Shortest Path Algorithms for Dynamic Transportation Networks

Wedad Alhoula

A thesis submitted in partial fulfilment of the requirements of The Nottingham Trent University for the degree of Doctor of Philosophy

PhD

May 2019

ABSTRACT

Over the last decade, many interesting route planning problems can be solved by finding the shortest path in a weighted graph that represents a transportation network. Such networks are private transport networks or timetabled public transportation networks. In the shortest path problem, every network type requires different algorithms to compute one or more than one shortest path. However, routing in a public transportation network is completely different and is much more complex than routing in a private transport network, and therefore different algorithms are required.

For large networks, the standard shortest path algorithms - Dijkstra's algorithm (1959) and Bellman's algorithm (1958)- are too slow. Consequently, faster algorithms have been designed to speed up the search. However, these algorithms often consider only the simplest scenario of finding an optimal route on a graph with static real edge costs. But real map routing problems are often not that simple – it is often necessary to consider time-dependent edge costs. For example, in public transportation routing, consideration of the time-dependent model of these networks is mandatory.

However, there are a number of transportation applications that use informed search algorithms (where the algorithm uses heuristics that guide the search toward the destination), rather than one of the standard static shortest path algorithms. This is primarily due to shortest paths needing to be rapidly identified either because an immediate response is required. For example, the A* algorithm (Nilsson, 1971) is widely used in artificial intelligence. Heuristic information (in the form of estimated distance to the destination) is used to focus the search towards the destination node. This results in finding the shortest path faster than the standard static search algorithms.

Road traffic congestion has become an increasingly significant problem in a modern society. In a dynamic traffic environment, traffic conditions are time-dependent. For instance, when travelling from home to the work, although an optimal route can be planned prior to departure based on the traffic conditions at that time, it may be necessary to adjust the route while en route because traffic conditions change all the time. In some cases, it is necessary to modify the travelling route from time to time and re-plan a new route from the current location to the destination, based on the real-time traffic information. The challenge lies in the fact that any modification to the optimal route to adapt to the dynamic environment necessitates speeding up of the search efforts. Among the algorithms suggested for the dynamic shortest path problem is the algorithm of Lifelong Planning A* algorithm (LPA*) (Koenig, Likhachev and Furcy, 2004). This algorithm has been given this name because of its ability to reuse information from previous searches. It is used to adjust a shortest path to adapt to the dynamic transportation network.

Search space and fast shortest path queries can be used for finding fastest updated route on road and bus networks. Consequently, the efficient processing of both types of queries is of first-rate significance. However, most search methods focus only on one type of query and do not efficiently support the other. To address this challenge, this research presents the first novel approach; an Optimised Lifelong Planning A* (OLPA*) algorithm. The OLPA* used an appropriate data structure to improve the efficiency of the dynamic algorithms implementation making it capable of improving the search performance of the algorithm to solve the dynamic shortest path problem, which is where the traveller may have to re-compute the shortest path while travelling in a dynamic transportation environment.

This research has also proposed bi-directional LPA* (BLPA*) algorithm. The proposed algorithm BLPA* used bi-directional search strategy and the main idea in this strategy is to divide the search problem into two separate problems. One search proceeds forwards from the start node, while the other search proceeds backwards from the end node. The solution requires the two search problems to meet at one middle node. The BLPA* algorithm has the same overall structure as the LPA* algorithm search, with some differences that the BLPA* contains a priority queue for each direction.

This research presented another algorithm that designed to adaptively derive the shortest path to the desired destination by making use of previous search results and reducing the total execution time by using the benefits of a bi-directional search strategy . This novel algorithm has been called the bi-directional optimised Lifelong A* algorithm (BiOLPA*). It was originally proposed for road transport networks and later also applied to public transportation networks. For the road transport network, the experimental results demonstrate that the proposed incremental search approach considerably outperforms the original approach method, which recomputed the shortest path from scratch each time without utilization of the previous search results. However, for public transportation, the significant problem is that it is not possible to apply a bi-directional search backwards using estimated arrival time. This has been further investigated and a better understanding of why this technique fails has been documented. While the OLPA* algorithms give an impressive result when applied on bus network compared with original A* algorithms, and our experimental results demonstrate that the BiOLPA* algorithm on road network is significantly faster than the LPA*, OLPA* and the A* algorithms, not only in terms of number of expansion nodes but also in terms of computation time.

ACKNOWLEDGMENTS

In the name of Allah, the Entirely Merciful, the Especially Merciful.

“I only intend reform as much as I am able. And my success is not but through Allah. Upon him I have relied, and to Him I return. “ Surah Hud (88)

“We raise in degrees whom we will, but over every possessor of knowledge is one [more] knowing.” Surah Yusuf (76)

The research work which resulted in this thesis has been carried out under the supervision of Dr. Joanna Hartley. I am most grateful for her continuous support of my Ph.D. study and related research, for her patience, motivation, and immense knowledge. Her guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Ph.D. study.

I wish to express my sincere thanks to my parents for their continual encouragement and support throughout this work. I am extremely thankful and indebted to my husband Mr Abdul Hakim for taking care of my children (Sarah, Mohammad, Sereen, Haroon and Sally) and for the unceasing encouragement, patience, motivation, support and attention throughout this research.

I am also grateful to all my brothers and sisters especially Mr Abdul Aziz and Dr. Aisha for their supporting me spiritually throughout writing this thesis and my life in general. I must also acknowledge the support given by my close friend Dr. Hayat AlZahrani, for her encouragement regarding this project and my life in general.

COPYRIGHT

LIST OF CONTENTS

List of Figures

List of Tables

List of Abbreviations

Cases

BFS : Breadth-First Search

BiOLPA* : Bi-directional of the OLPA* Algorithm

BLPA* : Bi-directional of the LPA* Algorithm

DFS: Depth-First Search

DSP : Dynamic Shortest Path

DynamicSWSF-FP : Ramalingam and Reps' algorithm

FIFO : Last In, First Out

ITS : Intelligent Transportation Systems

KSP : Set of K-Shortest Paths

LC: Label Correcting Algorithm

LPA* : Lifelong Planning A* Algorithm

LS: Label-Setting Algorithm

OLPA* : Optimised LPA* Algorithm

rhs: Right Hand Search (one-step look-ahead value)

RR L: Ramalingam and Reps' Algorithm

SP : Shortest Path

SSHP : Stochastic Shortest Path Algorithm

TDSP : Time-Dependent Shortest Path Algorithm

CHAPTER 1

INTRODUCTION

1.1 Shortest Path Problem

The Shortest Path (SP) problem is essentially, an optimisation problem (Dimitris, et al, 2013; Vitaly et al, 2005; Frank, et al, 2014; Hazem et al, 2007; Songhua et al, 2010 and Yue et al, 2006). The aim of the shortest path problem is to find the optimal path from the source to the destination. “Optimal” can refer to the shortest time, shortest distance, or least total path cost. This research study has focused on finding the optimal path with the shortest time duration. Shortest path problems have a wide-range of applications in areas such as communications (Yue et al, 2006), vehicle navigation systems (Dimitris, et al, 2013; Frank et al, 2014) and game development (Steve and Nathan, 2013). This research study has focused on the area of transportation in urban environments (vehicle navigation systems).Vehicle navigation systems depend on three main tasks: positioning (locating the vehicle using GPS), routing (computing an optimal route from a source location to a destination location using algorithms), and guidance (providing visual and audio feedback on the route). This research focused on the task of the routing algorithms. Many interesting route-planning problems can be solved by finding the optimal path in a weighted graph representing a transportation network. Such networks are natural road networks or timetabled networks of public transportation.

1.2 Dynamic Traffic Routing

Traffic congestion is a serious problem that affects the mobility of people in society. People live in one area of the city and work in another. They also visit friends and family living in different parts of the country. Intelligent Transportation Systems (ITS) have worked towards improving the efficiency of the transportation networks using advanced processing and communication technology. The analysis and operation of these systems necessitates a variety of models and algorithms.

When a traveller travels from home to work, he/she can plan his optimal route before his departure, based on the current traffic conditions of the transportation network. However, it may not be the final optimal route due to frequent changes in traffic conditions. Therefore, the route needs to be modified while en-route, and a new path planned from the current location to the destination based on current real-time traffic conditions.

Dynamic shortest paths route finding is a fundamental problem in the field of ITS applications. The proposed dynamic shortest path algorithms (developed during this research project) will decrease the search efforts and reduce the computational time when finding alternative paths (due to incidents, for example). Dynamic routing requires the re-planning of a route based on updated dynamic information becoming available during travel. In this research, dynamic information is referred to as time-dependent deterministic information (not stochastic information where random events become available over time rather than at the time of departure from the node).

This research used Nottingham city's network as an experimental study to evaluate the proposed algorithms (road and bus networks). Nottingham city's network has been used by other researchers as their case study. Wu and Hartley (2005) investigated some of the approaches to solving the shortest path(s) problem in a stochastic time-dependent scheduled

transportation network. In (2004), they also developed two different solutions – single-purpose shortest path algorithms and the *K* shortest path algorithm (is an extension algorithm of the shortest path routing algorithm in a given network. *K* shortest path algorithm is used to find more than one path. Some experiments have conducted based on the public transportation network of Nottingham City. However, they did not consider updating routes based on updated traffic information. Similarly, the optimal dynamic paths in Nottingham's bus network cannot be satisfactorily solved by navigation systems (e.g., Google Maps), as they are not able to update routes based on current and dynamic traffic conditions. For bus network route optimisation in Nottingham city, Google Maps finds the routes based on the scheduled timetable only (Driving Directions and Maps ,2014).

In this thesis, three dynamic shortest path algorithms have been developed and evaluated. These efficient algorithms find the fastest alternative paths to a goal node when random traffic incidents occur. The problem that is being solved is finding the updated quickest route (least travel time) from the source node to the destination node. These algorithms have been successfully tested in both road and bus networks.

1.3 Motivation

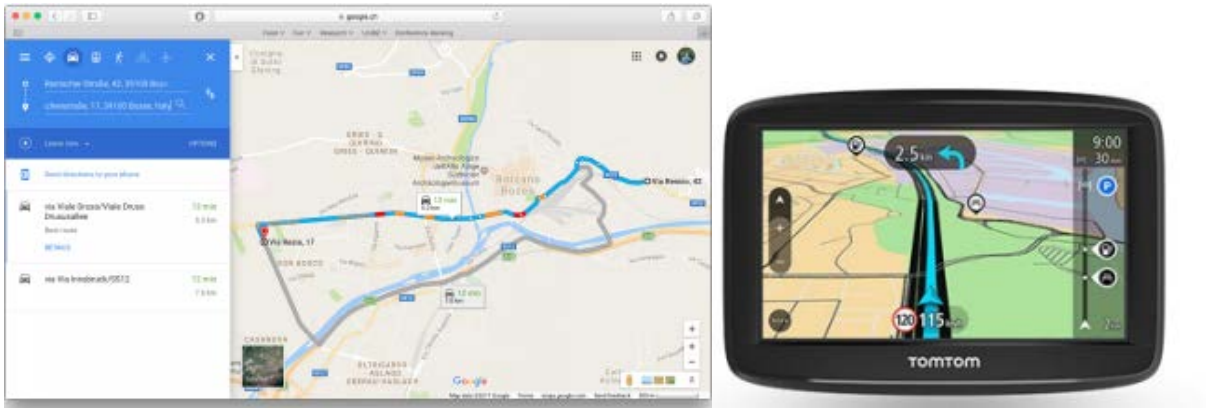
The main challenge of finding the optimal route while traveling is related to the fact that real-time traffic information is not static data. The computed shortest path needs to be updated based on current traffic conditions. However, some travel link times will remain the same and will not need to be updated. Consequently, it is possible to use the unchanged links of a previous search to speed-up the new search and reduce the computational time. This is necessary as a driver does not like to wait while his route is being computed, particularly in the case of emergency situations. Therefore, the planning of alternative routes must be done very fast.

