types of queries for navigation service. The first query deals with finding an optimal route from the current location to the desired destination. The other query allows users to locate the closest facility of a certain category, such as the nearest hotel, hospital or gas station, without knowing the destination in advance. In this case, the best destination and an associated optimal route need to be found based on network distance. The challenge lies in the fact that traffic condition always changes such that the optimal route has to be recomputed from time to time in order to adapt to the dynamic environment. Since both traffic condition and the current positions of mobile users change over time, existing shortest path algorithms are either incapable of solving this problem, or are too complicated and time consuming.

To address the above challenge, two incremental shortest path algorithms have been proposed to efficiently deal with the two types of queries. For the first query type, an incremental A* algorithm is designed to adaptively derive the optimal path to the desired destination by making use of previous search results. For the second query type, a shortest path based dynamic network Voronoi diagram is devised to implement a service area for each facility. The corresponding shortest path is derived and maintained dynamically using the incremental approach. The experimental results demonstrate that the proposed incremental search approach considerably outperforms the traditional method which recomputes the shortest path from scratch each time without utilization of the previous search results.

Acknowledgments

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 thesis supervisor, Professor Bo Huang, for his numerous insightful comments which helped me improve the quality of the thesis and for his motivation and support.

I am grateful to my friends Chenglin Xie and Qiaoping Zhang for their friendship and advice. They provided some very useful information regarding my research.

Finally, I would like to thank my friend Matthew Reid for his editorial advice.

Table of Content

List of Figures

List of Tables

Chapter 1: Introduction

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 [1]. In working towards this goal, dynamic traffic routing is required since the traffic conditions change over time.

Up-to-date real-time information about traffic conditions can be collected through loop detectors, probe vehicles and video 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 GIS and Location Based Service

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 [2].

The last decade has witnessed the rapid emergence of Internet-enabled mobile terminals (smart phones, PDAs, in-car computers, etc), mobile/embedded computing and spatial information technologies led by GIS and GPS. As a result, a new generation of mobile services known as Location-Based Services (LBS) have been developed, which are capable of delivering geographic information and geo-processing power to mobile users via the Internet and wireless network in accordance with their current location. The standardization work related to location-based services was started by the OpenGIS Consortium [3], as well as global industry initiatives, such as the Location Interoperability Forum (LIF), formed by Motorola, Ericsson and Nokia [1].

The architecture of location based services consists of three parts:

- Positioning of mobile terminals based on either GSM/GPRS/UTMS mobile communication systems or GPS/GLONASS/Galileo satellite positioning systems.
- Wireless communication networks based on GSM/GPRS/UTMS.
- Internet GIS that provides spatiotemporal data and services over the Internet.

With the expansion and proliferation of LBS, 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. Therefore, 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, mobile clients derive their own paths using on-board computers, based on either static road maps in CD-ROMs, or real-time traffic information provided via wireless network. However, mobile networks have high costs, limited bandwidth, and low connection stability making it expensive to deliver detailed traffic information to all mobile users. As well, geo-processing is time-consuming and mobile terminals usually have limited memory and computational power. Therefore, it may take a long time to perform the computation locally or may even be impossible in some cases. On the other hand, navigation services are often used in time-critical circumstances (e.g. 911 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) to answer path queries submitted by mobile clients. 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 mobile clients. In this thesis, I discuss 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. I will focus my research 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 time, without knowing the destination explicitly.

- Routing query for known destination

For this query, the mobile client 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 travel from the airport to the conference centre, 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 mobile user) 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, mobile clients 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 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 Motivation of the Research

It seems that little attention has been paid to the problems associated with the two types of queries discussed in the previous section. Most existing dynamic algorithms are either incapable of solving these problems, or too complicated and time consuming. Considering the limitations of these dynamic algorithms, how to *reuse*

previous searching results to answer *en route queries* is the emphasis of my research. Next I will discuss the challenge to solving these problems and briefly describe my solutions.

The main challenge to solving the problems associated with the above queries lies in the fact that traffic conditions are not static. The optimal route has to be recalculated from time to time in order to adapt to the dynamic environment. Because the traffic conditions may only partially change between sequential time intervals, some searching results stay the same and do not need to be recomputed. This fact makes it is possible to use the unaltered portion of previous search results to facilitate subsequent searches with minimal computational cost. Therefore the motivation of my research is to try to devise an approach, which can reuse information from previous searches to more efficiently perform path planning for a series of similar routing queries than is possible by solving each path planning problem from scratch.

Some existing dynamic routing algorithms are capable of using previous search results in subsequent searches in order to reduce computation time, but they only compute the new optimal path for each time interval based on a fixed starting point. If the starting point (query point) changes, the previous search results are invalid and cannot be reused in subsequent searches. To compute the new optimal path based on the current position, they have to search from scratch similar to static methods and they lose their strength. Hence, they are not able to efficiently answer the en route query.

The LPA* algorithm is a dynamic shortest path algorithm, which computes dynamic shortest paths between a fixed origin and destination. In other words, it is able to adjust the optimal route to adapt to the dynamic transportation network, but the origin cannot be changed. To answer an en route query for a known destination, I improve the existing LPA* algorithm and make it capable of handling the dynamic routing queries based on the changing traffic conditions and current positions of mobile

clients. In addition, my approach also improves the searching performance.

To deal with the routing queries about the closest facility, I take the advantage of the Voronoi diagram to find the best destination (e.g., hotels, restaurants, etc.) and derive the respective route dynamically and efficiently. Voronoi diagrams benefit the transportation field in that it is easy to identify the closest facility for multiple mobile users located in the same Voronoi cell constructed using network distance. Therefore, Voronoi diagrams are able to provide batch service for nearest neighbor queries, and the performance is not significantly affected by an increase in the number of mobile clients.

Although the majority of Voronoi diagram applications are based on Euclidian distance in the 2-D plane, previous research has shown that there is a straightforward equivalent in graph theory called the network Voronoi Diagram, which is based on the shortest paths from Voronoi sites to other locations. Network Voronoi Diagrams can be used to identify the closest facility in the road network. Since the closest facility may vary for each time interval due to the mobility of mobile clients and changes in traffic conditions, the challenge is how to frequently modify the network Voronoi diagram to adapt to the dynamic environment without significant geo-computation. Meanwhile, the adaptive shortest path trees from every location to their respective closest facility (e.g., hotels, restaurants, etc.) need to be derived and adjusted. To date, few researchers have discussed this problem and there no effective algorithm has been proposed.

To solve this problem, I propose a novel Incremental Parallel Dijkstra's algorithm (IP-Dijkstra for short) to construct and maintain a shortest path based dynamic Voronoi diagram for time-dependent traffic networks. As a result, I implement a dynamic service area for each facility. The service areas can then be used to answer the closest facility queries and provide adaptive route guidance based on current client position and traffic conditions.

1.6 Outline of this Thesis

The rest of the thesis is organized as follows: Chapter 2 introduces the background of graph theory and discusses various network data models. Chapter 3 analyzes the shortest path problem, the search strategies and describes some existing shortest path algorithms for both static and dynamic networks. In Chapter 4, I first introduce the existing LPA* algorithm and describe my improvement to answer the first type en route query for known destination. Chapter 5 discusses the nearest neighbor problem and the common used approaches. Chapter 6 illustrates my proposed method to answer the second type en route query for unknown destination, including the intuition, algorithm. Chapter 7 concludes the thesis.

Chapter 2: Transportation Network Analysis

2.1 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 (see Figure 2.1) 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 $E \in [V]^2$; 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 $E \in [V]^2$. 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 2.1.

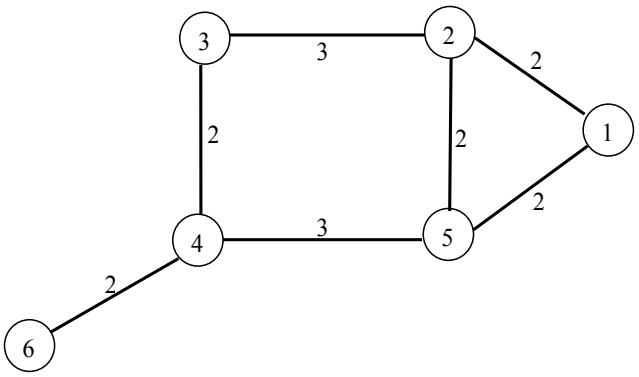


Figure 2.1: A diagram of a weighted graph with 6 nodes and 7 links.

- Degree of a Graph

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 at 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

$$d(G) = \frac{1}{|V|} \sum_{v \in V} d(v) \qquad \textbf{(2.1)}$$

Clearly,

$$\delta(G) \leq d(G) \leq \Delta(G) \qquad \textbf{(2.2)}$$

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,

$$|E| = \frac{1}{2} \sum_{v \in V} d(v) = \frac{1}{2} d(G).|V|, \qquad \textbf{(2.3)}$$

and therefore

$$\varepsilon(G) = \frac{1}{2} d(G) \qquad \textbf{(2.4)}$$

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.

- Definition of a Path

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, i.e., $\sum_{i=1}^{k} W(v_{i-1}, v_i)$. 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.

2.2 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.

2.2.1 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 [4]. 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 in Figure 2.2 is shown below.

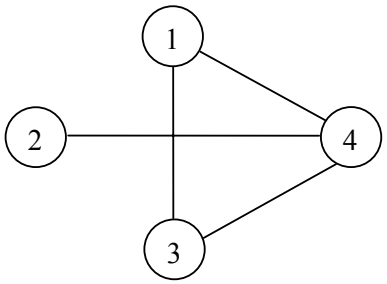


Figure 2.2: An Undirected Graph

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

2.2.2 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 2.2.

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

2.2.3 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 2.2 can be described by Figure 2.3.

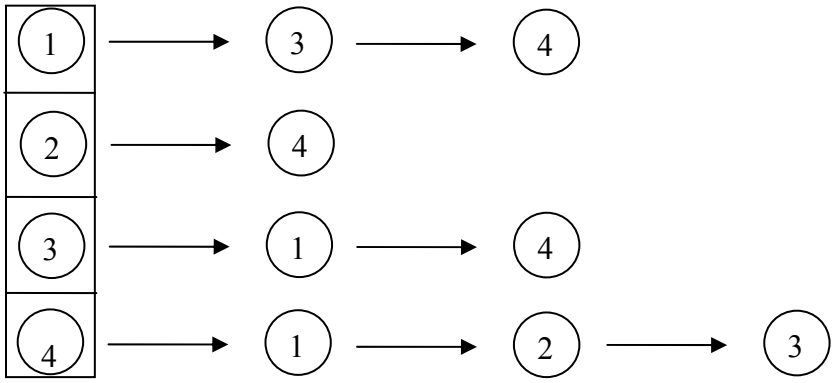


Figure 2.3: An Adjacency List

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.

2.2.4 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, I will 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

$$16 \times (|E|+|V|) \geq |V|^2/8$$

Based on equation (2.1.2) in section 2.1, we have,

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

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

$$d(G) \geq \frac{|V| - 128}{64} \qquad \textbf{(2.5)}$$

This means that the adjacency list representation occupies more space when equation (2.5) holds.

In reality, firstly, most transportation networks are large scale sparse graphs with many nodes but relatively few links as compared with the maximum number possible ($|V| \times (|V| - 1)$ for maximum). That is, there are no more than 5 links ($\Delta(G) \approx 5$) connected to each node. In most cases there are usually 2, 3 or 4 ($\delta(G) = 2$) links, although the maximum links is $|V|$-1 for each node. Secondly, road networks often have regular network structures and a normal layout, especially for well planned modern cities. Thirdly, 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 2.5 holds only if $|V| \leq 448$. In reality, most road networks contains thousands of nodes where $|V| >> 448$. As a result, equation 2.2.1 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 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 2.2 in section 2.1, we obtain,

$$d(G) \leq 2 \qquad \textbf{(2.6)}$$

This means that the adjacency matrix representation occupies more space if and only if equation 2.6 holds. Since the minimum degree of a transportation network is 2 ($\delta(G) = 2$), then equation 2.6 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.

2.3 Chapter 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, are introduced at the beginning of this chapter. In the discussion of the degree of a graph, the dense graph and sparse graph are defined and used in data model discussion.

In the data model discussion, three types of data models for graph representation are 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. In my research, I will utilize an adjacency list to construct topology of the experimental road network in order to implement my routing computations.

Chapter 3: Shortest Path Problem

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.

3.1 The Classification of the Shortest Path (SP) Problem

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 [5] 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.

3.2 Analysis of the Searching Strategy

3.2.1 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 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 3.1.

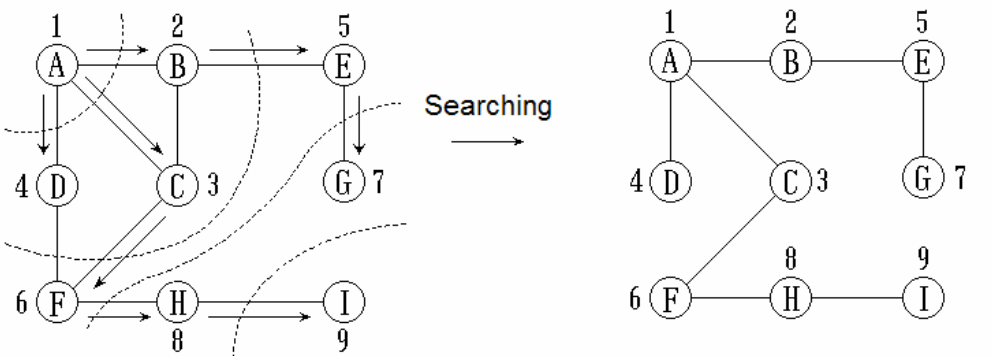


Figure 3.1: Breadth-first Search

3.2.2 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 3.2 shows a DFS applied to an undirected graph, with the nodes labeled in the order they were explored.

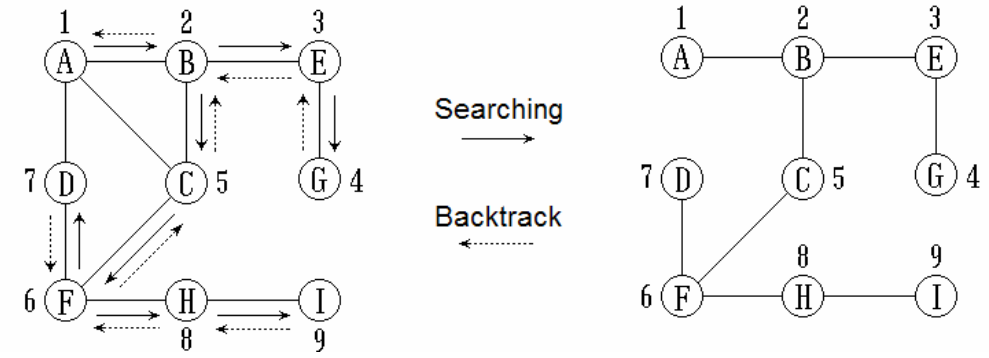


Figure 3.2: Depth-first Search

3.2.3 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.

3.3 Classical Shortest Path Algorithms for Static Networks

Because 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 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 node 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, $P_{k=}\{p_1, ..., p_k\}$, 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.