The main challenge of this research is to produce a new more efficient technique, which can use the benefits from the previous search information and subsequently, reduce the route computation time.

Many of the studies on routing techniques have investigated how to solve shortest path-planning problems on static networks (e.g., Dijkstra's algorithm (1959) and Bellman's algorithm (1958)). Where there are changes based on the updated dynamic information, these algorithms need to search for an updated route from scratch. However, this can be inefficient when dealing with a large network with frequent changes. A complete update of the best shortest path can be considered inefficient because some of the previous information results can be reused. Many of these standard algorithms use either the Breadth First Search (BFS), or Depth First Search (DFS) approach. These are uninformed search algorithms (also called blind algorithms) because they don't have any knowledge about the problem domain (except for the source and destination locations). These algorithms are very inefficient, as they update routes from scratch (Miller and Shaw, 2001).

In some cases, the updated route has to be computed in a few seconds. Moreover, when large real road networks are involved in an application, the determination of the shortest path in a large network is very intensive. The increasing popularity of online navigation systems using road networks (e.g., Google Maps, OpenTripPlanner and OpenStreetMap) has recently attracted a considerable interest from the scientific researchers. Distance and shortest path queries are an integral part of applications such as Google Maps and GPS navigator. A distance query returns the length from a start node to a goal node, while the shortest path query calculates the shortest actual route starting from a start node to a goal node. Figure 1.1a

illustrates GoogleMaps, the most popular web mapping service, while Figure 1.1b shows a GPS navigator for cars developed by TomTom.



(a) Google Map

(b) TomTom navigator for cars

Figure 1.1 Two applications offering routing services

This research, will focus on a search space instead of a distance query that returns the number of node expansions from a start node to a goal node. The classic solution for both search space and shortest path queries is Dijkstra's algorithm (1959). Google Maps and most navigation applications (e.g., OpenTripPlanner) initially used Dijkstra's algorithm (1959) to find the most efficient route. (Lanning, Harrell and Wang, 2014) Dijkstra's algorithm works on a static network (where the edge weights on the network are static and deterministic). It works by examining the closest node to the start node. However, despite its simplicity, Dijkstra's algorithm is inefficient for large road networks. This algorithm has high time complexity and

takes up a larger amount of storage space. A more detailed description of Dijkstra's algorithm has been provided in section 3.3.1.

To achieve better performance, a variety of speeding up techniques have been proposed (Bast et al., 2014) (Sommer, 2014; Wu. L et al., 2012). In particular in relation to search space, the first improvement of Dijkstra's algorithm was bi-directional search (Pohl, 1969); starting the search from a start node and an additional search from a goal node, performed in a backwards direction, with the termination search occurring when both directions meet. The Bounded-hop Method reduces search space (Cohen et al., 2002; Akiba et al., 2014; Abraham et al., 2011). In the field of shortest path queries, the most efficient method is the Hierarchical method family, which pre-computes a hierarchy of shortcuts and applies it to process the queries (Geisberger et al., 2008; Sanders and Schultes, 2005; Zhu et al., 2013). All approaches focus on a single type of query, either search space or shortest path.

However, there are a number of transportation applications that use informed search algorithms, rather than one of the standard static shortest path algorithms. This is primarily due to shortest paths needing to be rapidly identified either because an immediate response is required (e.g., in-vehicle route guidance systems) or because the shortest path needs to be recomputed repeatedly (e.g., vehicle routing and scheduling). For this reason, a number of different heuristic shortest path algorithms have been investigated for the purpose of reducing the execution time of the shortest path algorithms. For example, the A* algorithm (Nilsson, 1971) is widely used in artificial intelligence. Heuristic information (in the form of estimated distance to the destination) is used to focus the search towards the destination node. This results in finding the shortest path faster than the standard static search algorithms.

The A* algorithm is not the best approach for route finding under dynamic traffic conditions, because the A* algorithm re-computes the shortest route from scratch. A solution to this problem is to change the A* algorithm from a re-plan strategy to a reuse strategy, in order for it to be suitable for any updated dynamic data. (Russell and Norvig 2009). Reuse planning attempts to use as many calculations of the previous plan as possible. Re-planning does not have this requirement (Koenig et al, 2005).

Another approach to speed up searches is an incremental search. An incremental search is a search technique for re-planning and reuses the previous search. This results in finding solutions faster than when solving each search problem from scratch. Some existing incremental shortest path algorithms can use the benefit (of previous search information) to reduce the computation time (e.g., D* algorithm (1994), Focused D* algorithm (1995) and D*-Lite (2002)). These algorithms are used when there is incomplete information (e.g., unknown destination). They can find the shortest paths from the source node to all other nodes in the graph and are able to quickly recalculate the route. Ramalingam and Reps' algorithm (Ramalingam, 1996) (RR for short, also known as the DynamicSWSF-FP algorithm) starts searching from the destination node to all other nodes. After dynamic changes have occurred, the algorithm updates only the nodes whose link travel time has changed. The DynamicSWSF-FP algorithm is the most useful when finding the distance from the destination node to multiple nodes after update associated with traffic information. Such an incremental approach is often used in robotics, navigation, and planning.

It is clear that heuristic search algorithms are guaranteed to find the shortest path faster than static search algorithms. Incremental search algorithms are guaranteed to find the shortest paths faster than algorithms that solve each path re-planning problem from scratch. Koenig, et al,

(2005) developed the LPA* algorithm. It is a fully dynamic shortest path algorithm that is used to incrementally find the shortest path from a known source to a known destination in a given graph (as links or nodes are removed or added, or the travel time of the links changes). The LPA* algorithm is a reusing method rather than a re-planning method. It combines the DynamicSWSF-FP and A* algorithms. The combination of these two algorithms results in speeding up the search and reducing the computation search time. It is able to adjust the shortest path to adapt to the dynamic transportation network and is guaranteed to find the shortest path faster than both the DynamicSWSF-FP and A* algorithms methods individually.

However, when used in very large networks, this algorithm needs to be made more efficient. Therefore, this thesis has further developed the LPA* algorithm, as a fast re-planning method named Optimised LPA* (OLPA*). The name was chosen because it is able to reduce the running time by improving the search performance of the LPA* algorithm. The OLPA* algorithm uses a priority queue dictionary instead of an open set. The priority queue dictionary (**pqdict**) is implemented based on the heap data structure of (key, priority value) the pairs. The priority queues dictionary is useful in applications where the priorities of items may frequently change (e.g., optimisation algorithms, simulations, etc.) (Beazley, 2015). The set of predecessors of the node were implemented as a priority queue dictionary (**pqdict**) rather than an open set, and this is a novel idea. It provides $O(1)$ to search and retrieve items with the highest priority regardless of the number of items in the queue. This is instead of $O(n)$ in terms of big O notation.

The main reason for choosing the incremental LPA* algorithm is, firstly, that it finds the shortest path from the known start node to the known goal node. The D*, Focused D*, D*-Lite (2002)) and the DynamicSWSF-FP algorithms do not search from two known locations.

Secondly, when the first shortest path gets blocked, the LPA* algorithm is able to take advantage of the previous calculations. The A* and the Dijkstra's algorithms do not take this advantage from the previous calculations. The A* and the Dijkstra's algorithms need to re-compute the route from scratch.

This research has also focused on how to reduce search space, which can further speed up the search. For this important point, this thesis has proposed a novel algorithm that is able to speed up the search via a heuristic search method. This is the bi-directional heuristic search algorithm (Pohl, 1969). A bi-directional method is used to reduce the search space and time by searching forward from the start and backward from the goal, simultaneously. When the two search frontiers intersect, the algorithm can reconstruct a single path that extends from the start nodes through the frontier intersection to the destination. For example, in a search problem modelled by a tree with branching factor b and solution depth d, a bi-directional search will expand $2b^{d/2}$states instead of the b^d required by a unidirectional search. The bi-directional Dijkstra's algorithm is an example of this technique. (Padua, 2011).

This thesis has also proposed a novel algorithm called the Bi-directional Lifelong A* algorithm (BiOLPA*). The BiOLPA* algorithm searches forwards from the start node and backwards from the destination node using a novel search strategy. This proposed search strategy is called the autonomous strategy**.** It improves the strategy of node selection in the algorithm and increases the search speed by searching forwards and backwards simultaneously, searching alternatively such as via Poul's strategy (1969), or based on the number of nodes in both priority queues. The side that has the fewest number of nodes in the priority queue is started and expanded first. This strategy was proposed by Poul (1971), named the cardinality comparison strategy. In this strategy, the algorithm will decide the direction (forward or

backward) based on the number of nodes in both priority queues. The side that has the fewest number of nodes in the priority queue will start to expand first. This thesis proposes a novel strategy called an autonomous strategy. We chose this name because we do not implement exactly in the program which direction to start to search first, and the algorithm itself decides based on the heuristic values. The autonomous strategy chooses the most promising node that has the highest probability of being on the shortest path and has the smallest $f(value)$ in relation to both priority queues as shown in (1.1).

$$f(value) = g(n) + h(n) \quad 1.3)$$

For more details, see section 5.2.1.2. However, the BiOLPA* algorithm can adjust the shortest path to adapt to the dynamic transportation network and guarantee to find the shortest path faster than both the OLPA* and the bi-directional search methods individually because it combines their techniques.

Consequently, in this research, a novel algorithm has been developed that combines an incremental search algorithm with a bi-directional heuristic search approach. See Figure 1.2.

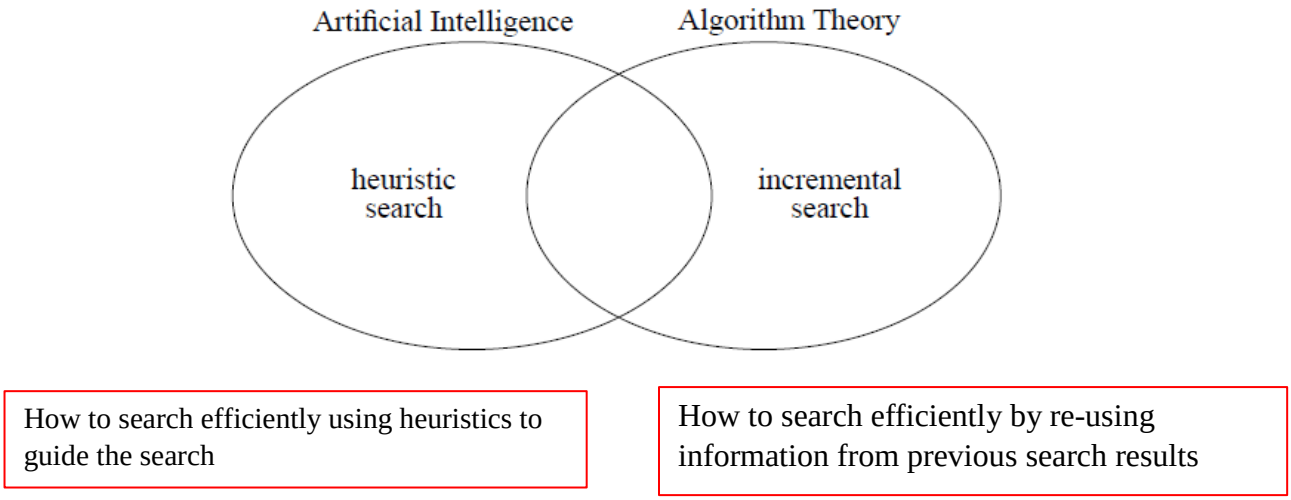


Figure 1. 2 Combination incremental search and heuristic search.

Two different ways of decreasing the search efforts for determining the shortest path have been investigated.

• Firstly, some of the edge costs are not affected by the changes, and thus do not need to be recomputed. Heuristic function knowledge, in the form of approximations of the goal distance, can be used to speed up the search. This is what the **OLPA*** algorithm does.

• Secondly, the heuristic searching strategy, using the heuristics from the start node to the goal node and the heuristics from the goal node to the start node, will reduce the search space and then speed up the search by half. This is what the **bi-directional search method** does.

This research also presents a bi-directional implementation of the LPA* algorithm. Named the **BLPA*** algorithm, the proposed algorithm combines the benefits of the bi-directional search and re-uses a previous calculation in an updated search method. The BLPA* algorithm uses Pohl's bi-directional strategy (1971), also named a "cardinality comparison strategy" (monotonicity hypothesis). Cardinality comparison strategy chooses the direction with the smaller priority queue, rather than simply alternating the directions. This thesis demonstrates that the two ways of decreasing the search effort are efficient by developing the BiOLPA* algorithm that combines both and can update the route faster than either the OLPA* algorithm or bi-directional search method. However, navigation systems such as GoogleMaps have never published information about their routing algorithms. Therefore, the proposed algorithms will be compared with other published algorithms such as the A* and LPA* algorithms and their variants in terms of performance (number of node expansion and computation time).

1.4 Blocked links and alternative route

To improve the effectiveness of travel information in real-world scenarios, determining solely the shortest path is not enough. There is a need to compute an alternative route. For example, when some links on the computed shortest path are blocked, it is necessary to update the shortest path and find an alternative route to the goal node. Most commercial route planning applications and navigation systems recommend alternative paths that might be longer than the shortest path (Chondrogiannis et al, 2015).

For example, the exist LPA* algorithm, in case of a blocked link that is close to a goal node and there is no alternative route available close to the blocked link, the second search (updated route) will take a long time to find a goal node because the LPA* algorithm will have lost its benefits from reusing the previous calculation. In this case, the A* algorithm that updated the route from scratch will be faster than the LPA* algorithm. This research shows that the BiOLPA* algorithm outperforms the existing LPA* and A* algorithms, and always updates the route faster than all existing algorithms regardless of the location of the blocked link.

1.5 Objectives

This research project aims to reduce the search space and speed up the computation time in dynamic networks when determining the optimal route and computing an alternative route (when network changes have occurred). The proposed approaches focus on both the search pace and shortest path queries.

The algorithms were tested on the road and bus timetable networks in the context of Nottingham’s urban network.

1. Speeding up the search process by making use of the previous search results and using appropriate data structures to improve the efficiency of the dynamic algorithms. This objective was implemented by improving the LPA* into the developed **OLPA*** algorithm.
2. Reducing the search space by three ways. **Firstly**, a bi-directional search method is used. **Secondly**, the previous information results are used (OLPA* algorithm). **Thirdly**, a novel search strategy (**autonomous strategy)** is used to improve the strategy of node selection within the algorithm. This search process helps the search to avoid node re-expansion from both directions and therefore speeds up the search for the shortest path more. These points can be achieved by proposing a novel algorithm called the **BiOLPA*** algorithm.

1.6 Original Contributions

1.6.1 Overview

This project proposes two novel efficient algorithms in two types of networks (road and bus timetable). These algorithms are significantly more efficient than standard algorithms, not only in terms of reducing route computation time, but also with regards to reducing the search space.

1.6.2 Optimised LPA* Algorithm

This research has developed the Optimised Lifelong Planning A* (OLPA*) algorithm, a fast re-planning method that is an improvement of the LPA* algorithm. The OLPA* algorithm is

faster than the LPA* algorithm. It reduces route computation time in the first search (when finding the optimal route) and in the second search (when some links are blocked and an alternative route needs to be computed). This algorithm has been implemented and applied to both road and bus timetable networks.

1.6.3 Bi-directional OLPA* Algorithm

This research has also proposed the bi-directional optimised Lifelong A* algorithm BiOLPA* algorithm, which is designed to reduce the search space to speed up route computation time, by benefiting from bi-directional heuristic searching and the use of previous information results. Using the **autonomous strategy**, the BiOLPA* algorithm has been implemented in the road and bus timetable networks.

1.6.4 Novel Autonomous Search Strategy

This search has presented a novel search strategy (autonomous strategy) to enhance the intelligence of node selection in the algorithm. The strategy determines the best selection of either the forward or backward direction. This search method has contributed to increased search speed instead of using forward and backward searching alternatively. This selection process avoids nodes from re-expansion from both the forward or backward direction, and therefore produces the shortest path more quickly.

1.6.5 Bi-directional LPA* Algorithm

This research has also proposed the bi-directional LPA* (BLPA*) algorithm that benefits from the bi-directional heuristic search, the use of previous calculations and Pohl's (1971) bi-directional strategy named the "cardinality comparison strategy" (monotonicity hypothesis). The cardinality comparison strategy chooses the direction with the smaller priority queue, rather than simply alternating the directions. The BLPA* algorithm has been implemented and compared with the BiOLPA* algorithm.

1.7 Outline of the thesis

We have organised the rest of the thesis in the following way.

Chapter 2: This chapter introduces the background of the graph theory, graph representation in the memory, fundamental concepts (i.e. the definition of a graph, the degree of a graph, and the definition of a path) at the beginning of this chapter. In the discussion of the degree of a graph, dense graph and sparse graph have been defined and used in the data model discussion.

Chapter 3: The chapter classifies the common search strategies, including uninformed search, informed search, and incremental search. The two classic SP algorithms (Dijkstra's and the A* algorithms) will be presented in detail. Some related research on the time-dependent shortest path (TDSP) and stochastic shortest path (SSHP) problems will be also put forward. The incremental shortest path algorithms, the Dynamic SWSF-FP Algorithm and the LPA* algorithm, will be explained and discussed in detail.

Chapter 4: Introduces the dynamic single pair shortest path problem. A novel contribution to the OPLA* algorithm will be presented and discussed in details. A demonstration of updated routes using the OLPA*, A* and LPA* algorithms for road and bus networks will also be illustrated and discussed.

Chapter 5: In this chapter, a novel algorithm for dynamic road networks will be presented, and the BiOLPA* algorithm will also be presented and discussed in detail. A demonstration of an updated route using BiOLPA*, A*, LPA* and OLPA* for the road network has been illustrated and discussed. As part of this contribution, this chapter introduced a novel search strategy (called Autonomous strategy) to enhance the intelligence of node selection in the algorithm.

Chapter 6: In this chapter, a novel BiOLPA* algorithm has been implemented in both road and bus networks. The experimental results have demonstrated the evaluation of the BiOLPA* compared with the A*, LPA* and OLPA* algorithms in term of computation time and number of node expansions in both road and bus networks.

Chapter 7: This chapter produced a new BLPA* algorithm, and the experimental results demonstrated the evaluation of the BLPA* algorithm compared with the BiOLPA* algorithm.

Chapter 8: Finally, the conclusion has been presented and possible future research work discussed in this chapter.

CHAPTER 2

TRANSPORTATION NETWORK

2.1 Overview of a Graph Theory

In this chapter, some fundamental concepts of graph theory are introduced. In graph theory, the shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized. The concepts in this chapter are essential for understanding later discussions involving graphs.

2.1.1 Definition of a Graph

A graph is a mathematical structure consisting of a set of nodes or vertices, connected by a set of links, known as edges, this research, a graph $G = (V, E)$ where

- V is a set of nodes.
- E is a set of edges.
- Each edge is a pair of nodes.

The graph can be directed or undirected.

Directed graph

A directed graph is a graph consisting of a set of nodes that are connected by a group of links and all links have a specific direction. In a directed graph, edges are written using parentheses to denote ordered pairs. For example, edge (1,2) is directed from 1 to 2, which is different than the directed edge (2,1) from 2 to 1. Directed graphs are drawn with arrowheads on the links, as shown in Figure 2.1 (Sedgewick and Wayne, 2011).

Example: The Figure 2.1 shows the following graph:

- V = {1,2,3,4,5,6}
- E = {{1,2}, {1,3}, {2,3}, {2,5}, {3,4}, {4,6}, {5,6}}
- G = (V, E)

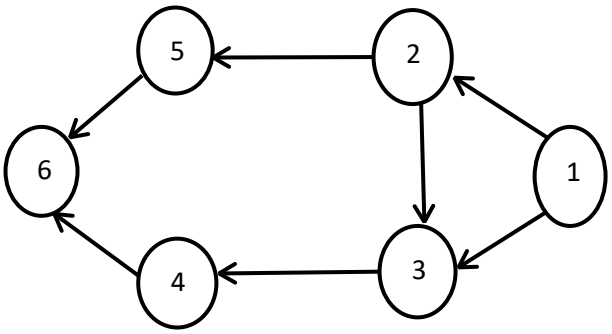


Figure 2. 1 A directed graph.

Undirected graph

An undirected graph is a graph consisting of a set of nodes that are connected by a group of links, where all the edges are two ways (bidirectional). For example, an undirected edge {2,5} from node 2 to node 5 is the same link {5,2} from node 5 to node 2.

The following Figure illustrates an undirected graph; the edges are typically drawn as lines between pairs of nodes. (Sedgewick and Wayne, 2011).

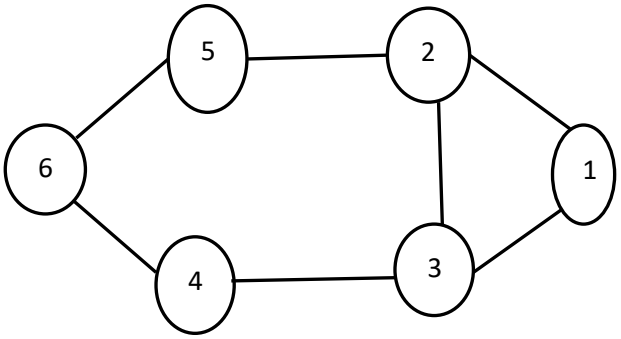


Figure 2. 2 An undirected graph

Weighted graphs

A weighted graph is a graph for which each edge has an associated weight or cost. The edge's cost can be real numbers, which represent a concept such as distance or affinity. Figure 2.3 shows a weighted graph, which shows the cost of $(1 \rightarrow 2) = 4$.

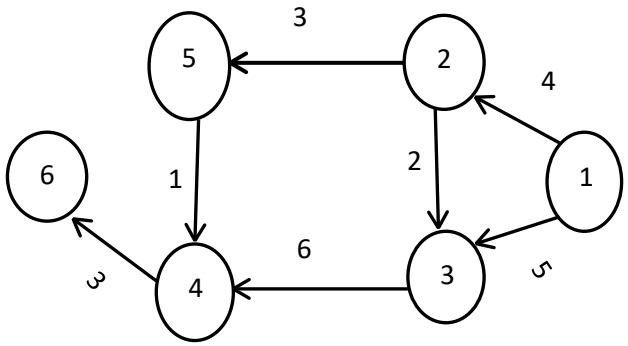


Figure 2. 3 A weighted graph.

A Degree of a Graph

In graph theory, the degree of a node of a graph is the number of links incident to the node. The degree of a node v is denoted $\boldsymbol{deg(v)}$, and the **maximum degree** of a graph G, denoted by $\Delta(G)$, and then $\Delta(G) = \max \{d(v) \mid v \in V\}$.

The **minimum degree** of a graph, denoted by $\delta(G)$, and then $\delta(G) = \min \{d(v) \mid v \in V\}$.

In Figure 2.3 the maximum degree is 6 and the minimum degree is 0 (Diestel, 2005).