3.3.1 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 = ∅
3)      S' = V ∈ G
4)      while S' ≠ ∅
5)          do u = Min (S')
6)             S = S ⋃ {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
```

3.3.2 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 not only considers 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 do 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 [6]. 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 [6].

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.

3.3.3 Comparison of Algorithms Based on 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 3.3.1, 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 $|V|+(|V|-1)+...+1=O(|V|^2)$. 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.

Table 3.1 Time Complexity Comparison between Classical Algorithms

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

In section 1.4, 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 time complexity of the Dijkstra algorithms are 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 scenario 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.

3.4 Dynamic Traffic Routing

3.4.1 Dynamic Transportation Network Scenario

Time is an essential part of today's mobile 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 office. 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 weigh 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.

3.4.2 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 [7] first proposed a TDSP algorithm in 1958. The algorithm they suggested is a modified form of Bellman's label [8] correcting the shortest path algorithm. Hall [9] 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 [10] 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 [11]. A number of incremental search methods have been suggested in the algorithms literature [17–28], 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 [12]. 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 [13] (RR for short, also referred to as the DynamicSWSF-FX algorithm) seems to be the most used [14, 15, 16]. It is a fully-dynamic DSP algorithm which updates the shortest paths incrementally. A more detailed description of the algorithm

will be given in section 4.4.3.

In their work on algorithms for the DSP problem, Demetrescu et al. [29] proposed a fully dynamic algorithm, which is a specialization of the RR algorithm for updating a shortest path tree [16]. It is a modification of their previous work on a semi-dynamic incremental algorithm.

In this chapter, I show 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 [29]. Even and Shiloach [30] 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 [31] 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. [32], which is theoretically better than RR, was usually outperformed by RR in computational testing [32].

Many theoretical studies of DSP algorithms have been carried out but few experimental results are known. Frigioni et al. [33] compared the RR algorithm with the algorithm proposed by Frigioni et al. [32] 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 [34].

3.4.3 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:

$$rhs(s) = \begin{cases} 0 & \text{if } s = s_{start} \\ Min_{s' \in pred(s)}(g(s') + w(s, s')) & \text{otherwise} \end{cases} \quad \textbf{(3.4.1)}$$

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.

3.5 Chapter 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: Dynamic Routing Algorithm to Known Destination

To answer a dynamic routing query to a known destination, an efficient and dynamic algorithm is required to solve the single pair shortest path problem. Most dynamic algorithms cannot answer the en route query in a dynamic environment, in which both the traffic conditions and the query point position change over time. My solution is based on modifying the existing LPA* algorithm to make it capable of handling this problem. This modification will also improve the search performance of the algorithm. In this chapter, I will give a detailed description of the LPA* algorithm and the changes I have made to modify the existing algorithm for use in a dynamic routing environment.

4.1 LPA* Algorithm

The Lifelong Planning A* (LPA*) algorithm is an incremental version of A* that uses a heuristic, *h(s)*, to control its search. The first search of LPA* is the same as that of A*, but all subsequent searches are much faster because it reuses those parts of the previous search tree that are identical to the new search tree. The main principle of the LPA* algorithm is described in the following statements. 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, $0 < c(s, s') \leq \infty$ denotes the cost of moving from node s to node $s' \in succ(s)$ and $g(s)$ denotes the start distance of node $s \in S$, i.e., the cost of a shortest path from s_{start} to s. As for A*, the heuristic approximates the goal distances of the nodes s. They need to be consistent, i.e., satisfy $h(s_{goal}) = 0$ and $h(s) < c(s, s') + h(s')$ for all nodes $s \in S$ and $s' \in succ(s)$ with $s \neq s_{goal}$.

There are three estimates held by LPA* in its lifetime. The first one is the *g(s)* of the start distance of each node *s*, which directly corresponds to the *g*-values of A* and can be reused in subsequent searches. The second one is the *h(s)* of the approximate distance to s_{goal}, which has the same meaning as the *h*-value in A* and is used to drive the search in the goal direction. The last one is another estimate of the start distance, namely *rhs*-values which are one-step look-ahead values based on the *g*-values and thus are potentially better informed than the *g*-values. They always satisfy the following relationship: $rhs(s) = 0$ when *s* is the start node or $rhs(s) = Min_{s' \in pred(s)} (g(s') + c(s, s'))$ otherwise. As with the A* algorithm, each node is locally consistent if its *g*-value equals its *rhs*-value. This concept is important because the *g*-values of all nodes equal their start distances if all nodes are locally consistent. Actually, there is no need to make every node locally consistent in LPA*. Instead, it uses the *h(s)* heuristic to converge the search and update only the *g*-values involved in the shortest path computation from s_{start} to s_{goal} [35].

LPA* maintains a priority queue that always exactly contains the locally inconsistent nodes. These are the nodes whose *g*-value may need to be updated in order to make them locally consistent. The node keys in the priority queue correspond to the *f*-values used by A*. Similar to A*, LPA* always expands the node in the priority queue with the smallest key (*f*-value). The key, *k(s)*, of node *s* is a vector with two components: $k(s) = [k1(s); k2(s)]$, where $k1(s) = Min (g(s), rhs(s)) + h(s)$ and $k2(s) = Min(g(s), rhs(s))$. Similar to A*, LPA* always expands the node in the priority queue with the smallest *k1*-value (*f*-value). Any ties are broken by favoring the node with the smallest *k2*-value (*g*-value). The resulting behavior of LPA* and A* is also similar. LPA* expands nodes until s_{goal} is locally consistent and the key of the node set for expansion is no less than the key of s_{goal}.

As shown in Figure 4.1, the goal is to find the shortest path from A to K in the graph. The upper-left graph gives the weight for each link. For illustration convenience, the start distance and heuristic are also given in the brackets near each node. When LPA*

performs the first search, it initializes the *g*-value and *rhs*-value of all nodes as infinity. Actually, we cannot initialize all of the nodes in a large map and only initialize each node whenever we encounter it while searching. In the following iterations, there is also a bracket for each node: the two values denote the *k1*-value and *k2*-value respectively. The number above the bracket is the start distance (*g*-value). Any single values in the brackets denote the *g*-value of the nodes which are locally consistent. The black square indicates a node that is being visited in the current iteration. In this example, I use the Manhattan distance between any node and goal node as the heuristic for LPA*.

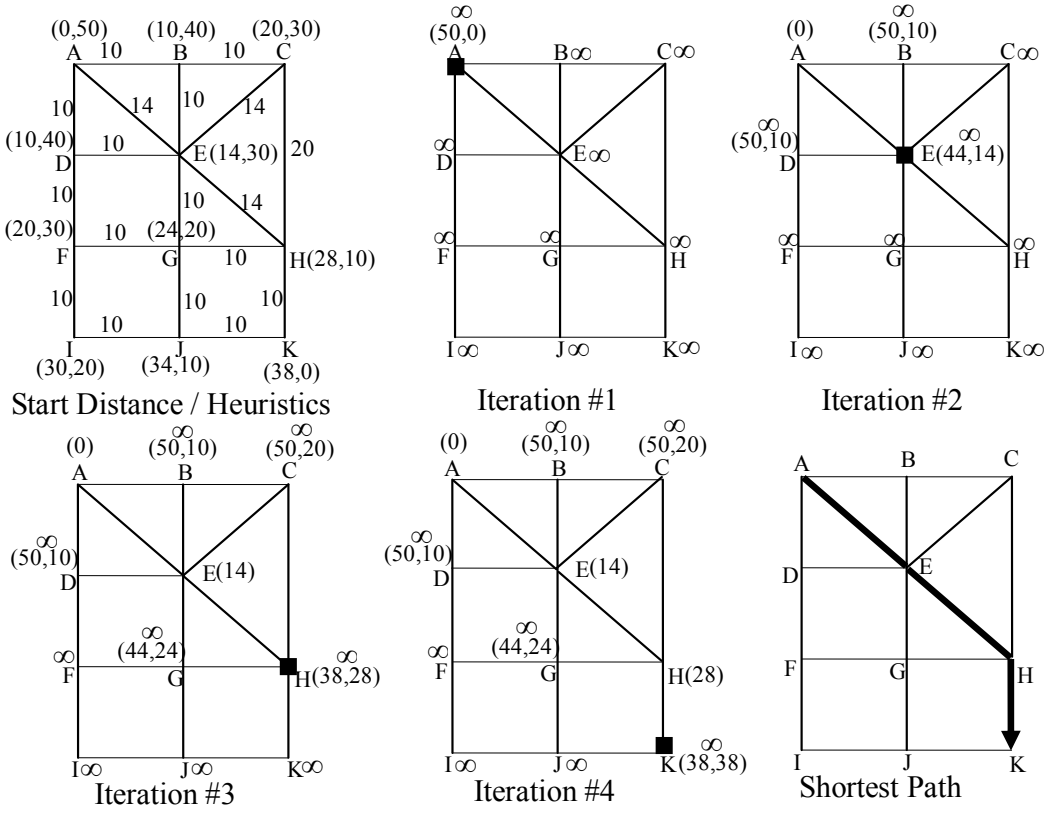


Figure 4.1: LPA* First Search

In iteration #1, the search expands from start node A, finds three successors (B, E and D), assigns their keys and inserts them into a priority queue. They are ordered in the queue based on the value of their keys. Next, the node with the smallest priority is taken (popped) from the priority queue. In our example, the node with the smallest

priority is node E (*k1*=44). The node is now locally consistent and has been popped from the priority queue. In the same way, the search expands to the nodes C, H, and G. In this iteration, *rhs*(C) has been updated by 20 because the smallest *g*-value of its neighbors is *g* (B) =10, and its parent is assigned as B. Hence, we maintain the shortest path from the start node to each visited node. Finally, H (*k1*=38) is popped from the priority queue. The search terminates when node K is reached and it is locally consistent, as any node expanded from K does not have a smaller key than that of K.

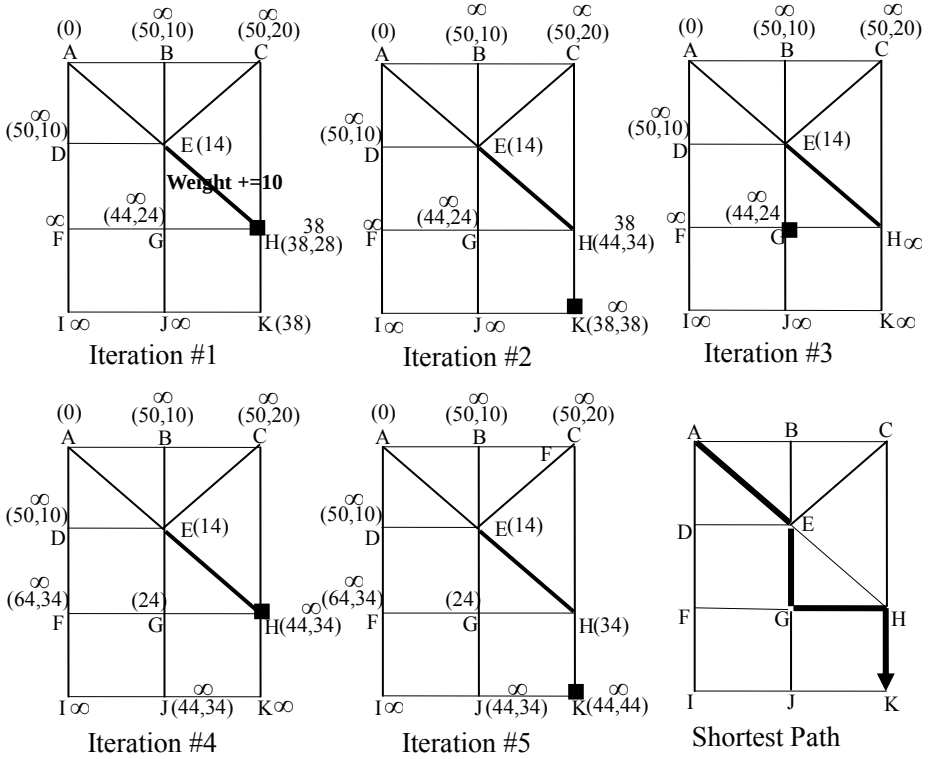


Figure 4.2: LPA* Second Search

Figure 4.2 is an example showing what happens when the weight of any link arbitrarily changes. In this case, the weight of the link EH increases by 10. To adapt to this change, we first check the estimates (*g, rhs*) of the nodes around the Link EH, which have the most potential to be affected by this change. They are nodes E and H. Here, node E is not affected by this change, but the start distance of node H changes (*g*(H)=38) and its *rhs*-value changes to 34 after updating. The next step is to update

node K, and then it becomes locally inconsistent. Next, node G is popped from the priority queue. By expanding nodes G to H and J, the search is led to the current shortest path without visiting many unnecessary nodes that are not affected by the changes. In this way, LPA* reuse the calculation result coming from the last search and facilitate faster route recalculation by incrementally updating the locally inconsistent node.

The main advantage of LPA* is the capability of carrying forward the start distances (*g*-value) and reusing them from search to search. Although the LPA* can efficiently manage dynamic environments, it cannot deal with start node positions changing over time. While the mobile user is moving along the previous shortest path and querying new optimal routes to adapt to changes in the environment, the LPA* is not able to perform an incremental search as the start distances (*g*-value) are no longer valid for the current start node. With the current method, it is impossible to rebuild the *g*-values for these nodes unless an independent search is performed from scratch which loses the power of LPA*.

4.2 Improved LPA* Algorithm

4.2.1 Extend LPA* with Changing Starting Point

The start distance (*g*-value) of a node is very important in LPA*. In order to utilize the advantages of LPA*, the key issue is how to retain the start distance of a node that was assigned in the last search. It is important to note that the destination does not change when the start node changes. Inspired by this, I have modified the original LPA* to this way. Now, when a user will move from node *v* to *w* (*v, w*$\in$*S*) and intend to compute the shortest path between them, we can switch the search direction. In other words, we do not search from node *v* to *w*, but assign *w* as the source and search from *w* to *v*. In this situation, the start node will not change, but the goal always changes. Hence, the start distance of a node can be carried forward and reused from

search to search. With the goal changing, the heuristic of each node should be modified according to the new goal. No matter what metrics are employed as heuristics, either the Manhattan distance or Euclidean distance, they are can be easily updated for each node in the priority queue. Note that one should adopt the weight of the opposite direction in a directed graph to ensure that the final shortest path leads from the true start node to the goal when calculating the start distance for each node. Therefore, my contribution is extending LPA* to a new application.

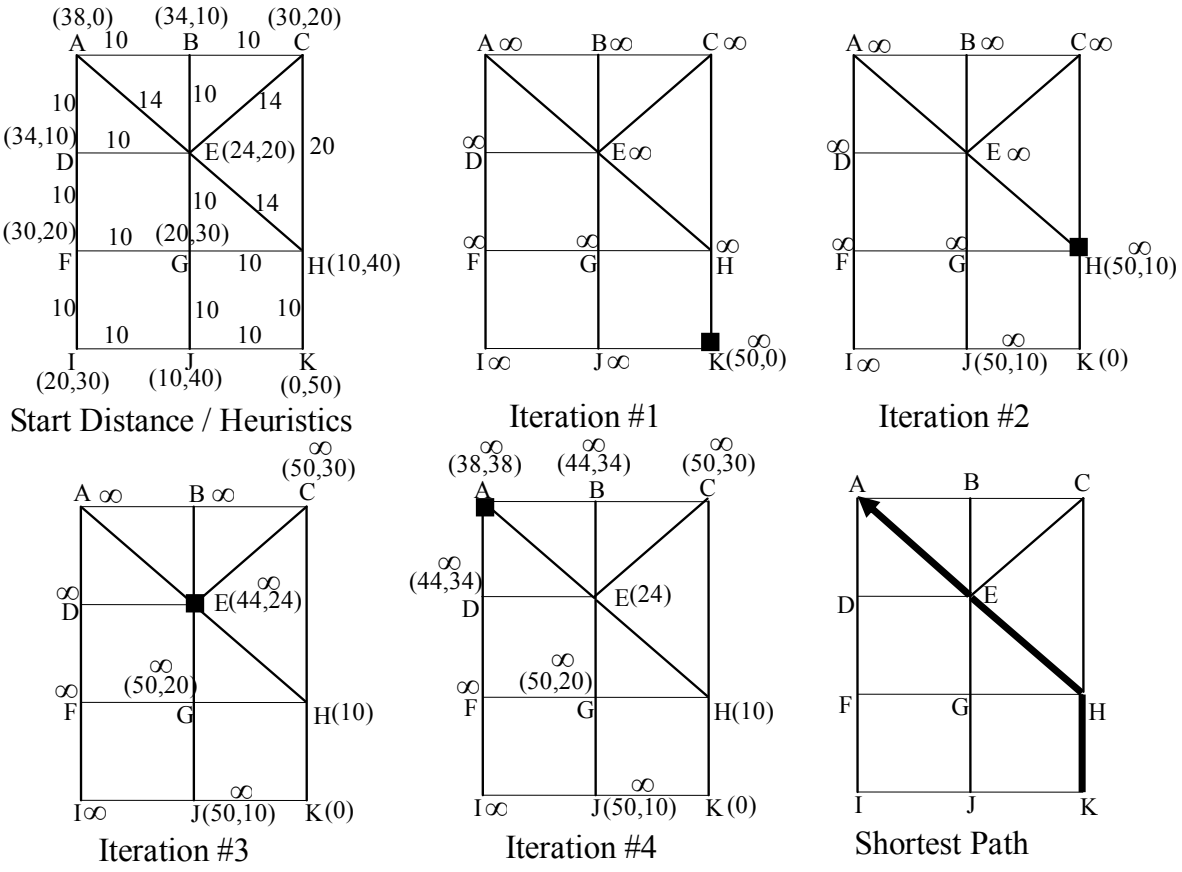


Figure 4.3: Improved LPA* First Search

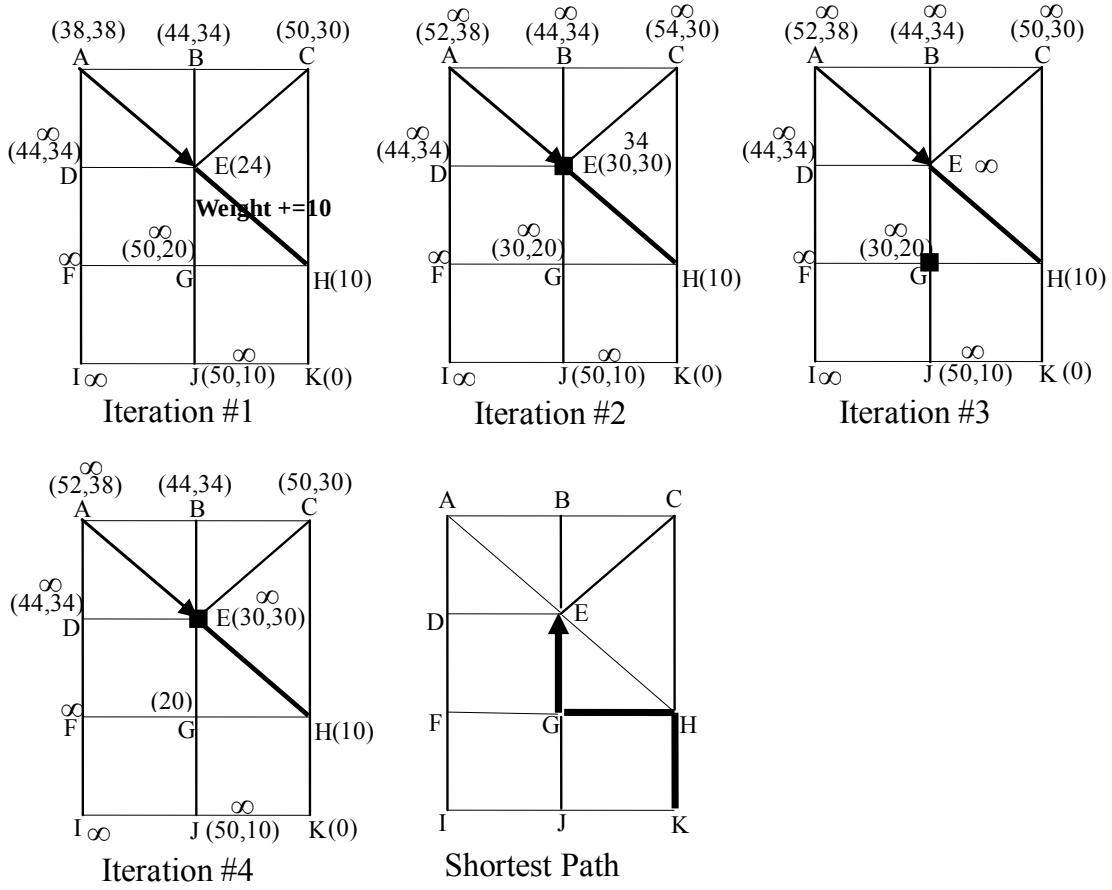


Figure 4.4: Improved LPA* Second Search

I will illustrate my approach using the graphs in Figure 4.3 and Figure 4.4. Imagine that a mobile user wants to move from A to K. In the first LPA* search (see Figure 4.3), we follow the reversed direction to perform an LPA* search from K to A and acquire the start distance of the involved nodes.

As shown in Figure 4.4, the mobile user starts from A moving along the designed optimal route. When the user reaches node E, current traffic conditions are provided with information of a traffic jam in the link EH. In this graph, it is represented with an increase of 10 in the link cost. To determine the optimal path in this case, we need to update the *g*-value and *rhs*-value for node E because its *g*-value comes from H. Obviously, it is locally inconsistent. Node G is then popped from the priority queue and again expanded to E. Thus the new route is successfully recalculated. The actual

details of my algorithm are described below.

The pseudo-code uses the following functions to manage the priority queue:

U.TopKey() - returns the smallest priority of all nodes in priority queue *U*. (If *U* is empty, then U.TopKey() returns $[\infty;\infty]$.)

U.Pop() - deletes the node with the smallest priority in priority queue *U* and returns the node

U.Insert(*s, k*) - inserts node *s* into priority queue *U* with priority

U.Remove(*s*) - removes node *s* from priority queue *U*.

Swap(s_{start}, s_{goal}) - switch the start and goal node to perform reversed search

Procedure CalculateKey(*s*)

return [min(*g*(*s*), *rhs*(*s*)) + *h*(*s*);min(*g*(*s*), *rhs*(*s*))];

Procedure Initialize()

$U = \varnothing$;

for all $s \in S$ $rhs(s) = g(s) = \infty$;

$rhs(s_{start}) = 0$;

U.Insert(s_{start}, [$h(s_{start})$; 0]);

Procedure UpdateNode(*u*)

if ($u \neq s_{start}$) $rhs(u) = \text{Min}_{s' \in pred(u)}(g(s') + c(s', u))$;

if ($u \in U$) U.Remove(*u*);

if ($g(u) \neq rhs(u)$) U.Insert(*u,* CalculateKey(*u*));

Procedure ComputeShortestPath()

while (U.TopKey() ˙<CalculateKey(s_{goal}) OR $rhs(s_{goal}) \neq g(s_{goal})$)

{

u = U.Pop();

if ($g(u) > rhs(u)$)

$g(u) = rhs(u)$;

for all $s \in succ(u)$ UpdateNode(*s*);

Else

$g(u) = \infty$;

for all $s \in succ(u) \cup \{u\}$ UpdateNode(s);

}

Procedure Main()

Initialize();

Swap(s_{start}, s_{goal});

while ($s_{start} \neq s_{goal}$)

{

ComputeShortestPath();

s_{start} =Top*(Pathlist)*.next

Move to s_{start}

Detect the weight change in graph

If any change occurs

for all directed links (u, v) with changed link costs

Update the link cost $c(u, v)$;

UpdateNode(v);

for all $s \in$ U

U.Update*(s,* CalculateKey(s));

}

4.2.2 Constrained Shortest Path Search

To further improve the efficiency of my proposed method, I also apply additional constrained conditions to converge the search space for the LPA* algorithm. Since an ellipse is the simplest geometric shape we can employ besides a circle to deal with distance, I use some of the features of an ellipse to restrict the search space while the LPA* is performing subsequent searches.

An ellipse is the trajectory of all points whose distances to two specific points (i.e., the two foci of that ellipse) are fixed and equal to the length of the major axis. All points inside the ellipse are nearer to the two foci than those on the ellipse, while all points outside the ellipse are farther from the two foci. As you can see in Figure 4.5, if we know the network distance *d* between two nodes *s* and *g* in the graph, we can use *d* as the major axis, take the position of the two nodes *s* and *g* as foci and draw an ellipse. We can assert that if there is a shortest path existing between *s* and *g*, this path must lie in the ellipse. To prove it, we assume that there is a node *v* that belongs to the shortest path (*s, g*) and is located outside the ellipse, then (*s, v*) + (*v, g*) > *d*. Even if there are straight-line paths existing between (*s, v*) and (*v, g*) respectively, the length of this route must be greater than *d*. Therefore, node *v* cannot lie in the shortest path between *s* and *g*.

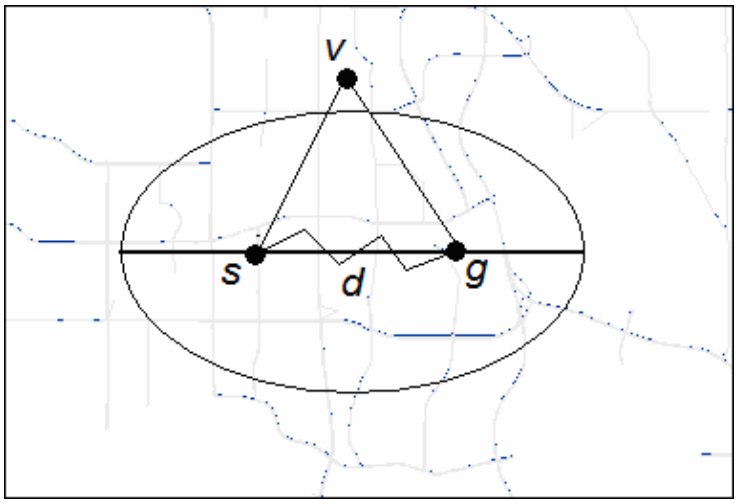


Figure 4.5: Shortest Path Constrained by Ellipse

Based on this theorem, we can use an ellipse during searching to prune the nodes which do not belong to the expected shortest path. The next issue is how to determine the size of the ellipse. From the first LPA* search, we are aware of the shortest path along which the mobile user travels. While the mobile user will try to find another optimal path if the weight of any link changes, it is easy to derive the network distance between the current node and the goal based on the new weight value. It is possible to define an ellipse by using this network distance as the major axis and

employing the goal and current node as foci. Thus, if there is an existing shorter alternative path, it must lie in the ellipse. Any node beyond this ellipse can be safely pruned from the search space and thus the efficiency of the LPA* can be improved with the assistance from a constrained ellipse.

However, using an ellipse directly as a constraint condition is less efficient since it involves many power and evolution computations. To solve this problem, we utilize the Minimum Bounded Rectangle (MBR) of the constrained ellipse to simplify the calculation as shown in Figure 4.6. The following example will present how to compute the MBR for a given ellipse. Suppose an ellipse has two foci ($x1; y1$) and ($x2; y2$), and a major axis, the ellipse can be represented by equation 4.1. If partial derivatives of x and y for the ellipse equation are used, we can obtain the extreme values of xm and ym from equation 4.3.

$$\frac{[Cos\theta(x-x_c)+Sin\theta(y-y_c)]^2}{a^2}+\frac{[-Sin\theta(x-x_c)+Cos\theta(y-y_c)]^2}{b^2}=1 \quad \textbf{(4.1)}$$

$$\text{Where } \theta=\arctan(\frac{y_2-y_1}{x_2-x_1}) \quad x_c=\frac{x_1+x_2}{2} \quad y_c=\frac{y_1+y_2}{2} \quad c=\frac{\sqrt{(y_2-y_1)^2+(x_2-x_1)^2}}{2} \quad \textbf{(4.2)}$$

$$b=\sqrt{a^2-c^2}$$

$$\text{and } x_m=x_c\pm\sqrt{a^2\cos^2\theta+b^2\sin^2\theta} \qquad y_m=y_c\pm\sqrt{a^2\sin^2\theta+b^2\cos^2\theta} \quad \textbf{(4.3)}$$

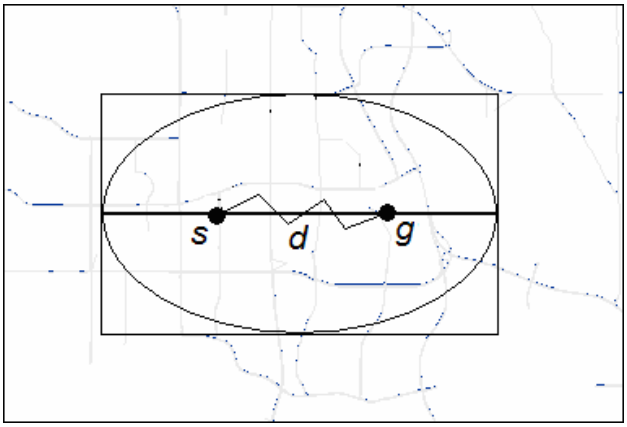


Figure 4.6: MBR Constrained Search

The ellipse is bounded by the MBR with corner coordinates of (*xm*, *ym*). When combined with the LPA*, the pseudo-code described above should be modified as follows:

Procedure UpdateNode(*u*)

if (not check(*u,*MBR)) return;

if ($u \neq s_{start}$) $rhs(u) = \text{Min}_{s' \in pred(u)}(g(s') + c(s', u))$;

if ($u \in U$) U.Remove(*u*);

if ($g(u) \neq rhs(u)$) U.Insert(*u,* CalculateKey(*u*));

Procedure Main()

Initialize();

Swap(s_{start}, s_{goal});

while ($s_{start} \neq s_{goal}$)

{

ComputeShortestPath();

s_{start} =Top*(Pathlist)*.next

Move to s_{start}

Detect the weight change in graph

If any change occurs

calculate_MBR(s_{start}, s_{goal})*;*

for all directed links (*u, v*) with changed link costs

Update the link cost *c*(*u, v*);

UpdateNode(*v*);

for all $s \in$ U

U.Update*(s,* CalculateKey(*s*));

}

4.3 Software Implementation

4.3.1 Development of an Interactive Environment

Since it is inconvenient to examine the performance of my algorithms utilizing existing GIS software, such as ArcGIS, I implemented the algorithms using the VC++ programming language. This software is designed to load datasets directly from shapefiles and to provide a user-friendly interactive interface to facilitate experimental studies. The interactive interface allows user to select the start point and destination arbitrarily in order to perform shortest path computations. After a shortest path is obtained, the user can then select an intermediate point randomly or manually along the path as a stop and assign a percentage weight change. The stop is used to simulate the situation of a mobile user submitting an en route query for a new optimal route in order to adapt to the arbitrary change in traffic conditions.

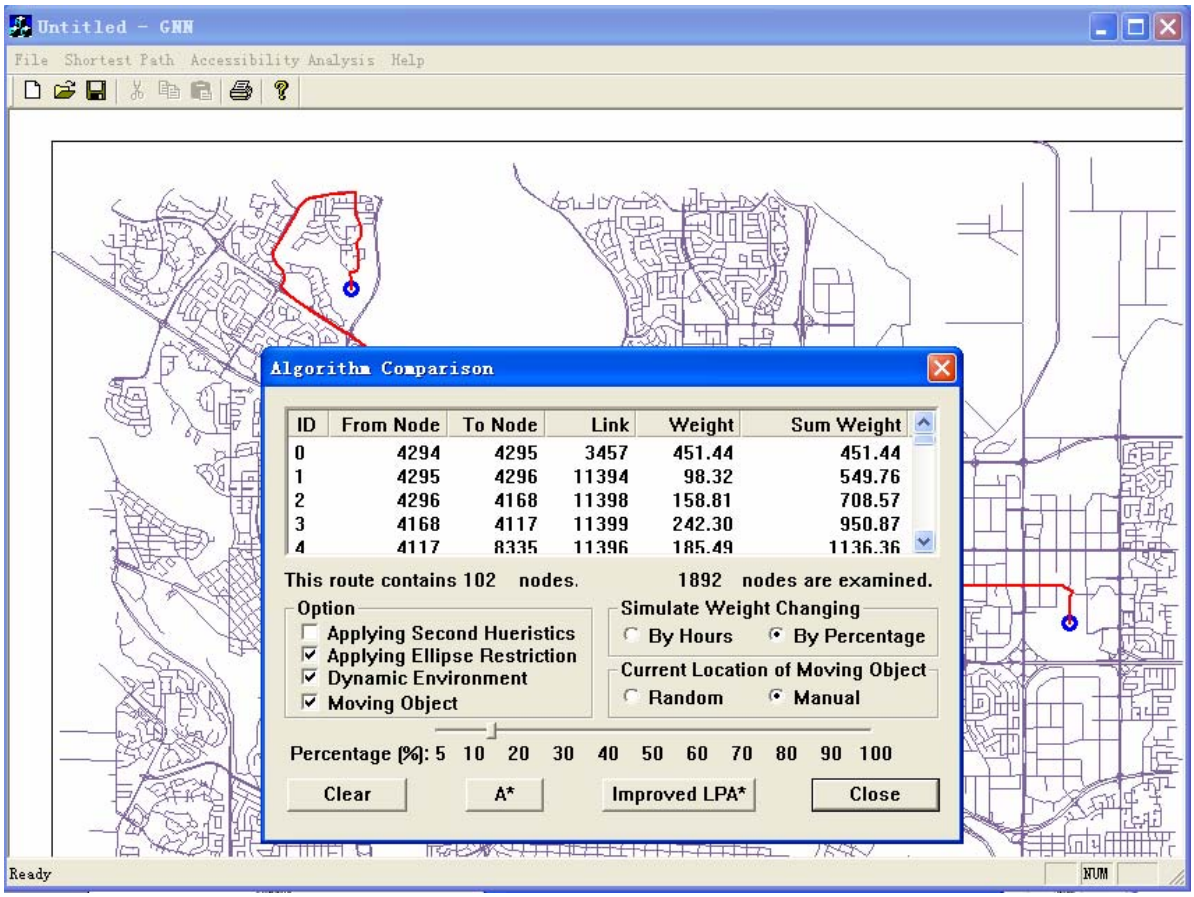


Figure 4.7: Software Interface

4.3.2 C++ Class Implementation

The key elements of any object-oriented software design is the use of structured, efficient data structures, referred to as objects. These objects are implemented in practice using *class* definitions. One advantage of using objects to model the problem domain is that they mirror the physical reality. In my program, the topology of the network is stored in a linked list which is implemented as a class. In addition, all algorithms are also implemented as classes, including A*, LPA*, IP- Dijkstra.

As an example, the node class is described below. To accommodate all algorithms, the class definition includes all variables which are used to support the different algorithms.