The degree sum formula of a given a graph $G\ (V, E)$

$$\sum_{v \in V} \deg(v) = 2|E| \tag{2.1}$$

The average degree of G is

$$d\,(G)\;=\;\frac{1}{|V|}\;\sum_{v\in V}d(v) \tag{2.2}$$

clearly,

$$\delta\,(G)\;\leq\;d\,(G)\;\leq\;\Delta(G) \tag{2.3}$$

The average degree globally quantifies what is measured locally by the node degrees, the number of edges of G per node. Occasionally 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. Consequently,

$$|E|=\frac{1}{2}\sum_{v\in V}d(v)=\;\frac{1}{2}d(G)\times\;|V| \tag{2.4}$$

and thus

$$\varepsilon\,(G)\;=\;\tfrac{1}{2}\;d\,(G) \tag{2.5}$$

Graphs with a number of edges that are roughly quadratic in their order are usually called dense graphs. A dense graph is a graph where $|E|\;=\;|V|^2$ (i.e. the number of edges is about equal to the number of nodes squared). While graphs where $|E|\;=\;|V|$ (i.e. the number of edges is equal to the number of nodes) called sparse graph. Clearly, the average degree $d\;(G)$ for a dense graph will be much greater than that of a sparse graph.

2.1.2 Definition of a Path

A path in a graph represents a route to get from an origin node to a destination node by crossing edges in the graph. For example, in the directed graph G = (V, E) in Figure 2.1, there are many paths from node 1 to node 6. One such path is highlighted in blue:

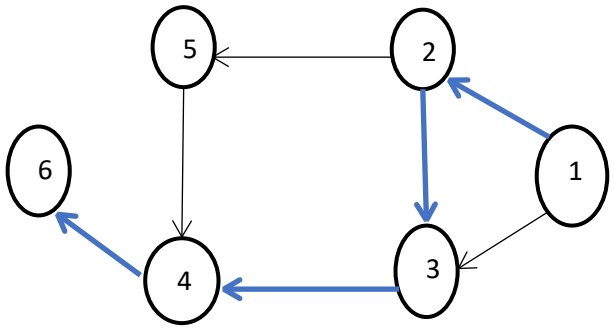


Figure 2. 4 A path in a directed graph.

2.1.3 Length of a path

The length of a path is the number of links that are used to create a connection between nodes.

Example as shown in Figure 2.4

- The length of the blue path is 4

2.1.4 Distance of a path

Given a graph G, the distance $D(a, b)$ between node a and node b is the length of the shortest path from a to b, considering all possible paths in G from a to b. The distance between any vertex and itself is 0. If there is no path from a to b then $D(a, b)$ is infinity (∞).

Examples as shown in Figure 2.4

- the distance from node 1 to node 6 is 3. There is only one path from node 1 to node 6 with length 3, and this path is $\{(1 \rightarrow 3), (3 \rightarrow 4), (4 \rightarrow 6)\}$. All other paths are longer than this path.

2.1.5 Shortest path

A shortest route or path in graph theory is a solution to find the path between two nodes in a graph such that the sum of the costs of its constituent links is minimized. This is an essential concept of graph theory widely practiced. It has proven to be an effective solution to problems regarding several fields, such as communication, routing, road networks. (Sedgewick and Wayne, 2011).

2.2 Graph Representation in Memory

Graph algorithms need efficient access to the graph nodes and edges that are stored in the memory. In typical graph implementations, nodes are implemented as structures or objects and the set of edges establish relationships (connections) between the nodes. There are several possibilities to represent a graph $G = (V, E)$ in memory. Let the set of nodes be $V = \{1, 2, \ldots, n\}$ with edges $E \subseteq V \times V$, the following two are the most commonly used representations of the graph.

2.2.1 Adjacency List

In graph theory and computer science, an adjacency list consists of a list of all nodes in a given graph. each node in the graph associated with the collection of its neighbouring nodes or links. The adjacency list for Figure 2.5 can be described in Figure 2.6.

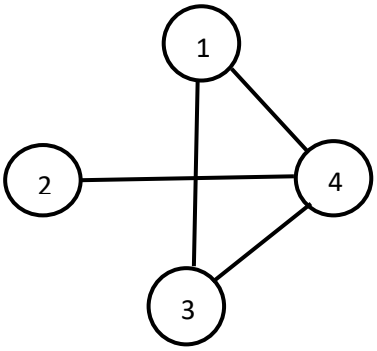


Figure 2. 5 An undirected graph with 4 nodes and 4 links

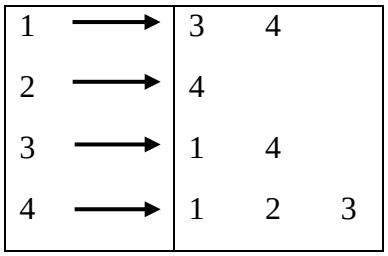


Figure 2. 6 A diagram of adjacency list.

2.2.2 Adjacency Matrix

An adjacency Matrix is another form of graph representation in a memory. The adjacency matrix is a two-dimensional array with rows and columns sorted by the graph vertexes, where each entry x_{ij} is equal to 1 if there exists an edge $=$ (vi, vj) $\in$ E and 0 otherwise.

Node vi and vj 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. Figure 2.5 can be described as an Adjacency Matrix as shown in Figure 2.7.

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

Figure 2. 7 An example of adjacency matrix.

2.3 Transportation Network

A transportation network is a type of directed, weighted graph. In a road network, nodes represented as junctions and edges are the road links between them. For bus timetable network, nodes represented as stops and edges are the links between them. The weights represent travel time between the nodes. This representation is an attempt to quantify the street system for use in a mathematical model.

However, a suitable data structure is required to represent the transportation network in the computer's memory. Comparing the two data structures, the adjacency list is easy to find successors of a node, easy to find all neighbouring nodes and the memory space efficient as it only stores connected nodes, and it does not necessitate space to represent edges which are not present. Based on big O notation that is (used to classify algorithms according to how their running time or space requirements grow as the input size grows. Mohr, (2014)) the space complexity of an adjacency list is $O(|E| + |V|)$, where V is the number of nodes and E is the number of edges in the graph. On the other hand, the adjacency matrix representations contain 0s in the diagonal which are useless in storage, and then the adjacency matrix need more memory to store a large graph, space complexity of adjacency matrix is $O(|V|^2)$.

Using a naive linked list implementation on a 32-bit computer, an adjacency list for an undirected graph requires around $16 \times (|E| + |V|)$ bytes of storage space. In contrast, the adjacency matrix requires only one bit at each entry, it 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;

$$16 \times (|E| + |V|) \geq |V|^2/8 \tag{2.6}$$

Based on Equation 2.2 in Section 2.1.1**,** we have,

$$16 \times (\frac{1}{2} d(G) \times |V| + |V|) \geq |V|^2/8 \tag{2.7}$$

Where $d(G)$ is the average degree of G.

$$d(G) \geq \frac{|V|-128}{64} \tag{2.8}$$

This means that the adjacency list representation occupies more space when Equation 2.8 holds. Firstly, in reality, most transportation networks are large-scale sparse graphs with many nodes but relatively few edges as compared with the maximum number possible ($|V| \times (|V| - 1)$ for maximum). That is, there are no more than 5 links ($\Delta(G) \approx 5$) Linked to each node. In most situations, there are usually 2, 3 or 4 ($\delta(G) = 2$) links, although the maximum links are $|V| - 1$ for each node. Secondly, road networks often have regular network structures and a normal layout, particularly for well-planned modern cities. Thirdly, most transportation networks are near connected graphs, in which any pair of points is traversable along a route. Assuming the average degree of a road network is 5, Equation 2.8 holds only if $|V| \leq 448$. In the real life, most road networks contain thousands of nodes where $|V| > 448$. As a result, Equation 2.3 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. The adjacency matrix size will be 10000 * 10000 around 10 megabytes of memory space is required to store the network, and this is a huge waste of memory. It will most likely take more computational time to manipulate such a large array.

Moreover, the different data structures also facilitate different operations. It is easy to find successors of a node, easy to find all neighbouring nodes in the adjacency list representation by simply reading its adjacency list. While the adjacency matrix, we must search over an entire row, spending $O(V)$ 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 edges vj 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 an edge $vi \rightarrow vj$, since this can be completed in $O(1)$ time by simply looking up the array entry $[vi;\ vj]$. On the other hand, the same operation using an adjacency list data structure requires $O(vj)$ time since each of the $vi \rightarrow vj$ edges in the node list for V must be examined to see if the target node is existing. 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. Moreover, the main operation of LPA* algorithm is finding the successors and the predecessors of a given node. The adjacency list is more feasible for this operation.

Based on the above discussions, it is clear to see 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 time.

2.4 Network Model

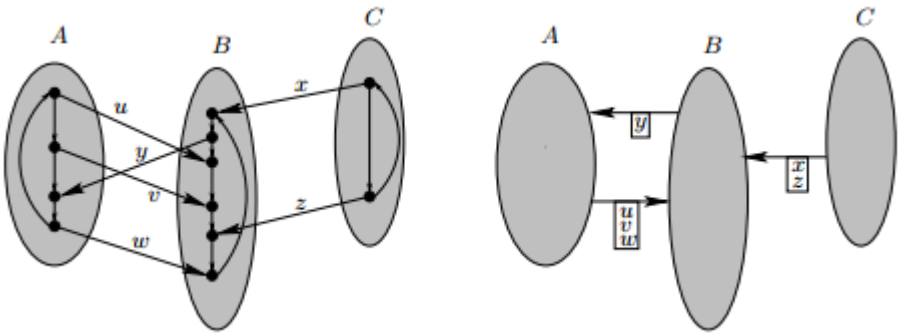
To deal with dynamic travel time the concept of dynamic graph model was introduced in two major approaches: Time-expanded model and time-dependent model. Road and bus timetable networks can be modelled as directed graphs. For bus networks, each node indicates to the bus-stop location, and the edges of the graph correspond to the route links. The cost of the link is presented as travel time.

2.4.1 Time-Expanded Model

A node exists for every event at allocation and links represents the time between these events. In Figure 2.8 (a) where time-expanded model models multi nodes at each bus-stop and each node corresponds to a time (arrival or departure), and each link has a static travel time. To allow transfers with waiting. Dijkstra's algorithm can be used to compute the shortest paths (Patrice and Sang, 1998; Frank et al, 2000; Matthias et al, 2001).

2.4.2 Time-Dependent Model.

Each geographic location is represented as a single node and all dynamic data is stored in the links themselves. The time-dependent model reduces the number of nodes compared to the time-expanded modal, which indicates to be efficient. Figure 2.8 (b) shows the stops graph model uses one node per station and edge models a connection between two stops, the main advantage of this model reduces the number of nodes. (Ariel and Raphael, 1990; and 1991; Hart et al, 1968; Gerth and Riko, 2004).



(a) (b)

Figure 2. 8 The time-expanded graph (a) and the time-dependent graph (b) of a timetable with three stations A, B, C.

There are three trains that connect A with B (elementary connections u, v, w), one train from C via B to A (x, y) and one train from C to B (z). (Daniel Delling, 2007)

In this research, and due to the large network, we select the time-dependent approach. Also, the bus timetable network is modelled as time-dependent so, that stations graph model uses exactly one node per bus-stop.

2.5 Summary

In this chapter, some important information about graph theory is discussed. Because transportation networks are a specialized type of graph, some essential knowledge of graph theory is needed. Some fundamental concepts, such as the definition of a graph, the 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 the data model discussion.

Two types of data models for graph representation are explained, the 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. This research uses the adjacency list to construct the topology of the experimental road and bus network in order to implement the proposed algorithms.

CHAPTER 3

SHORTEST PATH PROBLEM IN TRANSPORTATION NETWORKS

3.1 Introduction

The field of the shortest path problem has been widely researched, since it is a principal issue in transportation networks. Considerable research in computer science and Artificial Intelligence has addressed the question of what criteria ensures a good algorithm for (rapidly) finding a solution path in a given transportation network. This has resulted in the development of several methods for speeding up searches by reducing the search time of the resulting path. This includes using inadmissible heuristics (Pohl, 1970; Pohl, 1973) and searches with a limited look-ahead value (Korf, 1990; Ishida and Korf, 1991; Koenig, 2001). All of the shortest path algorithms presented in this chapter assume that there are used in a directed graphs with a non-negative edges cost, because the field of study is that of a transportation network.

3.2 Search Strategies

There are different strategies for exploring a search space that will be considered in this chapter. This section will focus, in more detail, on an uninformed search (where the algorithm does not make use of any means of estimating how close the search process is to a destination), an informed search (where the algorithm uses heuristics that guide the search toward the destination) and incremental search (a search technique for reusing information from previous

searches to find the updated shortest path solutions faster than is possible by solving each search problem from scratch).

3.2.1 Uninformed Search

3.2.1.1 Depth-First Search

Depth-First Search (DFS) is an unformed search algorithm. It starts by selecting an arbitrary node as a root in a given graph. It examines the nodes as far as possible along each path and then backtracks until it finds an unexplored path. This is then explored. The algorithm does this until the entire graph has been explored. An example has been shown in Figure 3.1. The depth-first search starts at node 1. It must always be assumed that the left hand side edges in the shown graph are chosen before the right hand side edges. It is important to make sure that the nodes visited are marked using a stack (**LIFO** (Last In, First Out)). This will prevent the search from visiting the same node more than once, which may end up in an infinite loop. Two principal operations will be used:

- **Push** adds nodes to the collection.
- **Pop** removes the most recently added nodes that were not yet removed.

In Figure 3.1, DFS will visit the nodes in the following order: **1**, **2**, **4**, **5**, **3** and **6**.

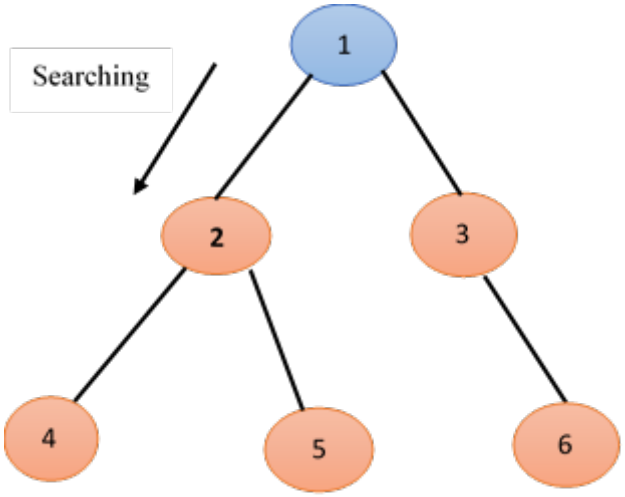


Figure 3. 1 Depth-first search.

3.2.1.2 Breadth-First Search

Breadth-First Search (BFS) is also an unformed search algorithm. It starts by selecting an arbitrary node as the root in a given graph. It examines the neighbouring nodes first, before moving to the next level's neighbours (searching level-by-level). The BFS uses a simple queue FIFO**,** so then the nodes that were inserted first in the queue will be removed first. This makes the current node 'visited' until all of its neighbours (vertices that are directly connected to it) are also marked. An example has been shown in Figure 3.2. The BFS starts at node 1, and visits its child nodes 2, before moving onto 3. It stores the nodes in the order in which they were visited. This will allow the child nodes of 2 to be visited first (i.e. 4 and 5), and then the child node of 3 (i.e. 6). In Figure 3.2, BFS will visit the nodes in the following order: **1, 2, 3, 4, 5 and 6.**

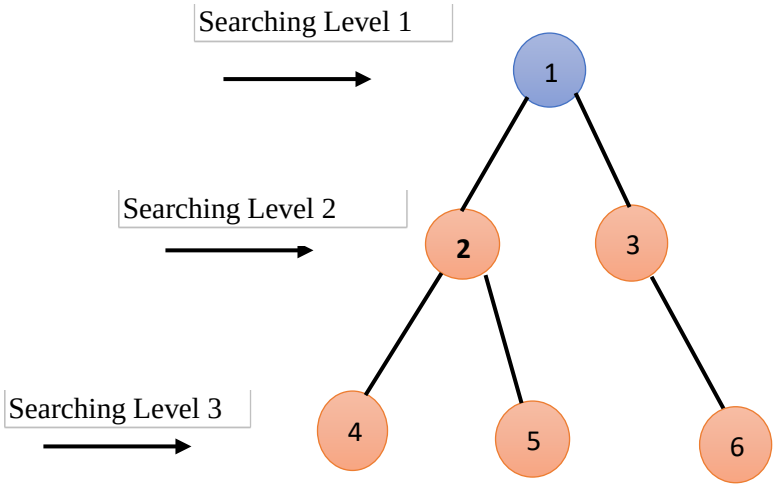


Figure 3. 2 Breadth-first search.

3.2.2 Informed (Heuristic) Search

The main advantage of heuristic strategies is that the search scale can be limited. Many shortest path algorithms based on a restricted searching area have been proposed. For example, the restricted ellipse searching area algorithm. The restricted ellipse area algorithm was first presented by Nordbeck and Bengt (1969), and then extended by Weihong (1995), also in Xingxing and Weihong (1996). The core of the algorithm is that the search area can be reduced greatly, because the set of search nodes is limited to the restricted ellipse searching area. However, the algorithm still involves a large amount of computation and has a high processing time. Consequently, it is not suitable for this application (Fu and Rilett, 2005). There are a number of transportation applications that require the use of a heuristic shortest path algorithm (rather than standard algorithms or optimal algorithms), such as when a shortest path needs to be recalculated repeatedly, for vehicle routing and scheduling. These types of heuristic search attempt to use different sources of additional information to reduce the search efforts. They can be classified into three strategies: decompose the search problem, limit the links searched, and limit the search area,

3.2.2.1 Decompose the Search Problem

This strategy of decomposing the research problem has been divided into two sections: the bi-directional search method and sub-goal method.

3.2.2.1.1 Bi-directional search method

This method is also called a forward and backward search process. The main idea in this strategy is to divide the search problem into two separate problems. One search proceeds

forwards from the start node, while the other search proceeds backwards from the end node. The solution requires the two search problems to meet at one middle node.

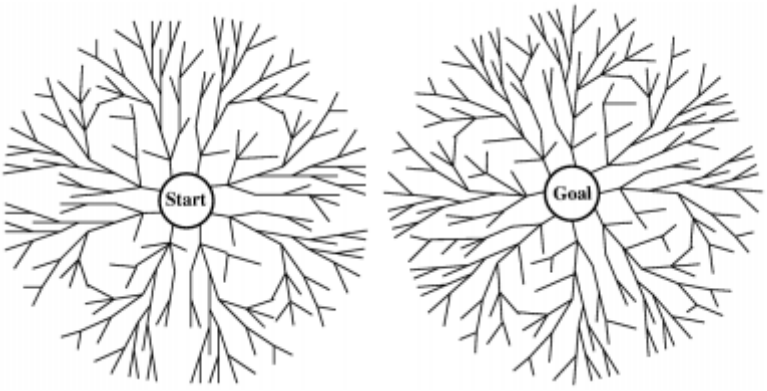


Figure 3. 3 Bidirectional search (taken from *Audrey Carpenter*, n.d).

Bi-directional searching was first proposed by Ira Pohl (1969). He showed that both a forward and backward search can be independently searched simultaneously, instead of two independent searches. Therefore, he combined the two searches into a bi-directional search with each contributing to part of the solution. Figure 3.3 shows a bi-directional search from a source node and from a destination node without finding a path.

Many algorithms use a bi-directional technique to speed up their search. Modifying A* into a bi-directional heuristic search is a possible way to improve performance. The main motivation for using A* in a bi-directional setting is the possible reduction of the number of expanded nodes. In fact, recent results show that the combination of the A* algorithm with a bi-directional search is able to significantly reduce the number of expanded nodes (Whangbo, 2007; Klunder and Post, 2006; Pijls and Post, 2009b).

3.2.2.1.2 Sub-Goal Method

Sub-goal are nodes that are located between the origin node and destination node. To find the shortest path from an origin node to a destination node, the problem can be decomposed into two or more smaller problems. For example, if there is one sub-goal node, the problem divides into two smaller problems: one is to find the SP from the source node to the sub-goal node while the other is to find the SP from that sub-goal to the destination node. The efficiency of this strategy depends on the number and location of the sub-goal nodes (Fu and Rilett, 2005).

3.2.2.2 Limit the Search Links (Hierarchical Search Method)

The main idea of the hierarchical search method is to skip the examination of the links that have a low possibility of being either on the shortest path or used in a specific situation. Firstly, the algorithm concentrates on the essential feature of the problem without considering the lower level details, as it completes the details later. This strategy is similar to when drivers try to find a route between two locations on the map. Firstly, the driver will find the main roads in the area to the origin location and destination location. Secondly, the driver will try to find the access road to the main road from the source to the goal (Fu and Rilett, 2005).

3.2.2.3 Limit the Search Area

The main idea in this strategy is to use some of the information about the attributes of the SPs from the source location to the destination location to constrain the shortest path within a limitation area. This strategy comes in two types: the branch pruning method or A* algorithm.

3.2.2.3.1 Branch Pruning Method

The main idea is to limit the search by excluding the intermediate nodes which have little chance of being on the shortest path to the destination node. In a real-world transportation network, each road is connected to the neighbouring locations and travel time on the road is correlated with its distance. This attribute allows the search area to be constrained within an

exact area surrounding the source location and destination location. The locations outside this area are assumed to have less possibility of being on the shortest path and hence will be skipped in further examination (Fu and Rilett, 2005).

3.2.2.3.2 A Algorithm*

The A* algorithm is one of the most known search algorithms in Artificial Intelligence (Nilsson, 1971). It uses a heuristic function to guide the search. At each iteration, it selects the most promising node according to an evaluation function $f(value)$, which includes the real cost of going to that node and an estimate of the cost from that node to the goal. More details about how the A* algorithm works is in Section 3.3.2.

3.3 Classical Algorithms

3.3.1 Dijkstra's algorithm

Dijkstra's (1959) algorithm is a single source shortest path algorithm. It is used to find the shortest path from a single source node to all other nodes on a directed graph with non-negative edges cost only. Dijkstra's algorithm works on a static network (where the edge weights on the network are static and deterministic). Similar to the BFS strategy, it works by examining the closest node to the start node. However, Dijkstra's algorithm uses a weighted graph and a priority queue to decide which is the most promising node (the node's link that has the minimum weight) to be expanded first.

A priority queue is an abstract data type (similar to the general queue and stack data structures). However, in a priority queue, each node has a "priority" associated with it. A node with high priority is examined before a node with a low priority. If two nodes have the same priority, they are examined based on their order in the queue.

Dijkstra's algorithm uses two data structures during the search to manage node expansions: priority queue (**Q**) and an array (**A**), which keeps the record of the total distance from the source location ($\boldsymbol{s}$) to all other nodes in the graph. When Dijkstra's algorithm starts the search, the g($value$) of the source node will be 0, (the distance from the source node itself), $\boldsymbol{A}\ (\boldsymbol{s}) = \mathbf{0}$. For all other nodes $\boldsymbol{N}$, is set to infinity $\boldsymbol{A}\ (\boldsymbol{N}) = \infty$. The priority queue is constricted, which contains all of the nodes of the graph. The loop is executed when there are no nodes in the priority queue and the goal node has not been selected for expansion. In each iteration, the node with the minimum priority (smallest distance $A\ (N)$) is removed from the priority queue for expansion. Dijkstra algorithm used the following formula to check the node.