```
class _asNode {
public:
    _asNode(double a = -1,double b = -1) : x(a), y(b), Id(0), ChildrenNum (0)
    {
        parent = next = NULL;
        memset(children, 0, sizeof(children));
    }
    ~_asNode();
    int         Id;             //   Node Id
    double      x, y;           // coordinates
    double      f, g, h, rhs;   // estimates for A*, LPA*, IP- Dijkstra
    _asNode     *parent;        // predecessor of the node
    int         ChildrenNum;_   // Number of Children
    asNode      *children[5];   // 5 Children allowed, the maximum degree
    double      weight[5];      // link weights
    double      Key;            // measure the priority of the node in LPA*
    int         V_Id;           // corresponding Voronoi site in IP- Dijkstra
};
```

4.3.3 Priority Queue and Binary Heap

4.3.3.1 Priority Queue

For all algorithms mentioned in this paper, a priority queue is needed to determine the search strategy and optimize the shortest path computation. In general, the priority queue is called an open list, which contains the not-yet-examined nodes. At each iteration, the top node is retrieved and expanded. Meanwhile, this node also needs to be removed from the queue. In my implementation, I adopt the binary heap to realize the priority queue. This will be discussed in more detail later in the thesis. In following discussion, I use the term "key" to denote the priority of the node, which may have a different meaning for different algorithms. In Dijkstra's and IP- Dijkstra's, it refers to the start distance *g(s)* of node *s;* for A*, it is the estimates $f = g(s) + h(s)$; in LPA*, it has two components: $key = [k1(s); k2(s)]$, where $k1(s) = Min(g(s), rhs(s)) + h(s)$ and $k2(s) = Min(g(s), rhs(s))$. The critical issue is how to save and retrieve nodes according to its key.

A simple way to save an open list is to keep the list sorted. This speeds up node removal. We just need to grab the first node from the list that has the lowest key. However, every time we add a node to the list, we need to insert it in the proper place. A naive approach would be to start at the beginning of the list every time we need to add a new node and then successively compare the key of the current node we are adding to the list with each node already in the list. Once we find a node in the open list with an equal or higher key value, we could insert the new node before that node in the list. There are many methods to keep the list sorted, such as selection sorts, bubble sorts, quick sorts, etc.

This approach could be improved by keeping track of the average key value of the nodes already in the list, and using that to decide whether to start at the beginning of the list (insertion of new nodes with a lower key than the average) or to start at the end of the list and work toward the front of the list (insertion of new nodes with a

higher key than the average). This approach will save the search time in half.

A more complicated, but faster approach could be to use a quick sort algorithm, which basically starts by comparing the key of the new node to the node at the middle of the list. If the new node has a lower key, we would then compare it to the node 1/4 of the way through the list. If the key was lower than this key, we would compare it to the one 1/8 of the way through the list, and so on. This algorithm successively divides the list in half and compares the new node to the current nodes in the list until it finds the proper place for the new node.

4.3.3.2 Binary Heap

A binary heap is very similar to the quick sort method described above. Using a binary heap can significantly speed up path searches, especially on large road maps with many nodes

In a sorted list, every node in the list is in its proper order, lowest-to-highest. This is helpful, but it is actually more than we really need. We don't actually care about the order of the entire list. All we really need is the node with the highest priority (lowest key) to be easily accessible at the top of the list and the rest of the list can be unsorted. Always keeping the rest of the list properly sorted is not necessary until the next node is needed.

In general, a binary heap is a bunch of items where either the lowest or highest key item is at the top of the heap. Since we are looking for the lowest key node, we will put that at the top of our heap. This node has two children, each of which has a key equal to, or a little higher than itself. Each of these children has two children of its own that has a key that is equal to, or a little higher than it, and so on. Figure 4.8 is an example of a binary heap.

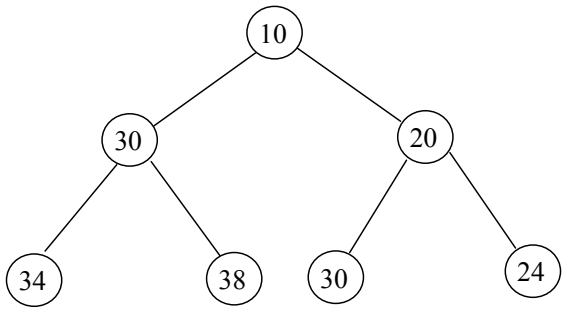


Figure 4.8: An Example of a Binary Heap

Notice that the lowest node in the list (10) is at the top and the second lowest (20) is one of its children. The third lowest node in the list is 24, which is two steps down from the top. It is also lower in the list than 30, which is only one step from the top on the left side. It doesn't matter what the value of the other nodes are in the heap, each individual node in the heap needs only to be equal to or higher than its parent, and equal to or lower than both of its children. Those conditions are met here, so this is a valid binary heap.

The major reason to use a binary heap is that it is very easy to use. It can be saved in a simple, one dimensional array. In this array, the node at the top of the heap would be in the first position of the array (position 1, not position zero, which is possible in an array). Its two children would be in positions 2 and 3. The four children of these two nodes would be in positions 4-7. Figure 4.9 describes the order of a binary heap.

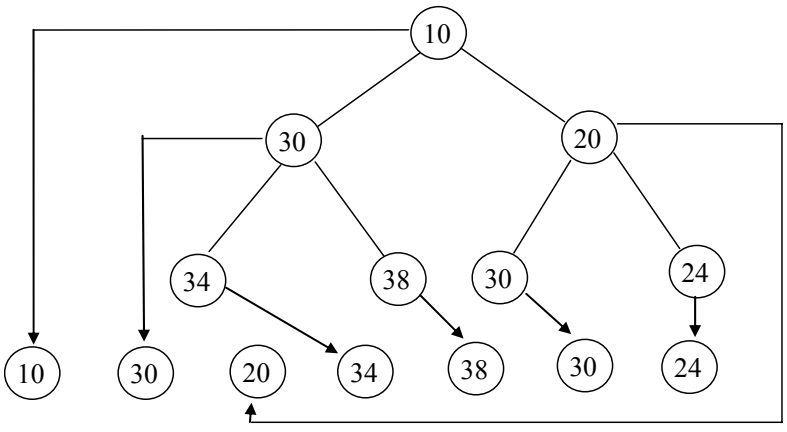


Figure 4.9: The Order of a Binary Heap

In general, the two children of any node in the heap can be found in the array by multiplying the node's current position in the array by two (to find the first child) and adding one (to find the second child). For example, the two children of the third node in the heap (with a key of 20), can be found in positions 2*3 = 6, and 2*3 +1 = 7 of the array. In this case, the nodes in these positions are 30 and 24, respectively.

- Adding a node to the Heap

In order to add a node to the heap, we place it at the very end of the array. We then compare it to its parent, which is at location (node's number in the heap)/2, rounding all fractions down. If the new node's key is lower, we swap these two nodes. We then compare the new node with its new parent, which is at location (current position in the heap)/2, rounding all fractions down. If its key is lower, we swap again. This process is repeated until the node's key is not lower than its parent, or until the node has bubbled all the way to the top, which is position #1 in the array.

- Removing a node from the Heap

Removing a node from the heap involves a similar process, but in reverse. First, we remove the node in slot #1, which is now empty. We then take the last node in the heap and move it up to position #1. This node is then compared to each of its two children, which are at locations (current position * 2) and (current position * 2 + 1). If it has a lower key than both of its two children, it remains in its current position. If not, we swap it with the lower of the two children. This process is repeated until we have reached the bottom level of the heap.

Since a binary heap uses a hierarchical data structure to store nodes, it can considerably reduce the computational cost of the comparison operations. In terms of the insertion and deletion operations in the priority queue, my algorithms are able to greatly benefit from the binary heap.

4.4 Experimental Studies

4.4.1 Experimental Dataset

To justify the universal validity of my approach, without the favor of certain circumstances or topological structures, the experiments are performed using two real-world road networks: Calgary and Singapore. The Calgary road map contains about 8000 nodes and 12500 links, while the Singapore road map contains about 7000 nodes and 11800 links. For each map, we have a facilities datasets which consists of a set of points representing shopping malls, hotels, restaurants and gas stations. Figure 4.10 and Figure 4.11 show the road maps for Calgary and Singapore that I used in my research.

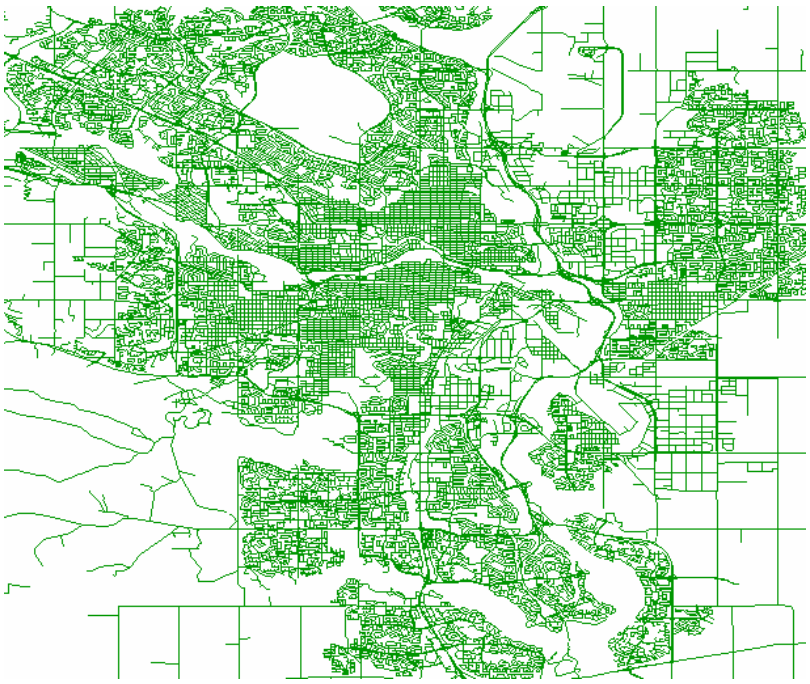


Figure 4.10: Road Network of Calgary

Figure 4.11: Road Network of Singapore

.

4.4.2 Demonstration of En Route Queries for Known Destination

In Figure 4.12, a mobile client submits a query, prior to departure, for the shortest path from B to A and the shortest path is returned (depicted by a blue line). The mobile client then starts traveling along the path. Assuming that the client is informed there is traffic congestion ahead, upon arriving at point C the client can submit a new query and receive an alternative route from C to A (depicted by a red line) in order to avoid any delays. This happens again when point D is reached. The route planner sends the client another optimal route solution (depicted by a black line) based on the user's new position, D, and current traffic conditions. The final path actually traveled by the user is shown in Figure 4.13. The whole process demonstrates how the improved LPA* algorithm works.

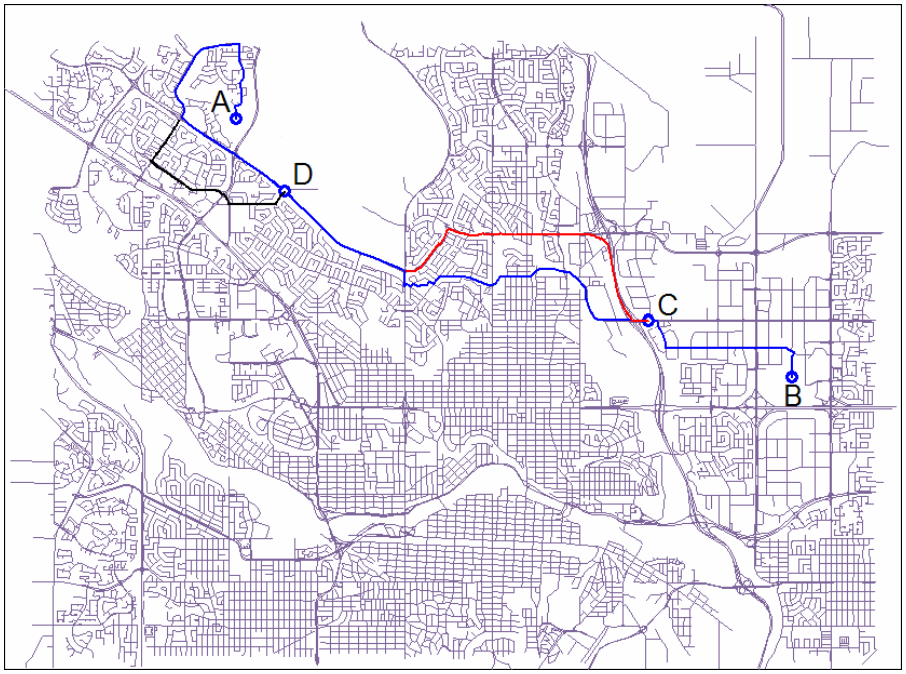


Figure 4.12: Optimal Route Update

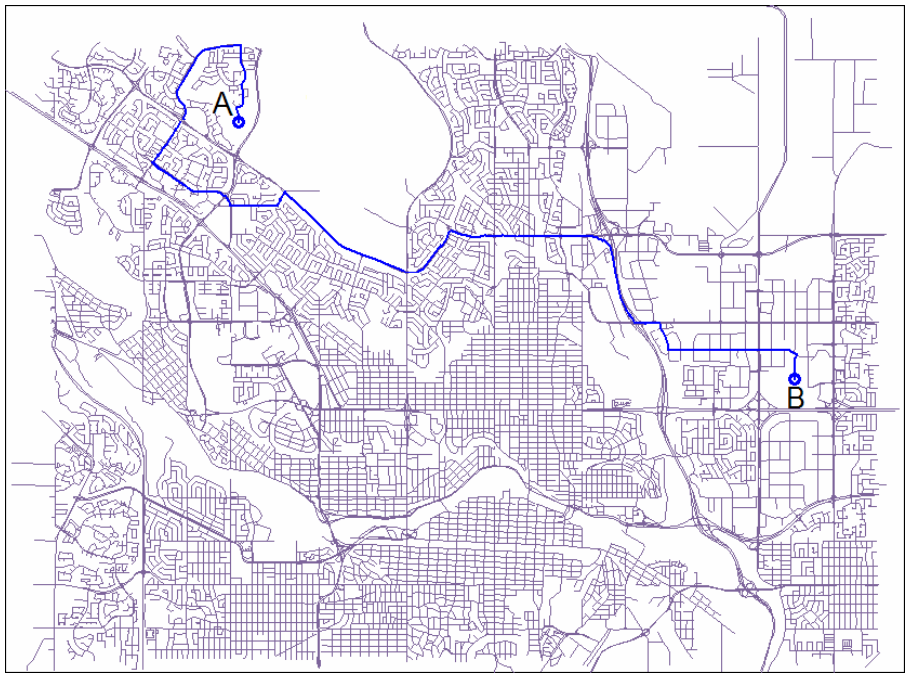


Figure 4.13: Final Optimal Route

4.4.3 Experimental Results for the Improved LPA* Algorithm

In my experiment, I compare the performance of the improved LPA* algorithm with the A* static algorithm. That is, for each path computation, the improved LPA* can partially reuse previous search results and the A* has to compute the optimal route

from scratch. To characterize the efficiency of LPA*, first we need to select different routes by their approximate length to examine to what extent LPA* is superior to A*. Next, I utilize my software to simulate a dynamic environment and make the link-weights change with different proportions, from 5% to 40%. For the improved LPA*, we test its performance with and without the assistance of the constrained ellipse respectively. To avoid biasing the experimental results in favor of any single approach, we use a binary heap to implement a priority queue for both A* and LPA*. For comparison purposes, the number of nodes expanded is taken as a benchmark to test the efficiency. The first search of LPA* is not involved because it is the same as A* if the new heuristic is not applied.

Table 4.1 illustrates the varying performance with various path cardinalities. The cardinality of a path refers to the number of nodes contained in the final shortest path. In general, the difference in cardinality stands for the difference in path length between each shortest path computation; a longer path may contain more nodes. From Table 4.1, we can discern that these paths contain between 10 and 60 nodes. In this test, only 5% of the links have been modified with a new weight value. Table 4.2 shows the experimental result from a dynamic environment where the weights of 10% of the links of the entire graph have been updated. Figure 4.14 illustrates the node expansion in a dynamic environment influenced by different proportions of weight being updated. This route contains about 40~50 nodes.

Table 4.1 Nodes Expansion in Different Routes with 5% Links Updated

Cardinality of the path / Approach	11	19	33	43	52
Original A*	34	75	156	284	497
Improved LPA* without constrained ellipse	7	20	32	73	128
Improved LPA* with constrained ellipse	5	16	25	55	98

Table 4.2 Nodes Expansion in Different Routes with 10% Links Updated

Approach \ Cardinality of the path	11	19	33	38	55
Original A*	32	78	165	280	512
Improved LPA* without constrained ellipse	9	28	37	72	134
Improved LPA* with constrained ellipse	6	19	26	59	104

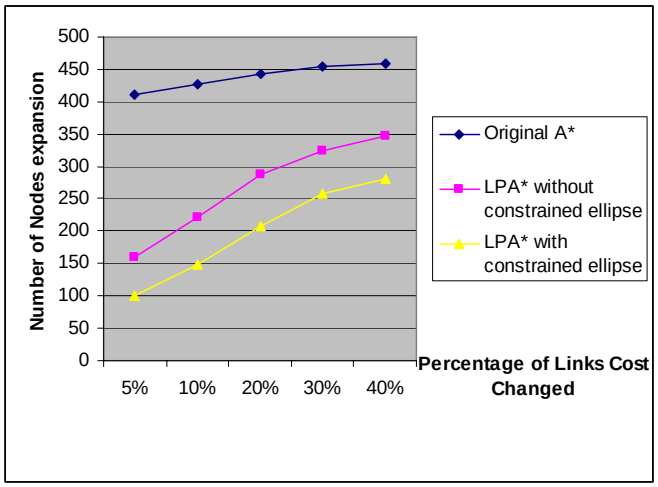


Figure 4.14: Nodes Expansion VS. Proportion of Links Updated

From the evaluations of the experiment, we can observe that the improved LPA* approach is significantly superior to A*. Table 4.1 demonstrates that with increased path length, LPA* greatly reduces computational costs as compared to A*. In some cases, the reductions are in the order of 70-80%. This means that in a navigation service area, a service provider may incur considerable computation costs on the shortest path queries for clients who start a long trip and frequently recalculate the optimal route. If the number of queries gets too large, the service may be degraded.. By reusing previous search results, the improved LPA* approach can handle vast numbers of query requests.

Table 4.1, Table 4.2 and Figure 4.14 prove that, with the assistance of a constrained ellipse, the LPA* can be more efficient for route optimization without missing any useful nodes. Although this improvement is not large, the innovation may reduce search costs by 10~20%.

Table 4.2 and Figure 4.14 show that our approach works well if the link-weights do not change significantly. If the link-weights change significantly, the performance of LPA* algorithm will approach that of the A* algorithm. This means that only a small portion of the previous search information can be reused by LPA* and it may lose its advantage. In extreme condition, it has to search from scratch like A*.

4.5 Chapter Summary