If the distance from the source node itself *A(S)* + the cost link to travel from *S* to N is less than the distance label for N.

$$A(S) + A(S, N) < A(N) \qquad (3.1)$$

In this case, a different path from the source node (S) to the current node (N) has been found. The distance of this node will be updated,

$A(N) = A(S)$ +(cost of link to travel from $S\ to\ N$). The algorithm will repeat the process until there are no nodes in the Q or the destination node is picked from the Q.

Figure 3.4 shows the pseudo-code of Dijkstra's algorithm. In the pseudo-code, **G** is the input graph, **S** is the source location, **Q** is the priority queue and **A** is the array.

```
{1} Dijkstra (G, S)
{2} A(S) = 0
{3} Q = G (N)
{4} While (Q ! = ∅)
```

{5} Do U= extraxtMin (Q)

{6} For each link U to N outgoing from U

{7} If (A (U) + cost of link < A (N))

{8} A (N) = A (U) +cost of link

{9} DecreasePriority (Q, N)

Figure 3. 4 Dijkstra's algorithm.

Analysis of Space Complexity

Since array **A** has been adopted, the space complexity is $O(E)$, where T is the edge number of the directed graph. In the worst case, if $E = n^2$, the space complexity is $O\ (n^2)$.

Analysis of Time Complexity

Dijkstra algorithm is shown in Figure 3.4. The algorithm first calls A(S) the time complexity of $\{2\}$ is $O(n)$ and the time complexity of $\{3\}$ is (n) ;

Then, the time complexity is $O\ (n^2)$. For $\{4\}$, the first cycle number is node n .The second cycle number obtained is $n-1$ In each iteration, the node with the minimum priority smallest distance $A\ (N)$) is removed from the priority queue for expansion. Then the time complexity is $(n\ -1)\ +\ (n\ -\ 2)\ +\cdots+1$, that is $O\ (n^2)$.

In the road network of a city, the nodes are uniformly distributed in the plane and the nearby nodes are connected by edges. In this case, the number of visited nodes grows with the square of *O (n).* The computation time therefore takes a long time and becomes slower.

3.3.1.1 K-Shortest Path Algorithm

To improve the effectiveness of travel information, there is a need to generate alternative routes for users of public transportation. For example, when the shortest path between the source node

and the destination node is congested, it is necessary to compute the second shortest path. If the shortest path is not available, then a third shortest path may be needed. This set of alternative paths is known as the set of K-Shortest Paths (KSP) (Meena and Geethanjali, 2010). There are many papers concerning several algorithms used for solving the KSP (Palmgren and Yuan, 1998) but not many papers deal with the application to real-world problems. The computational complexity of applying the KSP algorithm in a large network remains a big problem in terms of efficiency.

Several algorithms can be used to solve the KSP. Two examples of algorithms that calculate a list of the KSP between nodes in a weighted directed graph are Yen's (1971) algorithm and Lawler's (1972) algorithms. Yen's algorithm is a classical algorithm for finding the KSP between a pair of nodes in a directed graph. Firstly, it used Dijkstra's algorithm as a standard shortest-path algorithm to compute the best path from a source location to a destination location. For example, the following graph, see Figure 3.5 - three KSP, was computed from the source node (A) to the destination node (E).

The best shortest path is: A→B→D→E (the cost time is 11 min).
This is the first shortest path called A1, and it will be stored in list A. After that, Yen's algorithm takes every node in the computed path except for the destination node and computes another shortest path (called a spur) from each selected node to the destination node.

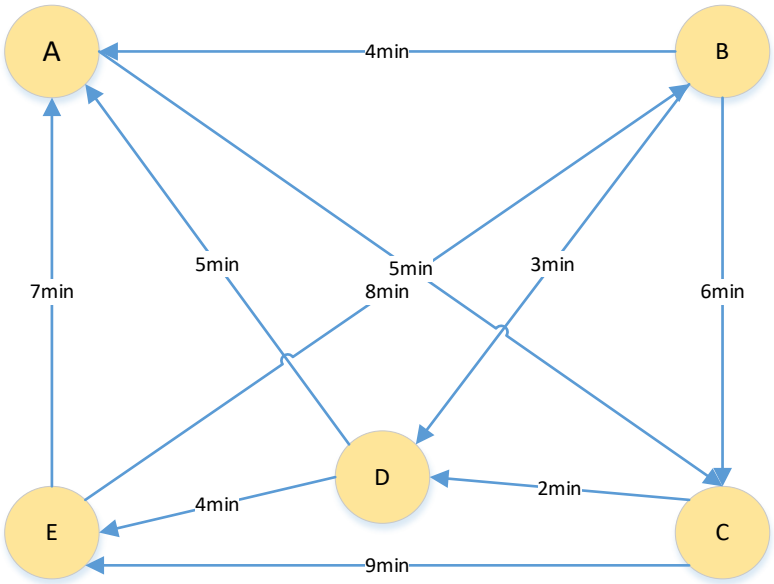


Figure 3. 5 Directed graph.

For such node, the path from the source node to the current node is the root path, which is A→B. Node A of A1 becomes the super node with a root path itself. Two main restrictions come with the super path. Firstly, it must not pass through any node on the root path. Secondly, it must not branch from the current node on the link used by the first shortest path (previously found) with the same root. Therefore, the link A→B will be removed and Dijkstra's algorithm is used to compute another spur path, A→C→D→E (the cost time is 12min).

This is the second shortest path called A2, and it will be stored in the list B. Node B of A2 becomes the spur node with a root path of B→D. However, the same process is repeated. The link B→D will be removed and Dijkstra's algorithm is to compute another spur path which is A→B→C→E (the cost time is 18min). This is the third shortest path called A3; node D of A3 becomes the spur node with a root path D→E. All such paths are stored in list B. Within this third iteration, we will choose from list B the paths which have the lowest cost in order of their shortest time.

The first path is A→B→D→E

The second path is A→C→D→E

The third shortest path is A→B→C→E

Lawler's algorithm is used to find k-shortest paths. Lawler's algorithm presents an adjustment to Yen's algorithm. Therefore, Lawler's algorithm computes only new paths from the nodes that were on the spur path of the previous shortest path. The improved efficiency of Lawler's algorithm can be clarified by, when finding more than two k-shortest paths, the next k-shortest paths will, on average, branch from the middle of the path. Therefore nearly a 50% improvement in speed is achieved over Yen's algorithm (Martins and Pascoal, 2003; Meena and Geethanjali, 2010).

Other types of algorithm are commonly used to solve the shortest path problem. These algorithms are known as labelling algorithms. The algorithms work in such a way that in finding the shortest distance from the source node to every other node in the network, each node (n) has a distance label $(DL[n])$ which represents the shortest distance from the source node to node (n). The algorithms have a list that contains nodes whose distance labels are not the shortest distance. There are two classes; label setting algorithms and label correcting algorithms. Each algorithm has a different way to remove nodes from the list and to find the shortest path. .

3.3.1.2 Label Setting Algorithm

The Label-Setting Algorithm (LS) is used to find the shortest path from the source location to every other node, by remove the node with the smallest distance label from the list during each iteration of the algorithm. Once the node is removed from the list, it will not be inserted again,

and the algorithm stops after n iterations, when the list is empty. This algorithm can be determine and find the shortest path when the label of that destination location is set.

Therefore, LS are very suitable for applications such as route guidance systems where the objective is to find the shortest path between two specific locations (Fu and Rilett, 2005).

The time complexity of the algorithm is based how the list is stored and how the minimum distance is found. The simple algorithm is attributed to Dijkstra's algorithm, which finds the shortest path in $O\ (N^2)$ time for N iterations. Irrespective of the data structures and search algorithms that are used to solve the shortest path problem, using a label sitting algorithm examines all nodes once. Hence the time complexity of the label sitting algorithm is $O\ (N^2)$ (Hribar et al, 1995).

3.3.1.3 Label Correcting Algorithm

Label correcting (LC) algorithms takes a constant amount of time per iteration to compute the paths but vary in the number of repetitions needed to complete the shortest path calculation. In each iteration, the node removed from the list does not essentially have the shortest distance label. Thus, the removed node may be re-entered at later time in the list. The LC cannot provide the shortest path between two nodes before the shortest path to every node in the network is known. This makes the algorithms more suitable in cases when the KSP from the source location needs to be found. Consequently, label-correcting algorithms are usually used in transportation planning applications when multiple routes have to be known (Fu and Rilett, 2005). The time complexity of the algorithm is $O(E\,N)$ times where E is a number of edges and for N iterations (Hribar, et al, 1995).

3.3.2 A* Algorithm

The A* algorithm is normally used to solve an optimal route problem from a start location to a goal location. The evaluation function f is a sum of the two functions:

- The path cost function, which is the cost from the source node to the current node (denoted by g(n)).
- An admissible "heuristic estimate" of distance to the destination (denoted by h(n))

Each node n maintains f(n) where f(n) = g(n)+h(n). The function $f(n) = g(n) + h(n)$ is an estimate of this cost and is used by A* to decide which node should be expanded next.

In comparison, Dijkstra's algorithm visits all nodes in every direction. This results in a large area being explored by expanding unnecessary nodes before the destination is found. However, the A* algorithm focuses the search towards the goal by using a heuristic function.

The heuristic function $h(n)$ has two important properties: it is admissible and consistent.

$h(n)$ is an admissible heuristic if in place for every node n, $h(n) \leq h^*(n)$. Where $h^*(n)$ is the real cost to reach the destination state from n, so, the heuristic function never overestimates the real cost. The other important property is that the heuristic function $h(n)$ is consistent if, for every node n, travelling through any successor n' of n will result in: (where $\boldsymbol{c}$ is the link-cost between node n and n', and $\boldsymbol{D}$ is the destination node)

$$h(n) \leq c(n, n') + h(n').$$

This can be viewed as a kind of triangle inequality as seen in Figure 3.6. Each side of a triangle cannot be larger than the sum of the other two. Every consistent heuristic function is also admissible (Russell and Norvig, 2003).

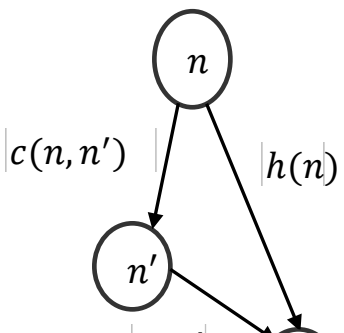


Figure 3. 6 Consistent heuristic function

The A* algorithm uses two data structures to manage node expansion during the search: open list and closed list. The open list is a priority queue that allows A* to always expand the node with the smallest f($value$). Therefore, A* avoids considering directions with non-favourable results and the search direction can efficiently lead to the destination. In this way, computation time is reduced. Thus, the A* algorithm is faster than Dijkstra's algorithm for finding the shortest path between a single pair of nodes. A closed list stores the nodes that have already been expanded. The major benefit of a closed list is that it allows A* to avoid node re-expansion.

By using a heuristic function, A* significantly reduces the number of nodes visited. Finding a solution without the loss of the solution optimality will depend on the type of heuristics used (i.e. an admissible will generate an optimal solution). In the worst-case scenario, the number of nodes visited will be exponential to the length of the optimal solution.

As discussed in Rios and Chaimowicz (2010), when the A* algorithm starts the search, the g($value$) for the source node will be 0. Then the algorithm will insert this node into the open list. The loop is executed while there are no nodes in this list and the destination node has not been selected for expansion from the open list. In each iteration, the node with the smallest f($value$) is removed from the open list for expansion. Its successors are generated and

inserted into the open list. It is possible for one or more of these successors' nodes to be present already in the open list. In this case, a different path from the start node to the current node has been found. If the new cost is smaller, then the g($value$) of the node will be updated, and the expanded node is inserted into the closed list.