In the beginning of this chapter, the existing LPA* algorithm is described in detail. LPA* is a dynamic shortest path algorithm which combines the A* and RR algorithm to answer similar routing queries. This algorithm employs a heuristic to prune unnecessary nodes and reduce the search space oriented to the goal. In addition, the integrated RR approach provides an incremental method to dynamically adjust the shortest path. It only modifies the portion of the route that is affected by changes in the link-weights.

The drawback of LPA* is that, like most other dynamic methods, it can only perform the dynamic route recalculation between a fixed start point and goal, and cannot satisfy the requirement of en route routing queries. To overcome this problem, I propose an improvement based on LPA*, which reverses the searching direction from the goal toward the start point. In cases where the start point changes due to the mobility of the user, the previous search results can still be reused since all nodes are labeled with the goal distance, instead of the start distance. Therefore, the only thing that should be done is establish a new heuristic for all involved nodes, which is easy to calculate.

To test the performance of my algorithms, I developed experimental software for all routing algorithms. The main software functions and the class implementations are briefly described. The emphasis of this chapter is on discussing the data structure and

manipulation of the priority queue, which may strongly affect the performance of the search algorithms.

To efficiently access the priority queue, a well-known data structure, the binary heap, is employed to facilitate insertion and deletion operations in the queue. The discussion illustrates its basic principle, algorithms and superiority. By using a binary heap, the performance of all the search algorithms can be improved.

The experimental results demonstrate that, in most cases, the improved LPA* is significantly superior to A*. In the worst case, i.e., if the weight of the network significantly changes, its performance approaches that of A*. Therefore, in most cases, my algorithm can preserve the strength of the LPA* algorithm and solve the en route query. As well, the performance of the improved LPA* can be further improved with the help of a constrained ellipse.

Chapter 5: Nearest Neighbor Problem

As mentioned in section 1.4, the second type of query deals with finding the closest facility, such as the nearest hotel, hospital or gas station, without knowing the destination in advance. This is defined as the nearest neighbor query, which retrieves the data point that is closest to a query point. In this chapter, I discuss some traditional methods for solving the problem. The main limitation of these methods is that they can only be applied in a static environment, i.e., they cannot answer en route queries in a dynamic environment.

Traditionally, there are two commonly used methods to solve the nearest neighbor problem: the indexing approach and the Voronoi diagram based method.

5.1 Indexing Approach

5.1.1 Background Knowledge on the Spatial Index

Because of the large volumes of spatial data and time-consuming geometric algorithms, which need underlying systems with extended features involving query languages, data models and indexing methods, extensive research has been conducted in this context on the design of efficient index structures to accelerate access to spatial data. The purpose of the spatial index structure is to reduce the set of objects which are examined when processing a query. R-tree is the most popular spatial index structure used widely in GIS and its structure is illustrated as follows.

R-tree is a spatial access method which utilizes hierarchically nested (and possibly overlapping) boxes to separate space. The tree is height-balanced; that is, all of the leaves are at the same level. The structure handles objects by means of their conservative approximation. The simplest approximation of an object's shape is the

Minimum Bounding Rectangle (MBR). Each node of the tree corresponds to exactly one disk page. The point is that only the MBRs are stored, not the objects themselves. Internal nodes contain entries of the form (R, child-ptr), where R is the MBR that encloses all of the MBRs of its descendants and child-ptr is the pointer to the specific child node. Leaf nodes contain entries of the form (R, object-ptr) where R is the MBR of the object and object-ptr is the pointer to the object's detailed description. Figure 5.1 is an example of an R-tree.

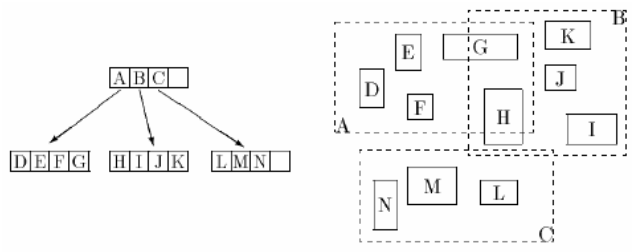


Figure 5.1: An example R-tree

5.1.2 Nearest Neighbor Search Using Indexing Approach

Roussopoulos et al. [36] propose a branch-and-bound algorithm that searches the R-tree in a depth-first manner. The basic idea of this algorithm is that the Euclidian distance between the query point and any node is always less than the network distance. The search starts from the root where all entries are sorted according to their minimum Euclidian distance (*mindist*) from the query point. The entry with the smallest value is visited first. The process is repeated recursively until the algorithm reaches the leaf level where the first potential nearest neighbor is found. It then employs a one-to-one shortest path algorithm to compute the shortest path to the most promising candidate in terms of network distance and stores this distance as the shortest path distance found so far. The algorithm then backtracks to the upper levels of the tree only visiting those entries whose *mindist* is smaller than the shortest path distance to the nearest neighbor already found. The search terminates once the known shortest path is less than the *mindist* of the next entry. In the worst case, the algorithm will search all objects in a certain category.

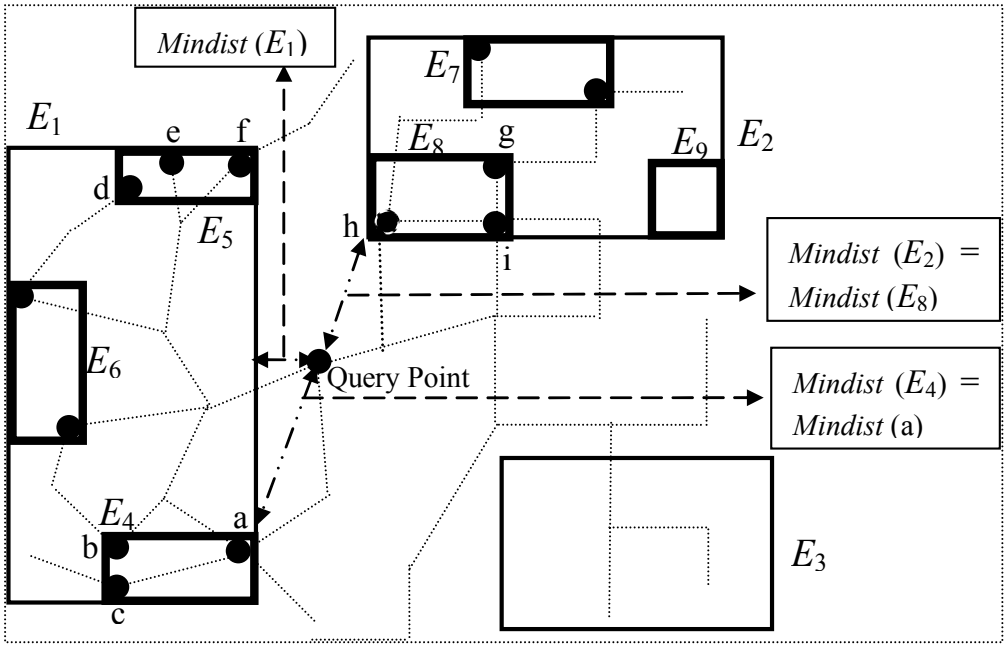


Figure 5.2: Indexing Nearest Neighbor

In the example shown in Figure 5.2, the algorithm first visits the root entry E_1 since it has the smallest *mindist*, and then E_4, where the first candidate node, *a*, is retrieved. When backtracking to the previous level, entries E_5 and E_6 are excluded, since their *mindist* is greater than the path distance to *a*. Then E_2 and E_8 are accessed, where the actual nearest neighbor (node *h*) is found. Samet and Hjaltason [37] develop an improved nearest neighbour algorithm. To determine what node should be examined next, it selects the node with the smallest distance in the set of nodes, which have been visited. This means that the algorithm uses a priority queue to track the nodes to be visited, instead of using a stack or a plain queue. The distance from the query object to each node is used as a key. Although the improved algorithm has performs better than Roussopoulos' algorithm, the search principle is similar. Both of them follow the filter and refine strategy. These two steps are time consuming since many unnecessary shortest path computations are involved before the actual nearest neighbor is found.

Consider an alternative situation where a user with a location-aware mobile device submits a continuous query with respect to his/her current position (e.g., the user

wants to know the closest restaurant while traveling). Due to the mobility of the user, the result may be immediately invalidated as the user's position changes. The conventional approach to attain current information is to submit a new query to the server after a position update, which could lead to high network overhead and extra processing effort.

To solve this problem, Zhang et al [38] suggest that, if a client remains in an area around the initial position, called the *validity region*, the result remains the same. In addition to the query result, the server has to return the validity region of the query. The clients use the validity region to determine whether a new query should be issued by verifying whether their current position is still inside the validity region. To derive the validity region, a Voronoi diagram is used to partition the data space.

5.2 Voronoi Diagram Approach

5.2.1 Fundamental Knowledge of Voronoi Diagram

The *Voronoi diagram* is a well known data structure extensively investigated in the domain of computational geometry [39]. Originally, it characterized regions of proximity for a set of k sites in a 2-D plane where the distance between points is defined by their Euclidean distance. In the following sections, we review the principles of the Voronoi diagrams, starting with the Voronoi diagram for 2-D Euclidean space. We then discuss the *network Voronoi diagram* where the distance between two objects in space is their shortest path in the network rather than their Euclidean distance. These diagrams can be used for spatial networks. Okabe et al. present a thorough discussion on regular and network Voronoi diagrams [35].

5.2.1.1 Definition

Consider a set containing a limited number of points, called *Voronoi sites*, in the Euclidean plane. We associate all locations in the plane to their closest Voronoi sites.

The set of locations assigned to each Voronoi site form a region called the *Voronoi region* or *Voronoi cell*, of that Voronoi site. The set of Voronoi regions associated with all the Voronoi sites is called the Voronoi diagram with respect to the Voronoi sites set. The Voronoi regions of a Voronoi diagram are collectively exhaustive because every location in the plane is associated with at least one Voronoi site. The regions are mutually exclusive, although they share boundaries. The boundaries of the regions, called *Voronoi edges*, are the set of locations that can be assigned to more than one Voronoi site. The Voronoi regions that share the same edges are called *adjacent regions* and their Voronoi sites are called *adjacent Voronoi sites*. The Voronoi region and Voronoi diagram can be formally defined by the following: Assume a set of Voronoi sites $P = \{p_1,...,p_n\} \subset \Re^2$, where $2 < n < \infty$ and $p_i \neq p_j$ for $i \neq j$; $i, j \in I_n = \{1,\ldots,n\}$. The region given by:

$$VP(p_i) = \{p \mid d(p, p_i) \leq d(p, p_j)\} \text{ for } i \neq j;\ i, j \in I_n, \qquad \textbf{(5.1)}$$

where $d(p, p_i)$ specifies the minimum distance between p and p_i (e.g., length of the straight line connecting p and p_i in Euclidean space), is called the *Voronoi Region* associated with p_i. The set given by:

$$VD(P) = \{VP(p_1),...,VP(p_n)\}, \qquad \textbf{(5.2)}$$

is called the *Voronoi Diagram* generated by P. Figure 5.3 shows an example of a Voronoi diagram, its regions and Voronoi sites.

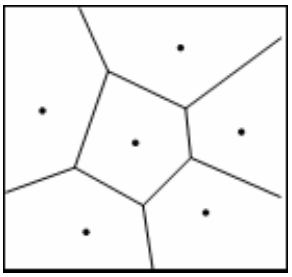


Figure 5.3: An Example of a Voronoi diagram

The nearest neighbor search method proposed by Zhang et al [38], and many other researchers, involves pre-computing Voronoi diagram to construct Voronoi cells for each point of interest. Each Voronoi region is the validity region of the corresponding

point of interest. Thus, if the query point remains in this region, its nearest neighbor remains the same. The main problem of this method is that the Voronoi diagram is constructed based on Euclidean distance. It cannot be applied to road networks in that the shortest network distance (e.g., shortest path, shortest time) between objects (e.g., the vehicle and the restaurant) depends on the connectivity of the network rather than the objects' locations.

.

5.2.1.2 Network Voronoi Diagram

A network Voronoi diagram is a specialization of Voronoi diagrams where the location of objects is restricted to the links that connect the nodes of the network. The distance between objects is defined as the length of the shortest link distance (e.g., shortest path or shortest time) instead of the Euclidean distance.

For a network Voronoi diagram, any node located in a Voronoi region has a shortest path to its corresponding Voronoi site that is always less than that to any other Voronoi site. In this way, the entire graph is partitioned into several subdivisions as shown in Figure 5.4, where *p1*, *p2* and *p3* are the Voronoi sites. We can assume that the set of Voronoi sites is the set of facilities (e.g., hotels, restaurants, etc.) and *p4* to *p16* are the road network intersections that are connected to each other by the set of streets, *L*.

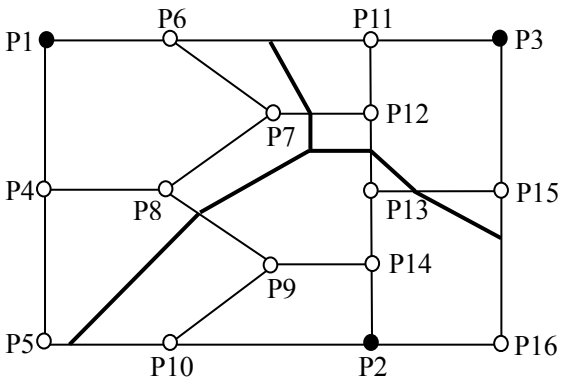


Figure 5.4: A Network Voronoi diagram

For the network Voronoi diagram depicted in Figure 5.4, the network Voronoi edges

usually intersect with the links in most cases. This means that a link may be divided into two parts and placed into two adjacent Voronoi regions. Although this partitioning is very precise, it is not necessary in vehicle navigation services since vehicles are constrained by the road network, i.e., they cannot change direction until they reach certain points such as U-turn points or intersections. Therefore, we are not concerned with which Voronoi region a link belongs to, we just need to know the Voronoi regions for the two nodes connected by the link. In real-world applications, any location that lies on a road can be referred to by the location of the intersection ahead for vehicles, or the nearest intersection for pedestrians. Hence, all locations in a network are further restricted to the nodes and the construction of a network Voronoi diagram is simplified as partitioning only the nodes, instead of the entire graph. For a directed graph, there are two types of network Voronoi diagrams: inward and outward Voronoi diagrams. This means that the Voronoi diagram is based on the shortest paths which lead toward the Voronoi sites (inward) or come from the Voronoi sites (outward). In this thesis, we concentrate on the inward Voronoi diagram for accessibility analysis since we are only concerned with the movement toward a facility.

5.2.2 The Network Voronoi Diagram Construction -- Parallel Dijkstra's Algorithm

As described in [40], a variation of Dijkstra's algorithm, Parallel Dijkstra's algorithm, was proposed to construct the network Voronoi diagram. The search starts from the nodes which are closest to the Voronoi sites. These start nodes are initialized with their Voronoi region label. Like Dijkstra's algorithm, all successors of these start nodes are inserted into the priority queue, labeled by their start distance and Voronoi region. The difference between the two algorithms is that the distance label for the Parallel Dijkstra's algorithm is the start distance with respect to the different start nodes. Similarly, for node *u,* if the distance label of *u* plus the cost of the out-edge *(u,*

v) is less than the distance label for *v,* then the estimated distance for node *v* is updated with this new value, even if the nodes *u* and *v* are expanded from different start nodes. Thus there is no need to keep the original start node of *v.* The predecessor and Voronoi region of *v* is updated in accordance with *u*. Since no destinations are specified for these start nodes, the search terminates when the priority queue is empty. Consequently, Parallel Dijkstra's algorithm derives a collection of one-to-some shortest path trees for each node group respectively. This algorithm is called Parallel Dijkstra's algorithm because the shortest path trees starting from each seed (i.e., Voronoi sites) grow simultaneously. Figure 5.5 demonstrates the growth of a network Voronoi diagram constructed by Parallel Dijkstra's algorithm.

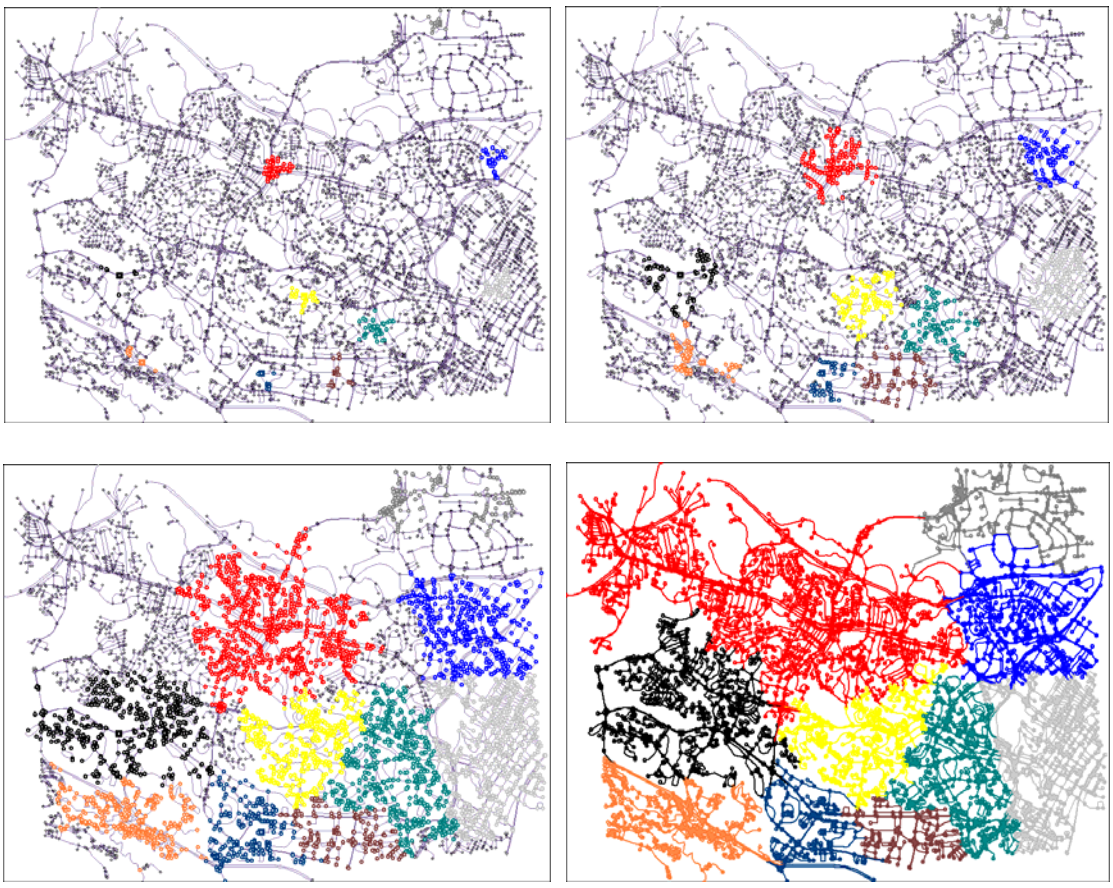


Figure 5.5: Parallel Dijkstra's Algorithm

By using the network Voronoi diagram, the road network is partitioned into several contiguous subdivisions without overlapping or disjunction. Each subdivision can be

regarded as a service area of its respective Voronoi site (e.g., hotels, hospitals, gas stations etc.). In addition, the algorithm builds a shortest path tree for each Voronoi site in the Voronoi diagram from which the shortest path from any node located in a service area to its Voronoi site is derived. In this way, it is easy to determine the nearest neighbor and associated path for any position in the transportation network.

5.3 Chapter Summary

In this chapter, a possible solution for the second query type is discussed. To find the nearest facility, indexing technique and Voronoi diagram are the most popular approaches. Indexing method use an index to filter the potential candidates and compute the shortest path to each candidate online. In a dynamic transportation environment where traffic conditions change over time, this approach has to search for the nearest neighbor and constantly recalculate the optimal route. Therefore, it is inefficient and may result in long latency time, especially for large number of clients.

Voronoi based methods solve the nearest neighbor problem by constructing a Voronoi region for each facility. This Voronoi region works as a validity region or service area for the mobile client. As long as the client remains within the validity region, the nearest neighbor to the client is always the same. Because the nearest neighbor of the clients located in a Voronoi region is the same, the Voronoi diagram has the potential to provide batch services to many clients. However, most previous research has focused on constructing Voronoi diagram based on Euclidian distance in a 2-D plane, which is not feasible for transportation networks and is not capable of answering closest facility queries. Even though some researchers adopted network distance to construct Voronoi diagrams, see [40] and [41], their approaches can only be used in a static environment. Therefore, there is a need to dynamically maintain the network Voronoi diagram.

Chapter 6: Dynamic Routing Algorithm for Unknown Destination

To answer the en route query about the closest facility, both the best destination and an associated optimal route need to be searched, based on network distance. If traffic conditions change, the optimal route also needs to be adjusted to adapt to the dynamic changes. Furthermore, the query result of the closest facility may vary over time due to the mobility of the user and the change in traffic conditions. On the other hand, the query result may vary over time even if the queries are submitted from the same position.

Since the facilities used in my research are denoted by the Voronoi sites, the closest facility is easily identified for any location by using a dynamic network Voronoi diagram. To maintain the dynamic network Voronoi diagram and derive adaptive shortest paths from the user's current location to the nearest facility, I combine the Parallel Dijkstra's algorithm and RR approach as a novel algorithm, namely the Incremental Parallel Dijkstra's algorithm (IP-Dijkstra for short). The proposed new algorithm is my contribution from this research.

6.1 IP-Dijkstra's Algorithm Overview

Similar to Dijkstra's algorithm, IP-Dijkstra always expands the node in the priority queue with the smallest key value, which is defined as: For node u,

$$k(u) = Min(g(u), rhs(u)) \quad \textbf{(6.1)}$$

The priority of a node in the priority queue is always the same as its key. We use the heap-based implementation of a priority queue for IP-Dijkstra, i.e., we have available the operations *insert (u; h)*, which inserts the node u into the heap h by its key value *k(u)*, and *pop(h)*, which removes the minimal element from h and return the element.

In the first search, the behavior of IP-Dijkstra is almost the same as the Parallel Dijkstra's algorithm. The only difference is that IP-Dijkstra labels the *rhs*-value for each node. At the beginning, IP-Dijkstra inserts the start nodes into the priority queue, which are the closest nodes from the Voronoi sites. Because their key values are all zero, the order is not important. Once a node is popped up from the priority queue, it is marked and deleted from the priority queue. Its successors are initialized as infinity for both the *g*-value and *rhs*-value, and then inserted into the priority queue. The search terminates once the priority queue is empty. Finally, all nodes of the graph have been traversed and assigned to a Voronoi site. Note that we desire an inward Voronoi diagram to ensure that the final shortest path leads from any node toward its Voronoi site with the correct cost value to evaluate the accessibility. However, the search is expanded from the Voronoi sites to the outer nodes; therefore we must adopt the link-weight of the opposite direction for node expansion in the directed graph. Then, for each node, one can trace back a shortest path to the corresponding Voronoi site by starting at the node and always decreasing the start distance to its predecessor. Thus, the shortest path based network Voronoi diagram is constructed.

The first search of IP-Dijkstra is illustrated in Figure 6.1. For simplicity, I assume there are only two Voronoi sites, A and H, symbolized by a blue and a red circle respectively. The left-upper graph gives the weight for each link. In the following iterations, there is a bracket around each node which encloses two values denoting the key value and the start distance (*g*-value) respectively. These are given a color in accordance with their corresponding Voronoi sites. A single value in a bracket denotes the *g*-value of the nodes which are locally consistent. The black square indicates the node that is expanded in the current iteration.

Observing Figure 6.1, IP-Dijkstra starts from node A and H. In iteration #3, node B holds the minimum key and is expanded first. Node E is then given a start distance of 6. In the next iteration, the start distance of node E is updated as 5 after node C is

expanded. The shortest path between A and E now changes to {E, C, A}, instead of {E, B, A}. In iteration #5, the expansion of node G causes the start distance of E to be updated. The new start distance is further smaller than before, which has the consequence of changing its corresponding Voronoi site to H. As a result, the new shortest path is {H, G, E}. This update also occurs for node D in iteration #6. As a result, the nodes, like D and E, help form the boundary of the Voronoi region.

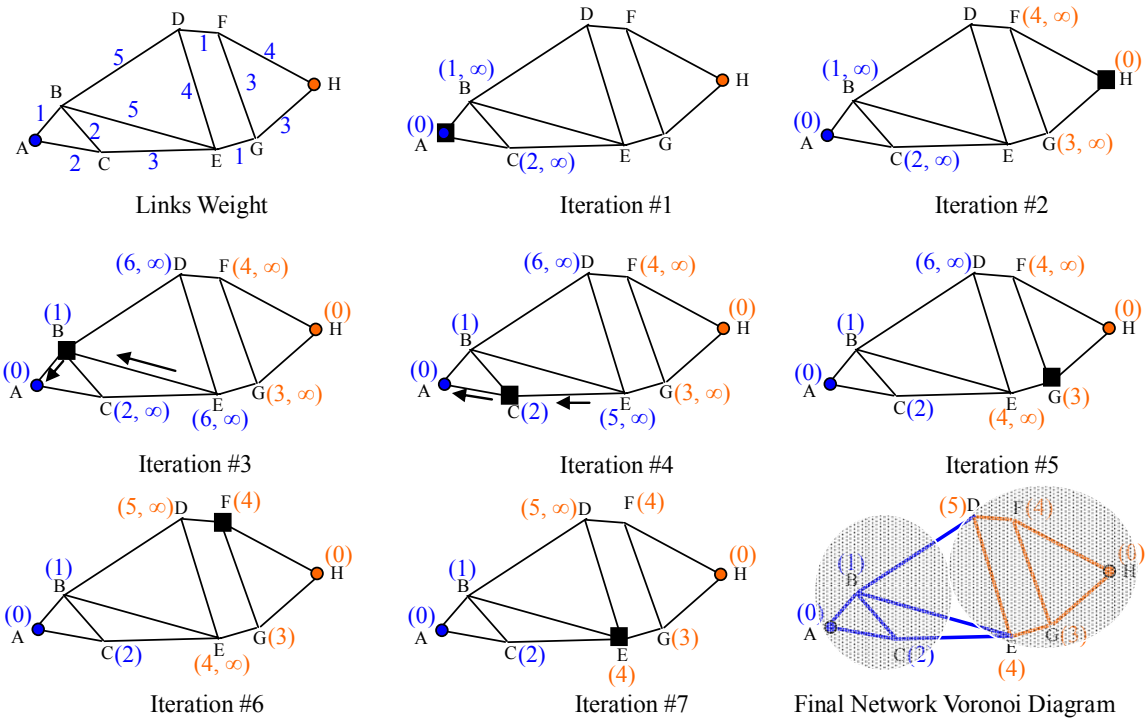


Figure 6.1 First Search of IP- Dijkstra

Since the network Voronoi diagram was constructed in the first search, it is easy to identify which Voronoi site is the desired destination (i.e., the nearest facility) for any location. We assume this Voronoi diagram is valid in time *T1* and that a mobile client located in node D submits a nearest neighbor query. Because H is the corresponding Voronoi site for node D, the mobile client is navigated to H along the route {D, F, H}. However, in a dynamic environment, the weight of any link may change arbitrarily. For example, when the mobile client arrives at node F along the designated route heading to H, the client is provided with current traffic conditions for time *T2*. The new information indicates that there is a traffic jam in the link FH. As can be seen in Figure 6.2, the traffic jam is represented as an increase of 3 in the weight of link FH.

For this time interval, the original network Voronoi diagram and the optimal path may no longer be valid and may need to be modified.

To adapt to this change, we first check the estimates (*g, rhs*) of the nodes around the Link FH, which have the most potential to be affected by this change. Here, nodes F and H are taken into account. In fact, node H is not affected by this change, but the tentative start distance of node F does change (g(F) =7). Node F now becomes locally inconsistent due to its *g*-value not being equal to its *rhs*-value. In iteration #1, the *rhs*-value of node F is updated to 6 by searching its neighbors while its *g*-value is assigned as infinity. In iteration #6, we expand node F and make it locally consistent again. So far, the route to H has been modified from {F, H} to {F, G, H}. In this sense, IP-Dijkstra is capable of continually providing en route navigation service during travel in that it is able to deliver a new optimal route to mobile clients based on their current location. In extreme circumstances, the new route may lead to a different Voronoi site.

In iteration #3, node D is expanded and its start distance changes to 6, which comes from node B. This causes the Voronoi site for node D to be changed to A. Due to this change, if any other mobile clients submit queries from node D, they will be navigated to A instead of H. This illustrates the main advantage of the IP-Dijkstra algorithm. It can dynamically derive the accessibility region for each Voronoi site.

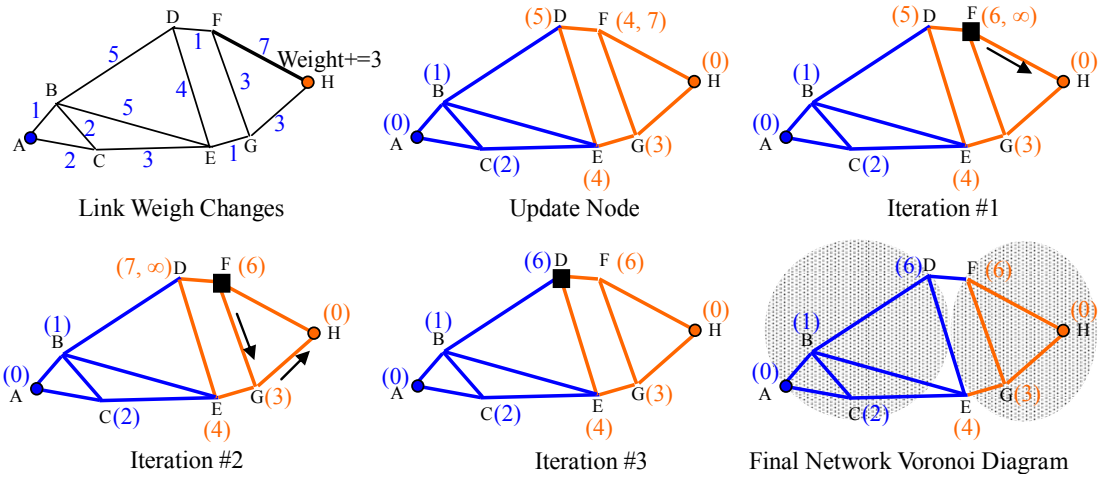


Figure 6.2 Second Search of IP- Dijkstra

6.2 Details of the IP-Dijkstra's Algorithm

In previous sections, I have given some details on how the IP-Dijkstra algorithm works. Now I will give some details about the algorithm pseudo-code. In the code given below, *Vor* denotes the node sets that are closest to the Voronoi sites and are used as start nodes, while *V(u)* denotes the respective Voronoi site of u. The following functions are employed to manage the priority queue.

U.TopKey() - returns the smallest priority of all nodes in the priority queue U. (If U is empty, then U.TopKey() returns $[\infty;\infty]$.)

U.Pop() - deletes the node with the smallest priority in the priority queue U and returns the node.

U.Insert(s, k) - inserts node s into the priority queue U with priority k.

U.Remove(s) - removes node s from the priority queue U.

Procedure CalculateKey(s)

(01) return min($g(s)$, $rhs(s)$);

Procedure Initialize()

(02) $U = \varnothing$;

(03) for all $s \in S$ $rhs(s) = g(s) = \infty$;

(04) for all $v \in Vor$

(05) $rhs(v) = 0$;

(06) *V(u)= NULL;*

(07) U.Insert(v,0);

Procedure UpdateNode(u)

(08) if ($u \notin Vor$) $rhs(u) = \mathrm{Min}_{s' \in pred(u)}(g(s') + w(s', u))$;

(09) Update*(V(u))*;

(10) if ($u \in U$) U.Remove(u);

(11) if ($g(u) \neq rhs(u)$) U.Insert(u, CalculateKey(u));

Procedure ComputeVoronoi()

(12) while (U.TopKey()$\neq\infty$) // while U is not empty

(13) u = U.Pop();

(14) if ($g(u) > rhs(u)$)

(15) $g(u) = rhs(u)$;

(16) for all $s \in succ(u)$ UpdateNode(s);

(17) Else

(18) $g(u) = \infty$;

(19) for all $s \in succ(u) \cup \{u\}$ UpdateNode(s);

Procedure Main()

(20) Initialize();

(21) while(true)

(22) ComputeVoronoi();

(23) Detect the weight change in graph

(24) If any change occurs

(25) for all links (u, v) with changed link weights

(26) Update the link cost $W(u, v)$;

(27) UpdateNode(v);

In this pseudo-code, the main function, Main(), first calls Initialize() {line 20} to set the initial g-values of all nodes to infinity and their rhs-values according to Equation 3.4.1 {lines 03-05}. Thus, the initial start nodes are locally inconsistent and inserted into the empty priority queue with a key {line 07}. Next, ComputeVoronoi() is called {line 22}, which expands the nodes in the well-known manner until the heap is empty. When updating a node's tentative start distance, $g(v)$, where the update was caused by