There are essentially two termination conditions for the A* algorithm. Firstly, if the open list is empty, then there are no more nodes to be expanded. This means that there is no solution for the problem, as there has been no path found. Secondly, if the destination state is selected for expansion from the open list, then an optimal solution has been found (if the heuristic is admissible and consistent).

3.4 Properties of Search Algorithms

The efficiency of the search algorithm is a critical problem in pathfinding since it relates to the practicality and effectiveness of the search algorithms. The properties of search algorithms involve two aspects: time complexity and memory space requirements. However, advances in computer hardware have made it possible to provide sufficient memory in most computational environments. The main concern is now the time complexity of the algorithm. This is how much time the algorithm requires in relation to the depth of the solution (usually proportional to the number of nodes visited.).

Dijkstra's algorithm does not search directly to the destination. It searches in all directions. This results in the algorithm expanding unnecessary nodes. Based on Figure 3.4, the pseudo-code of Dijkstra's algorithm from step {5} to step {9} takes up the most computation time. In step {5}, the algorithm finds the node that contains the shortest distance which requires $|N|$ times comparison in the first iteration. For the comparison operation, steps from {7} to {9}, all links

that are connected to the current node are examined, which takes |*E*| time. Thus, with a network consisting of N nodes, Dijkstra's algorithm has a computational complexity of $O\ (|N^2| + |E|)$ = $O\ (N^2)$.

The time complexity of the A* algorithm is based on a heuristic function. In this case, it's often more meaningful to measure the running time in terms of the branching factor of the tree (b) and search depth (i.e., the levels traversed in searching the tree. Once the algorithm finds the destination, the algorithm stops) (d) then the time complexity of the A* algorithm is $O(d^b)$ .This assumes that the destination exists. If not, then the algorithm will not terminate (Wu, 2006). Based on the above time complexity comparison, A* is an efficient algorithm to solve the shortest path problem, because in such a tree, if we examine every node at depth < d before we find the destination node, we'll end up visiting $O(d^b)$ nodes before the algorithm stops. Then the algorithm visits a subset of the graph with $|N| = O(d^b)$ (where now N includes only the nodes the algorithm visit). It is clear to see that $O(d^b)$ is more efficient than $O\ (N^2)$.

However, finding the nearest node with the smallest cost is time-consuming in large paths. The A* algorithm is suitable when the nodes change at an infrequent rate. Therefore, re-computing the path at each change is not a serious problem. On the other hand, if used with real-time traffic information, where the link cost can change frequently and randomly, having to re-plan with every modification becomes unacceptable. Therefore, A* is not optimised for pathfinding cases in dynamic traffic conditions, because, A* re-computes the shortest route from scratch. A solution to this problem is to change A* from a re-plan strategy to a reuse strategy, for it to be suitable for dynamic data (Russell and Norvig, 2009).

3.5 Incremental search

3.5.1 Dynamic Traffic Routing

There are many situations where it is important that the path searches are fast. For example, a commuter will spend a lot of time travelling to their destination. These high travel times are due to blocks in links in the route, which results from high traffic flow, incidents, events or road construction.

In Section 2.3, the transportation network is defined for a dynamic transportation network and changes in traffic conditions are considered as changes in the link-costs where the blocked link occurs. Since traffic conditions continually change over time, the centralised navigation service has to monitor the traffic fluctuations over a day-long interval and detect any congestion in order to allow drivers to take preventive action. By using dynamic shortest path algorithms, navigation services can also help a traveller to plan an alternative optimal route to their destination based on the updated traffic conditions. This re-routing feature enables the algorithms to be used in real-time traffic routing software.

In dynamic transportation networks, cost changes can be classified as either deterministic time-dependent or stochastic. In a deterministic time-dependent shortest path (TDSP) problem, the travel times on the edges are functions of time and the network is known as a dynamic (or time-dependent) network.

3.5.2 Dynamic Time-Dependent Shortest Path

It should be noted that some standard shortest path algorithms can be used to compute shortest paths in time-dependent (but not in stochastic) networks (Dreyfus, 1969; Orda and Rom, 1990; Kaufman and Smith, 1993; Ziliaskopoulos and Mahassani, 1993; Chabini, 1997). For example, Sung et al (2000) suggested another model and solution for the TDSP problem, where the travel time in a network depends on the time interval. They proposed a model for time-dependent networks where the flow-speed of each link depends on the time interval, using an algorithm

based on Dijkstra's label-setting algorithm. This algorithm will compute the travel time of each link according to the flow-speed at the time of passing the link. Cook and Halsely (1966) extended Bellman Ford's (1958) principle from finding the single source shortest path in a weighted directed graph to finding the shortest path between any two nodes in a time-dependent network. Dreyfus (1969) proposed that Dijkstra's labelling algorithm can be modified to compute the time-dependent shortest path problem when the edges' weights in the network are not static.

3.5.3 Dynamic Stochastic Shortest Path

Another type of dynamic transportation network is called a stochastic network. Nielsen (2004) noted that if a network follows a probabilistic distribution of travel times for different parts of the network, then the network is called a stochastic network. The link costs in this type of network are random variables (each assigned a probability). This type of network can be used in transportation networks, particularly when disruption to traffic networks has happened. As travel time in transportation networks is unpredictable, this type of network is suitable for transportation networks where one of the links may be blocked at any given time. This stochasticity means that the travel time can be only be estimated.

Much research has been carried out in relation to solving the SSP problem. For example, Frank (1969) computed the SP by replacing all of the links with their expected values and then using standard algorithms to solve the shortest path problem. Mirchandani (1976) developed some methods to compute the expected shortest travel time between two nodes in the network when the travel times were random variables. The most significant sources of randomness were the unpredictable disruptions to traffic networks.

3.5.4 Incremental Dynamic Time-Dependent Shortest Path

Dynamic shortest paths route finding is a fundamental problem in the field of ITS applications. Dynamic routing requires the re-planning of a route based on updated dynamic information becoming available during travel. In this research, dynamic information is referred to as time-dependent deterministic information (not stochastic information where random events become available over time rather than at the time of departure from the node). This research focuses on the Dynamic Shortest Path (DSP) algorithm itself. The DSP algorithm uses current traffic conditions to dynamically maintain the optimal path en route.

With a single cost of a link change, usually, only a small portion of the network is affected. For this reason, it is reasonable to update the portion of the network that is affected by the link-cost change and to avoid computing the shortest path from scratch.

Incremental search methods are used to solve dynamic shortest path problems. It is a search technique for re-planning. The reuse of the previous search is faster than computing the shortest path from scratch. Although incremental search methods are not widely used in artificial intelligence, several incremental search methods have been suggested in the algorithms' literature (Terrovitis, et al, 2005; Spira and Pan, 1975; MarchettiSpaccamela, 1993; Franciosa et al., 2001; Frigioni et al, 1996; Edelkamp, 1998; Al-Ansari, 2001). These algorithms differ in their purpose; some solve single source shortest path problems and others solve all-pairs shortest path problems. They also differ in the performance measure that they use; when they update the optimal route, which kinds of graph topology and edge costs they apply, and how the graph topology and link costs can change over time (Frigioni et al., 1998). If arbitrary sequences of link-cost insertions, deletions, or link-cost changes are allowed then the dynamic

shortest path problem is called a fully dynamic shortest-path problem. It is called a semi-dynamic shortest path problem if only the link-cost increment (or decrement) is allowed (Frigioni et al., 2000). Stentz (1994) presented a dynamic version of A* algorithm called D* algorithm. The algorithm dynamically adapts to random changes in traffic conditions. Stentz, (1995) also introduced a version called Focused D* algorithm, which reduced the computation time by two to three times by reducing the number of node expansions. This was achieved by using heuristics that guided the search direction. If there is no change on the route during the traverse, then the solution is identical to the A* algorithm. The D* and Focused D* algorithms are used when there is incomplete information (e.g., unknown destination). They can find shortest paths from the source node to all other nodes in the graph and are able to quickly recalculate the route.

There are also incremental search methods, such as Ramalingam and Reps' algorithm (Ramalingam, 1996). RR for short, also known as the DynamicSWSF-FP algorithm, is a fully dynamic shortest-path algorithm and reuses information from previous searches to find the shortest paths. When compared to other similar path-planning problems and solutions, the DynamicSWSF-FP algorithm is potentially faster as it does not solve each path-planning problem from scratch. The DynamicSWSF-FP algorithm starts the search from the destination node to all other nodes and thus maintains estimates of the destination distances rather than the start distances. The main advantage of using the DynamicSWSF-FP algorithm to solve the dynamic shortest path problem is that it uses a clever way of identifying the links-costs that have not changed and re-computes only the ones that have changed. Therefore, the DynamicSWSF-FP algorithm performs best in cases where only a small number of link-costs change. More details about the DynamicSWSF-FP algorithm will be discussed in the next section.

3.5.5 Incremental DynamicSWSF-FP Shortest Path Algorithm

In dynamic transportation networks, only a small portion of the network's links' costs change between each update. The cost for some links stay the same as before and thus do not need to be re-calculated. A complete update of the best shortest path can be considered inefficient when dealing with a large network, because the previous information results can be reused. Incremental search methods, such as the DynamicSWSF-FP algorithm (Ramalingam, 1996), reuse information from previous searches to find shortest paths for a series of similar path-planning problems. This use of previous searches means that the incremental search algorithm is faster than using other algorithms (e.g., Dijkstra algorithm and A* algorithm) which solve each path-planning problem from scratch.

The DynamicSWSF-FP algorithm runs the same basic algorithm each time at the cost of any link on the computed shortest path changes. The problem with reusing previous information results is how to detect these changes. The DynamicSWSF-FP algorithm uses a clever way (uses rhs-value for short, also known as the one-step look ahead value) to determine which link is affected and needs to get updated. The rhs-value is equal to the cost from the current node to the parent node plus the cost to travel from the current node to the child node. By comparing this value with the original cost between the parent and child nodes, the algorithm can detect link cost changes.

Suppose that S indicate the finite set of nodes s of the graph and $succ(s) \subseteq S$ indicate the set of successors of node $s \in S$. Similarly, $pred(s) \subseteq S$ denotes the set of predecessors of node s∈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 $s \in S$. (Koenig, Likhachev and Furcy, (2004); Wu (2006))

There are two estimates in the DynamicSWSF-FP lifetime:

- $g(s)$ is the start cost of each node s, which is equal to the g($value$) of Dijkstra's algorithm.
- rhs ($value$) is the one-step look-ahead value based on each g-value and thus is potentially better informed than the g-value (Koenig, Likhachev and Furcy, 2004).

rhs($value$) is computed from the following relationship:

$$rhs(s) = \begin{cases} 0 & if s = s_{start} \\ min_{s' \in pred(s)} \left(g(s') + c(s', s)\right) & \text{otherwise} \end{cases} \quad (3.4.2.1)$$

The DynamicSWSF-FP algorithm uses a priority queue that always contains nodes with a g($value$) that needs to be updated to make them locally consistent. Local consistency checks prevent the expansion of the same node twice (node re-expansion). This is very important for improved the efficiency. Nodes are locally consistent, which means that the g-value of all nodes is equal to their start distances, and if some of the link's costs are updated then the g($value$) of the affected nodes will be changed. These changes will make the nodes become locally inconsistent (Koenig, Likhachev and Furcy, 2004). The DynamicSWSF-FP algorithm uses a priority queue that contains the nodes whose g-value needs to be updated in order to make them locally consistent. Then the shortest path can be updated dynamically.