link (*u,v*), we additionally set *V(v) = V(u)* {line 09}. The IP-Dijkstra has now constructed the network Voronoi diagram and derived the shortest path trees for all nodes. After this, the algorithm waits for changes in link-weights. If some link-weights have changed, the pseudo-code calls UpdateNode() {line 27} to update the *rhs*-values and keys of the nodes that are potentially affected by the changed link-weights as well as their membership in the priority queue if they become locally consistent or inconsistent. It also updates the network Voronoi diagram, as well as any associated shortest path trees, by calling ComputeVoronoi() {line 22}. This procedure repeatedly expands locally inconsistent nodes in the order of their priorities.

A locally inconsistent node *s* is called locally overconsistent if, and only if, $g(s) > rhs(s)$. When ComputeVoronoi() expands a locally overconsistent node, it sets the *g*-value of the node to its *rhs*-value, which makes the node locally consistent {line 15}. A locally inconsistent node *s* is called locally underconsistent if, and only if, $g(s) < rhs(s)$. When ComputeVoronoi() expands a locally underconsistent node, it sets the *g*-value of the node to infinity {line 18}. This makes the node either locally consistent or overconsistent. If the expanded node was locally overconsistent, the change of its *g*-value can affect the local consistency of its successors {line 16}. Similarly, if the expanded node was locally underconsistent, it and its successors can be affected {line 19}.

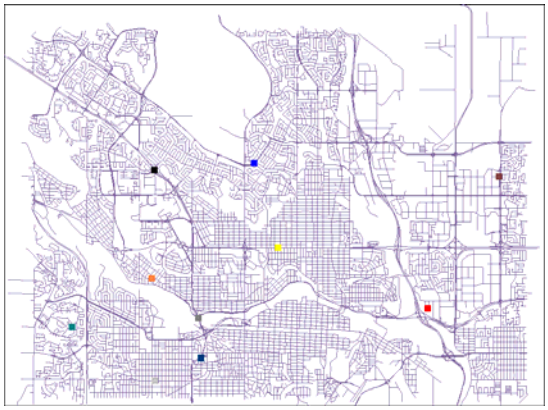
The above discussion verifies that the IP-Dijkstra algorithm is able to efficiently manage the nearest neighbor queries, especially for large number of mobile clients. Its superiority lies in the fact that all nodes of the graph are dynamically labeled with their corresponding Voronoi sites and need not involve another geo-computation to determine their Voronoi regions. Whenever a query for the nearest facility is submitted, it can be answered immediately based on the current network Voronoi diagram. Another prominence of IP-Dijkstra is that the network Voronoi diagram and associated shortest path trees are constructed together without additional computational overhead.

6.3 Experimental Studies

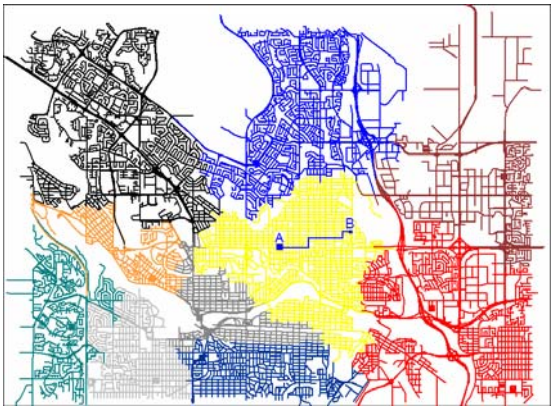
6.3.1 En Route Query Demonstration for Unknown Destination

In this type of query, the mobile client is trying to find the optimal route to the closest facility without knowing the destination in advance, e.g., the closest shopping mall. The 10 colored squares depicted in Figure 6.3 (a) and Figure 6.4 (a) stand for the shopping malls.

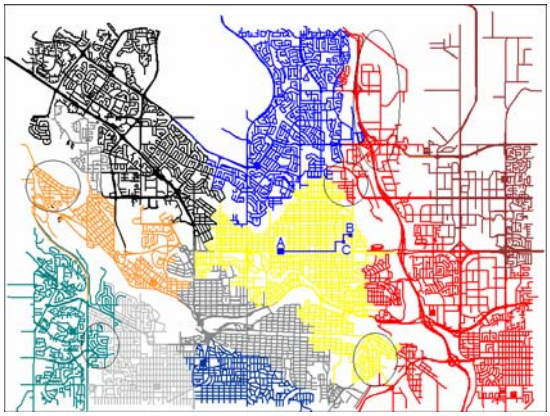
We assume that the initial network Voronoi diagram for these two road maps are constructed in time T1 as shown in Figure 6.3 (b) and Figure 6.4 (b), and the areas with different colors stand for the respective service area for each mall. With these service areas, one can evaluate the accessibility of the malls and find the closest mall as well as the optimal route in real time. To examine the validity of our algorithm, the simulated traffic conditions are updated in time T2 and T3. Based on the changing traffic condition, IP-Dijkstra not only modifies the partition of the service areas, but also adjusts the shortest path trees within each service area. In Figure 6.3 (c, d) and Figure 6.4 (c, d), there are several circles that are used to identify where the service areas have been modified compared with previous time slices. In addition, the shortest path to the nearest mall has also been updated for each location. Observing the yellow area in Figure 6.3 and the light brown area in Figure 6.4, assume the mobile client has submitted a query at location B in time T1 and learned that the nearest mall is A. The initial optimal route was derived as shown in Figure 6.3 (b) and Figure 6.4 (b). In time T2, the mobile client arrived at location C and had to recalculate the optimal route due to changing traffic conditions. The new optimal route is depicted in Figure 6.3 (c) and Figure 6.4 (c). Similarly, the traffic condition changed again in time T3 when the mobile client arrived at location D. Figure 6.3 (d) and Figure 6.4 (d) show the final optimal path. Therefore, queries submitted at different time may give different optimal solutions, including the best destination and the optimal route.



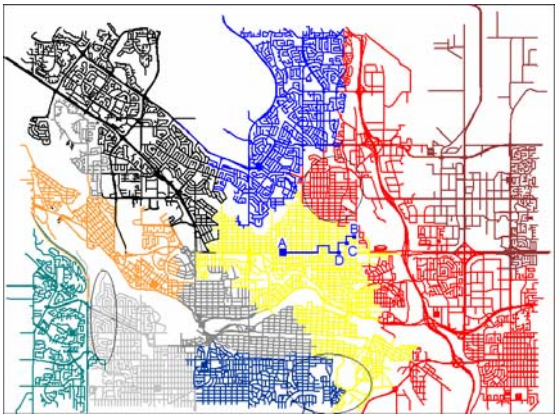
(a) Facilities Distribution



(b) Service Area Partition in T1

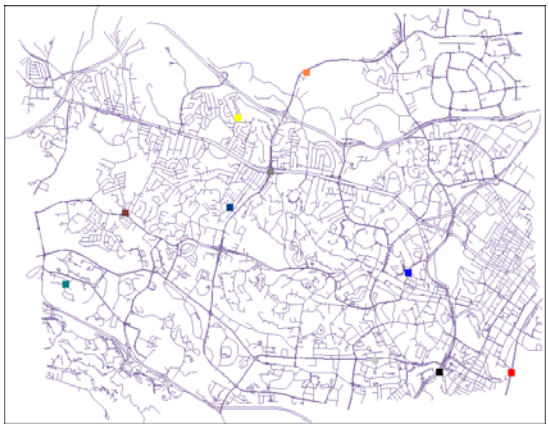


(c) Service Area Partition in T2

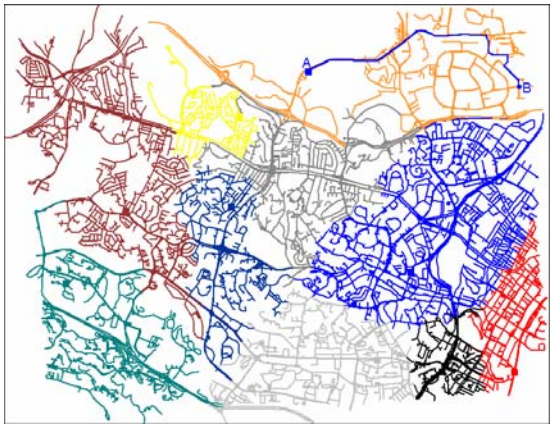


(d) Service Area Partition in T3

Figure 6.3: Closest Facility Query in Calgary



(a) Facilities Distribution



(b) Service Area Partition in T1

(c) Service Area Partition in T2 (d) Service Area Partition in T3

Figure 6.4: Closest Facility Query in Singapore

6.3.2 Experimental Results of IP-Dijkstra's Algorithm

In my experiment, we first compare the performance of my new IP-Dijkstra's algorithm with the Parallel Dijkstra's algorithm. That is, for each closest facility query and path search, IP-Dijkstra can partially reuse previous search results, and Parallel Dijkstra has to re-construct the network Voronoi diagram and derive the optimal route from scratch. I also use my software to simulate a dynamic environment and make the link-weights change with different proportions from 1% to 20%. In addition, I perform 50 time trials with each proportion and get an average performance, which is listed in Table 6.1 and Table 6.2. For comparison purposes, the number of nodes expanded is taken as a benchmark to test the efficiency. The first search of IP-Dijkstra is not involved because it is the same as Parallel Dijkstra.

Since Parallel Dijkstra is a static method, it has to construct the network Voronoi diagram from scratch by expanding all nodes whenever the traffic condition changes. Therefore, we only need to count the nodes expanded by IP-Dijkstra in each update. Table 6.1 and Table 6.2 give the experimental results regarding the number of nodes expanded by IP-Dijkstra in the Calgary and Singapore datasets respectively.

Table 6.1: Nodes Expansion for Dynamic Update in Calgary Road Network

Percentage / Dataset		Links Change with Different Percentage				
		1%	3%	7%	12%	20%
Facilities	Qty	Number of Nodes Expansion (Percentage of Nodes Expansion)				
Shopping Malls	10	534 (6.4%)	1026 (12.3%)	1867 (22.4%)	2793 (33.5%)	4135 (49.6%)
Hotels	36	428 (5.1%)	837 (10.0%)	1454 (17.4%)	2287 (27.4%)	3192 (38.3%)
Restaurants	84	271 (3.2%)	539 (6.5%)	851 (10.2%)	1288 (15.4%)	2236 (26.8%)
Gas Stations	162	172 (2.1%)	365 (4.4%)	533 (6.4%)	815 (9.8%)	1314 (15.8%)

Table 6.2: Nodes Expansion for Dynamic Update in Singapore Road Network

Percentage / Dataset		Links Change with Different Percentage				
		1%	3%	7%	12%	20%
Facilities	Qty	Number of Nodes Expansion (Percentage of Nodes Expansion)				
Shopping Malls	10	651 (9.4%)	1210 (17.5%)	2298 (33.2%)	3305 (47.8%)	5123 (74.1%)
Hotels	33	538 (7.8%)	1066 (15.4%)	1873 (27.1%)	2891 (41.8%)	4002 (57.9%)
Restaurants	79	316 (4.6%)	659 (9.5%)	1031 (14.9%)	1687 (24.4%)	2568 (37.1%)
Gas Stations	158	202 (2.9%)	435 (6.3%)	643 (9.3%)	1134 (16.4%)	1622 (23.5%)

As is evident from Table 6.1 and Table 6.2, we observe that IP-Dijkstra significantly outperforms Parallel Dijkstra in node expansion as it visits fewer nodes than Parallel Dijkstra, which has to search 100% of the nodes in each update. The experimental result illustrates that the search performance can be improved by a factor of up to 50. I can then state that IP-Dijkstra is able to efficiently adapt to changes in the link-weights. Table 6.1 and Table 6.2 also show a disadvantage of IP-Dijkstra, that is, it works perfectly only if the link-weights do not change significantly. If a large

proportion of the link-weights have been changed, its performance will drastically degrade. It means, in this case, that little previous information can be reused by IP-Dijkstra, and our approach may lose its advantage. In extreme circumstances, the algorithm may have to search from scratch similar to Parallel Dijkstra. The motivation for developing a dynamic routing algorithm is to be able to handle the situation where traffic conditions update frequently, but the change is not significant for each update, which is the usual case for real-world scenarios. This precondition gives the basis for most incremental approaches used to answer routing queries in real-time by reusing previous search results. IP-Dijkstra is suitable for solving the dynamic routing problem.

On the other hand, comparing the performance of IP-Dijkstra with different facility datasets, we observe that IP-Dijkstra is more efficient for large datasets, such as the gas station dataset in Table 6.1 and Table 6.2. The reason lies in the fact that a dataset with a large number of facilities will also have a large number of service areas (Voronoi cells). Thus, each service area will cover less area than those of a smaller dataset and fewer nodes will be contained by each service area. In other words, the average route length for all nodes leading to a corresponding facility will be fairly short. As a result, if a node is affected by the change in link-weights and has been updated, there are no too many successors of the node will be influenced accordingly. In real life, the number of facilities of most categories tends to be large. Hence, IP-Dijkstra is suitable for real-world applications.

Comparing the performance of IP-Dijkstra in the two maps, it is obvious that IP-Dijkstra is more efficient in the Calgary map, as demonstrated in Figure 6.5, which utilizes the gas station dataset to compare the performance in the different road networks. This difference comes from the characteristics of the two networks in terms of connectivity and topological structure. Let N and L denote the number of nodes and links respectively. We can calculate L / N as a ratio for the two maps to identify their link density. Since the number of links of the Calgary map is less than that of the

Singapore map, with respect to the number of nodes, the road network of Calgary is relatively simpler than that of Singapore, with low connectivity. It shows that the performance of IP-Dijkstra can be affected by the density of links and their connectivity. Nonetheless, Figure 6.5 shows that the performance difference is minor between different road networks, especially for the cases where link-weights do not significantly change. We can conclude that the incremental approach utilized in IP-Dijkstra is universally effective.

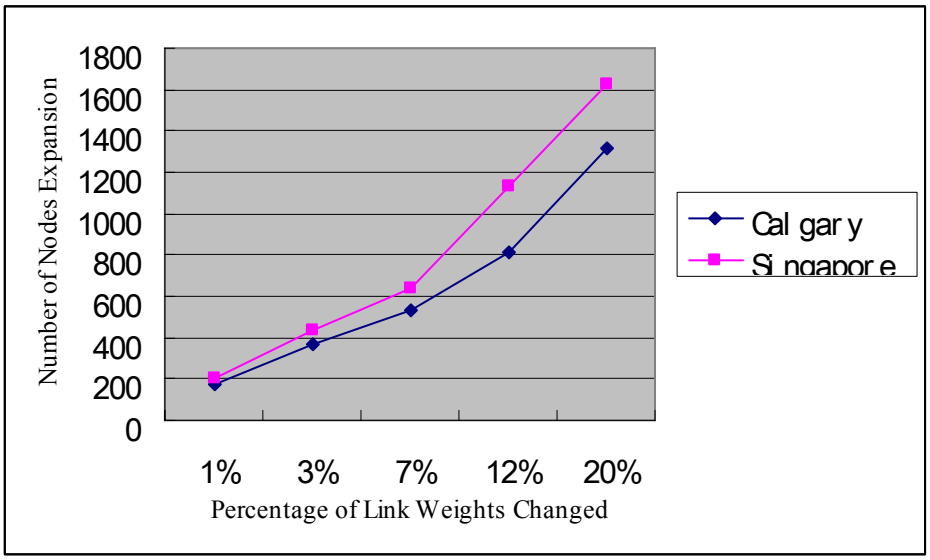


Figure 6.5: Nodes Expansion Comparison in two road maps

Next, we compare the performance of IP-Dijkstra with its competitor, which employs an indexing approach to perform a nearest neighbor search. In general, the indexing method involves two steps. First, it filters a small subset of a possibly large number of objects as the candidates for the closest neighbors of a query point based on Euclidian distances. Secondly, it requires a refinement step to compute the actual network distances between a query point and the candidates to find the actual nearest neighbor. The main disadvantage of the indexing approach is that it does not offer any solution for how to efficiently compute the distances between a query point and the candidates. It has to borrow algorithms from graph theory. For distance computations, the indexing method either pre-computes the shortest path trees for all node pairs, or

computes the shortest path online. Obviously, the first method will result in very large storage overhead, although it can notably speed up the searching. The second method involves several shortest path computations for all candidates and, thus, is not efficient. We implement the index approach using R-tree for the filter step and the pre-computation method for the refinement step in our experimental studies. In this experiment, we perform 1000 queries using each facility dataset in the Calgary map with a randomly selected query point, and then the sum of the running time is used as a benchmark. Figure 6.6 gives the result of the performance comparison.

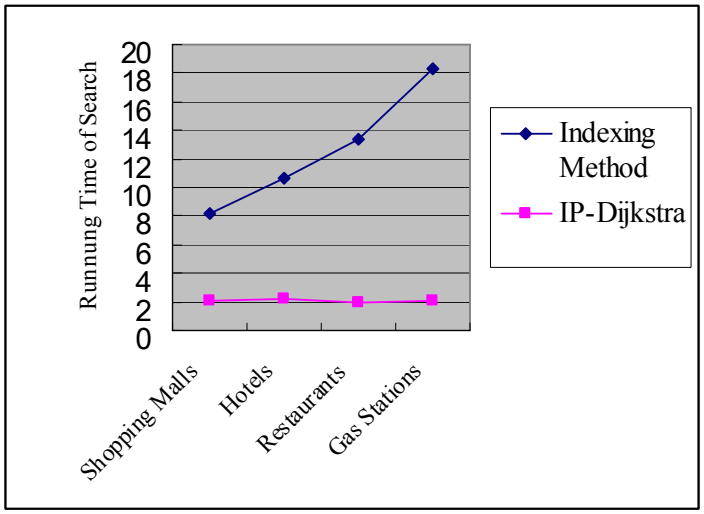


Figure 6.6: Running time for each dataset

From Figure 6.6, we learn that IP-Dijkstra does not need to visit any nodes to answer queries, as all nodes have been labeled with their respective nearest facility once the network Voronoi diagram is constructed or modified. To answer queries, one only has to allocate the query point to its closest node in the road network, and the query can then be answered immediately with information about the closest facility and optimal route, which is obtained from that node. Therefore, its performance will not be affected by different datasets when used to answer queries. In contrast, the index approach has to visit many nodes due to the filter and refinement steps. Consequently, it takes a longer time than IP-Dijkstra. Considering that the shortest path trees for all node pairs have been pre-computed and they cannot be recomputed for each

link-weight update, this approach can be only used in a static environment. Even though the indexing approach computes the shortest path online in a dynamic environment, it will perform poorly when it searches the shortest paths to different targets in the refinement step, especially in the case where facilities are not densely distributed in the network. It is then required to retrieve a large portion of the network for distance computation. Alternatively, if only a few mobile clients submit queries, this approach may be a better solution than IP-Dijkstra, as IP-Dijkstra involves complex computations in its first search. Unfortunately, the computational cost of this index approach will increase in proportion to the increase in the number of clients and will result in long latency time. The efficiency of IP-Dijkstra has been discussed above and its performance is not significantly affected by an increase in the number of mobile clients. In addition, IP-Dijkstra has the potential to provide batch service for groups of clients located in the same service area. The indexing approach cannot compete with IP-Dijkstra in accommodating large number of mobile clients in a dynamic environment.

6.4 Chapter Summary

In this chapter, I propose a novel routing algorithm, namely IP-Dijkstra's algorithm, which integrates Parallel Dijkstra's algorithm and the RR algorithm to construct a network Voronoi diagram and maintain it dynamically. In the network Voronoi diagram, the Voronoi sites denote the facilities, such as hotels, hospitals, restaurants, etc. Thus, each Voronoi region is the service area for each facility. The closest facility for the clients located in a region is the same. From the standpoint of the mobile user, this region also can be regarded as the validity region of that facility; the query results for the closest facility should remain the same until the mobile user moves out of this region. Moreover, IP-Dijkstra's algorithm builds a shortest path tree from each facility to all nodes within a certain service area. Then, the shortest path heading to the facility from any node is derived for each service area.

Similar to what the RR algorithm does in LPA*, this incremental approach is used to maintain the one-to-some shortest path trees. Changes in link-weights may affect the tree structure for all trees. As a result, the boundary of each service area will also changee. Queries submitted from the same position may give different answers for each time interval, although the shortest path from any node to the closest facility is updated. This implies that mobile clients can submit en route queries from any position.

The experimental results demonstrate that IP-Dijkstra's algorithm is superior to both Parallel Dijkstra and the indexing approach in that Parallel Dijkstra cannot maintain a dynamic network Voronoi diagram and the indexing approach has to compute the shortest path online. Therefore, they are not as efficient.

Chapter 7: Conclusion

7.1 Summary of the Improved LPA* Algorithm

In this research, I explore a novel approach based on the existing Lifelong Planning A* algorithm (LPA*) to solve the dynamic shortest path problem in navigation where the users have to recalculate the optimal route while traveling in a dynamic environment. Most previous research does not provide an efficient approach for dealing with the shortest path problem for a moving object.

LPA* deals with the dynamic shortest path problem, combining an incremental search method and heuristic concept. Optimal solutions can be found to series of similar path planning problems potentially faster than is possible by solving each path-planning problem from scratch. LPA* reaches this goal by using information from previous search results to speed up later searches. Originally, this algorithm is not capable of solving our problem, in which both the start node and link costs always change over time. My research extends this algorithm by reversing the search direction from the goal to the source and by dynamically modifying the heuristic. In this way, LPA* can be applied to this situation. The experimental result shows that, in most cases, it is more efficient than A*.

In addition, I employ a constrained ellipse to further restrict the search so the performance can be further improved. With the assistance of a constrained ellipse, the improved LPA* can save up to 70~80% in computational costs as compared with A*, which performs independent searches whenever the environment changes.

My proposed approach can be widely used in many areas, such as mobile robotics, in which a robot moves to a set of goal coordinates in unknown terrain. In this case, the robot has to detect any obstacles and recalculate its optimal route before moving to its next position. In future research, I will extend my approach into other application

areas and investigate its suitability.

7.2 Summary of IP-Dijkstra Algorithm

In this research, I proposed a novel approach, IP-Dijkstra, based on the existing Parallel Dijkstra's algorithm and RR approach to solve the dynamic routing problem in navigation where the users have to query the best destination and recalculate the optimal route while traveling in a dynamic environment. Most previous research does not provide an efficient solution to deal with the dynamic routing problem for mobile clients. Nor do they provide a scaleable solution for large numbers of users.

IP-Dijkstra handles the dynamic routing problem by pre-computing the shortest path-based network Voronoi diagram to partition the road network into a set of service areas with respect to the location of facilities, and adjusting the service areas with current traffic conditions. To efficiently construct and maintain the dynamic service areas, I combine a parallel search algorithm and an incremental approach as a hybrid solution to fulfill these requirements. This algorithm can find optimal solutions to series of similar nearest neighbor queries and path planning problems potentially faster than is possible by solving each routing problem from scratch. It reaches this goal by using information from previous search results to speed up later searches.

The experimental result shows that, in most cases, the algorithm can work more efficiently than either Parallel Dijkstra's algorithm or the indexing approach. Compared with static methods, IP-Dijkstra is 3 to 50 times faster than Parallel Dijkstra if less than 10% of the link-weights change. The experimental results also demonstrate that, although the performance of IP-Dijkstra may vary slightly in different networks depending on the connectivity and topological structure, this will not affect its superiority with respect to a static algorithm. The comparison between IP-Dijkstra and the indexing approach shows that, no matter how the indexing approach manages the shortest paths computation, IP-Dijkstra is always superior,

especially in dealing with large number of queries.

Finally, the success of this research satisfies the urgent need in the navigation service area for a search algorithm that can efficiently adapt to the dynamic traffic environment. In future research, I hope to extend this approach to solve the *K*-nearest neighbor problem by utilizing the properties of Voronoi diagram and heuristic searching algorithms. In this way, the mobile clients may have more than one choice as the best destination.

References

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

[2] 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.

[3] OpenGIS - **A** Request for Technology - In Support of an Open Location Services (OpenLS™) Testbed, 2000.

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

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

[6] 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, page. 28-29.

[7] 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, page. 493-498.

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

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

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

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

[12] Frigioni, D., Marchetti-Spaccamela, A., Nanni, U. (1998). "Semidynamic

algorithms for maintaining single source shortest path trees", Algorithmica 22 (3), page. 250–274.

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

[14] 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.

[15] 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.

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

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

[18] Ausiello, G., Italiano, G., Marchetti-Spaccamela, A., Nanni, U. (1991). "Incremental algorithms for minimal length paths", Journal of Algorithms 12 (4), page. 615–638.

[19] Even, S. and Shiloach, Y. (1981). "An on-line edge deletion problem", Journal of the ACM 28 (1).

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

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

[22] 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, page. 212–221.

[23] Goto, S. and Sangiovanni-Vincentelli,A. (1978). "A new shortest path updating algorithm", Networks 8 (4), page. 341–372.

[24] Italiano, G. (1988). "Finding paths and deleting edges in directed acyclic graphs", Information Processing Letters 28 (1), page. 5–11.

[25] Klein, P. and Subramanian, S. (1993). "Fully dynamic approximation schemes for shortest path problems in planar graphs", in: Proceedings of the International Workshop on Algorithms and Data Structures, page. 443–451.

[26] Lin, C. and Chang, R. (1990). "On the dynamic shortest path problem", Journal of Information Processing, 13 (4), page. 470–476.

[27] Rohnert, H. (1985), "A dynamization of the all pairs least cost path problem", in: Proceedings of the Symposium on Theoretical Aspects of Computer Science, page. 279–286.

[28] Spira, P. and Pan, A. (1975). "On finding and updating spanning trees and shortest paths", SIAM Journal on Computing 4, page. 375–380.

[29] Demetrescu, C., Frigioni, D., Marchetti-Spaccamela, A., Nanni, U. (2000). Maintaining shortest paths in digraphs with arbitrary arc weights: An experimental study. *Algorithm Engineering*, page. 218–229.

[30] Even, S. and Shiloach, Y. An On-Line Edge-Deletion Problem, Journal of the ACM (JACM), v.28 n.1, p.1-4, Jan. 1981

[31] Demetrescu, C. (2001). *Fully Dynamic Algorithms for Path Problems on Directed Graphs*. PhD thesis, Department, of Computer and Systems Science, University of Rome “La Sapienza”.

[32] Frigioni, D., Marchetti-Spaccamela, A., Nanni, U. (1996). Fully dynamic output bounded single source shortest path problem. *Proceedings of the 7th annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, page. 212–221.

[33] Frigioni, D., Ioffreda, M., Nanni, U., Pasqualone, G. (1998). Experimental analysis of dynamic algorithms for the single source shortest path problem. *ACM J. of Exp. Alg.*, 3, article 5.

[34] Ramalingam, G. and Reps, T. (1996). On the computational complexity of dynamic graph problems. *Theoretical Computer Science*, 158, page.

233–277.

[35] Okabe, A., Boots, B., Sugihara, K., Chiu, S.N. (2000). "*Spatial Tessellations, Concepts and Applications of Voronoi Diagrams*", John Wiley and Sons Ltd., 2nd edition.

[36] Roussopoulos, N., Kelly, S., Vincent, F. (1995). Nearest Neighbor Queries. *SIGMOD*.

[37] Samet, H. and Hjaltason, G. (1999). Distance Browsing in Spatial Databases. *ACM TODS*.

[38] Zhang, J., Zhu, M., Papadias, D., Tao, Y., Lee, D. (2003). Location-based spatial queries,. SIGMOD.

[39] Preparata, F.P. and Shamos, M.I. (1985). "Computational geometry", Springer-Verlag, Berlin.

[40]. Erwig, M. (2000). "The Graph Voronoi Diagram with Applications", *Networks, Vol 36*, No. 3, page. 156-163.

[41] Kolahdouzan, M. and Shahabi, C. (2004). "Voronoi-Based K Nearest Neighbor Search for Spatial Network Databases", Proceedings of the 30th VLDB Conference, Toronto, Canada.

[42] Preygel, A. (1999). "Pathfinding: A Comparison of Algorithms", Magnet Science page 3.

[43] Chabini, I. (1997). "A new algorithm for shortest paths in discrete dynamic networks", as presented at the 8th IFAC/IFIP/IFORS Symposium on transportation systems, Technical University of Crete, Greece, 16-18 June.

[44] Cooke, K.L. and Hasley, E. (1966). "The shortest route through a network with time-dependent intermodal transit times", Journal of Mathematical Analysis and Applications, vol. 14, page. 493-498.

[45] Dreyfus, S.E. (1979). "An appraisal of some shortest path algorithms", Operations Research, vol. 17, page. 395-412.

[46] Koenig, S., Likhachev, M., Furcy, D. (2004). "Lifelong Planning A*", Artificial Intelligence Journal, 155, (1-2), page. 93-146.

[47] Koenig, S. and Likhachev, M. (2002). "Incremental A*", In Advances in Neural

Information Processing Systems (NIPS).

[48] Djidjev, H., Pantziou, G., Zaroliagis, C. (2000). "Improved Algorithms for Dynamic Shortest Paths", Algorithmica, Vol. 28, No 4, page. 367-389.

[49] Adusei, I.D., Kyamakya, K., Erbas, F. (2004). "Location-based services-advances and challenges", In Proceeding Of the ITCC 2004 Mobile Enterprises : Enabling Applications/Enabling, USA.

[50] Buriol, L.S., Resende, M.G.C., Thorup, M. (2003). "Speeding up dynamic shortest path Algorithms", AT&T Labs Research Technical Report TD-5RJ8B.

[51] Frigioni, D., Marchetti-Spaccamela, A., Nanni, U. (2000). "Fully dynamic algorithms for maintaining shortest paths trees", Journal of Algorithms, 34 (2), page. 251–281.

[52] Gupta, P., Jain, N., Sikdar, P.K., Kumar, K. (2003). Geographical Information Systems in Transportation Planning, Map Asia Conference.

[53] Singh, A. K., Sikdar, P. K., Dhingra, S. L. (1999). Geographic Information System: Information Technology for Planning and Management of Transport Infrastructure in the Next Millennium, 59th Annual Session of the Indian Roads Congress, Hyderabad, 31st January to 3rd February.

[54] Zwick, U. (2001). Exact and approximate distances in graphs - a survey. In Proc. ESA 2001, vol. 2161 of Lecture Notes in Computer Science, page. 33-48.

[55] Pearl, J. (1984). Heuristics: Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley. p. 48