3.6 Summary

The chapter classifies the common search strategies including uninformed search, informed search, and incremental search. Two classical shortest path algorithms (Dijkstra's and the A* algorithms) are shown to be the typical solution for static environments. The A* algorithm is

most suitable for calculating the shortest path between single pair nodes using a static approach, due to its improved speed. However, the A* algorithm is not optimised for pathfinding cases under dynamic traffic conditions. This is because it has to recalculate the optimal route from scratch each time there is a change in traffic. And so, in order to satisfy the requirement of applications for real-world traffic networks, the DSP problem has been discussed. Some related research on the TDSP and SSHP problems have been briefly introduced in order to identify the research area in this thesis, which assumes that the dynamic information is referred to as time-dependent information. To solve a Dynamic Shortest Path problem to a known destination, an efficient and dynamic algorithm is required to solve the single pair shortest path problem. The dynamic algorithms discussed in this chapter are focused on finding the shortest route from the source location to an unknown destination. The DynamicSWSF-FP algorithm is shown to be the most efficient approach in most dynamic environments, when finding the distance from multiple nodes to the destination node after each change to the path. However, this means that it is not able to deal with the dynamic single pair shortest path problem.

The next chapter presents the first proposed OLPA* algorithm and the experimental results.

CHAPTER 8

CONCLUSION AND FUTURE WORK

8.1. Introduction

This research has presented new efficient algorithms in the context of dynamic route planning. These algorithms are significantly more efficient than any other previous algorithm. Three novel algorithms have been proposed based on the existing Lifelong Planning A* algorithm to solve the optimal route problem in a dynamic environment where the traveller has to re-compute the shortest path while travelling. The experimental results show that the most efficient of our new algorithms outperforms the previous algorithms.

8.1.1 Optimised LPA* Algorithm

This research presents the first novel approach; an optimised version of the LPA* algorithm, making it capable of improving the search performance of the algorithm to solve the dynamic shortest path problem, which is where the traveller may have to re-compute the shortest path while travelling in a dynamic transportation environment. To examine the efficiency of the OLPA* algorithm, the experiments were performed using the real-world road networks of Nottingham city. This algorithm was also implemented in a public transportation network. . This research used 108 bus lines in two directions (inbound/outbound).

To simulate real-time traffic conditions, this research used an accident event that was simulated arbitrarily along the path as a cost of some node changes to adapt to the random changes in traffic conditions.

The experimental results show that OLPA* algorithm has greater efficiency when implemented in both road and bus networks. In the road networks (according to Section 4.7.3.1, the results indicate that the OLPA* algorithm reduces the computation time by up to less than 3 seconds to expand 15000 nodes, while the LPA* algorithm spent 120 seconds to expand 15000 nodes (see Figures 4.20 and 4.21). Both algorithms LPA* and OLPA* examined the same number of nodes in the first search and in the second search. A* expanded 30000 nodes in 15 seconds. The OLPA* algorithm is faster than both the LPA* and the A* algorithms. When using bus networks, the results show that the running time of the OLPA* algorithm ranges under one second with a different number of expanded nodes, while for the LPA* algorithm, the computation time increases with the increase in the number of nodes expanded. Compared with the A* algorithm, the OLPA* algorithm expands less than 1000 nodes in less than 0.1 seconds in the second search compared to the first search (Figure 4.25) of 17000 nodes. It is easy to see the advantage of using the previous calculation. It makes the search faster and is significantly able to reduce the search space. Consequently, the OLPA* algorithm has greater efficiency when implemented in both road and bus networks.

8.1.2 Bi-directional OLPA* Algorithm

This research has proposed a second novel approach that works in the private transport network. The proposed algorithm combines the OLPA* algorithm and Bi-directional approach, named the Bi-directional Lifelong Planning A* algorithm (BiOLPA*). To examine the efficiency of the novelty approaches, the experiments were performed using the real-world

road networks of Nottingham city. The Nottingham road map contains 38344 nodes and 48545 links.

To simulate real-time traffic conditions, this research used an accident event that was simulated arbitrarily along the path as a cost of some node changes. The accident event was used to simulate the situation of a node submitted en-route for a new optimal route in order to adapt to the random changes in traffic conditions. The experiment result showed that, in all cases, the BiOLPA* algorithm is more efficient than the OLPA*, LPA* and A* algorithms. BiOLPA* is significantly superior to the OLPA*, LPA* and A* algorithms, in terms of the number of expansion nodes as well as in terms of computation time. The experiment results indicated that the number of examined nodes in the BiOLPA* algorithm is a significantly reduced number of expiation nodes compared to the OLPA*, LPA* and A* algorithms. In addition, the BiOLPA* algorithm has good performance when it comes to reducing the computation time y up to 0.001 (according to Figure 6.11). Regardless of the location of the accident, if it is close or away from the goal in all cases, the BiOLPA* algorithm is more efficient than the OLPA*, LPA* and A* algorithms. BiOLPA* is significantly faster than the OLPA*, LPA* and A* algorithms, in terms of the number of expansion nodes as well as in terms of computation time.

8.1.3 Bi-directional LPA* algorithm

To evaluate the BiOLPA* algorithm's ability to reduce the number of expansion nodes, this research implemented the Bi-directional LPA* algorithm. It is a new algorithm alternatively named the BLPA* algorithm. The BLPA* algorithm used the cardinality comparison strategy to select the search direction (forward or backwards), rather than simply alternating the directions. The BiOLPA* algorithm used a novel search strategy (called an autonomous strategy) to enhance the intelligence of node selection in the algorithm. The strategy determines

the best selection of either the forward or backward direction. This contributes to speeding up the search process instead of each of the forward and backward searches alternatively. This selection process avoids accessing many unnecessary nodes from both the forward or backward direction, and therefore makes the search for the shortest path converge to the goal node more quickly. The experimental results shows that the BiOLPA* algorithm is more efficient than the proposed BLPA* algorithm in terms of the number of nodes expanded and in terms of the computation time. The optimal solutions obtained by the BiOLPA* and OLPA* algorithms are better than the best solutions obtained by the existing LPA* and A* algorithms.

8.2 Future work

The proposed dynamic shortest-paths algorithms in this thesis have been presented and originally extensively developed as a means for the simplified implementation of the state dynamic algorithm in context of Nottingham's urban network. Further development should be addressed to improve the obtained results.

8.2.1 Data Source

There is a developing need for the improvement of the efficiency of urban traffic data to ensure the sustainability of modern cities. To simulate real-time traffic conditions, future work will include the SCOOT system to obtain real time traffic data. The idea of using the SCOOT data is to know the time, location and duration of a blocked link. Future work will include real time bus information. Using real time bus information will allow us to obtain a more accurate prediction for travel time, better suggestions for an alternative route and therefore, a better suggestion for starting the search from a backward search.

8.2.2 Travel Time Prediction Method

Travel time indicates the time spent travelling along a path. Travel time data is very important to improve the performance of the proposed algorithms in this research. It is quite important for the route guidance system and for the traveller to make a quick decisions to save their time. Based on accurate travel time predictions, travellers can change their departure time and choose the path that has the optimal expected arrival time. As to the in-vehicle route guidance system, it can benefit from accurate travel time prediction technologies, provide alternative route for travellers and avoid potential congestion areas. It is better to evaluate the total performance of the dynamic algorithms.

The future work will consider using the travel time prediction method, which provides the traveller with useful information to help them make a quick decision and to save their time by changing their departure time, to change their route or to cancel the trip. Predicting travel time accurately is done by considering both the traffic condition of a time range in a day and the traffic patterns of vehicle in a week.

8.2.3 Stochastic Link Travel Time

It would be interesting to apply the proposed algorithms on both a dynamic and stochastic link. That way, the road travel time is not only dynamic, but also contains an amount of uncertainty, with the edge's cost in the network being random variables with a probability. This type of network is useful in transportation networks, particularly when the disruption to traffic networks has happened. Therefore, travel time in transportation networks is unpredictable and this type of network is suitable for transportation networks where one of the routes can be blocked at any given time. The travel time can only be estimated. In future work, the dynamic

information is referred to as stochastic information, where random events become available over time rather than at the time of departure from the node.

REFERENCES

Abraham, I., Delling, D., Goldberg, A. and Werneck, R. (2011). A Hub-based Labelling Algorithm for Shortest Paths in Road Networks. In Pardalos, P.M. and Rebennack, S. (eds.), *Proceedings of the 10th International Symposium on Experimental Algorithms*, (SEA'11), Kolimpari, Greece, May 5-7, 2011. Berlin, Heidelberg: Springer-Verlag, pp. 230–241.

Akiba, T., Iwata, Y., Kawarabayashi, K. and Kawata, Y. (2014). Fast Shortest-path Distance Queries on Road Networks by Pruned Highway Labelling. In McGeoch, C.C. and Meyer, U. (eds.), *2014 Proceedings of the 16th Workshop on Algorithm Engineering and Experiments*, (ALENEX). Portland, Oregon, Jan 5, 2014. Philadelphia: Society for Industrial and Applied Mathematics, pp.147– 154.

Ausiello, G., Italiano, G., Marchetti Spaccamela, A. and Nanni, U. (1991). Incremental Algorithms for minimal length paths. *Journal of Algorithms*, 12(4), pp.615-638.

Bast, H., Delling, D., Goldberg, A., Muller-Hahnemann,M., Pajor, T., Sanders, P.,Wagner, D. and Werneck , R .(2014) Route Planning in Transportation Networks. *Technical Report MSRTR-2014-4*, Microsoft Research.

Bast, H., Funke, S., Sanders, P. and Schultes, D. (2007). Fast Routing in Road Networks With Transit Nodes. *Science*, 316(5824), p.566.

Bast, H. (2009). Car or Public Transport – Two Worlds. In Albers, S., Alt, H. and Näher, S. (eds), *Efficient Algorithms, Lecture Notes in Computer Science*, 5760. Berlin, Heidelberg: Springer, pp. 355–367.

Beazley, D. (2015). *Python Cookbook*, 3rd ed. Cambridge, Mass.: The MIT Press

Bellman, R. (1958). On a routing problem. *Quarterly of Applied Mathematics*, 16(1), pp.87-90.

Bertsekas, D. (1991). *Linear Network Optimization*. Cambridge, Mass.: The MIT Press.

Betsy, G. and Sangho, K. (2013). *Spatio-temporal Networks Modelling and Algorithms* . London Library of Congress Control Number.

Chaudhuri, S. and Zaroliagis, C. (2000). Shortest Paths in Digraphs of Small Treewidth. Part I: Sequential Algorithms. *Algorithmica*, 27(3), pp.212-226.

Chen, H. and Chang, M. (2000). Dynamic user-optimal departure time/route choice problem with time-window. *Journal of the Chinese Institute of Engineers*, 23(1), pp.71-81.

Cherkassky, B., Goldberg, A. and Radzik, T. (1996). Shortest paths algorithms: Theory and experimental evaluation. *Mathematical Programming*, 73(2), pp.129-174.

Chondrogiannis, L., Panagiotis, B., Johann, G., and Ulf, L. (2015) Alternative Routing: K-Shortest Paths with Limited Overlap. Faculty of Computer Science Free University of Bozen-Bolzano, Italy.

Cohen,E., Halperin, E., Kaplan, H. and Zwick, U. (2002). Reachability and Distance Queries via 2-hop Labels. In *Proceedings of the 13th ACM-SIAM Symposium on Discrete*

Algorithms, *SODA'02*, pp. Jan 6-10, 2002.

Cooke, K. and Halsey, E. (1966). The shortest route through a network with time-dependent internodal transit times. *Journal of Mathematical Analysis and Applications*, 14(3), pp.493-498.

Cormen, T., Leiserson, C., Rivest, R. and Stein, C. (2014). *Introduction to algorithms.* Cambridge, Mass.: The MIT Press.

Daniel, D. and Wagner, D. (2009). Time-Dependent Route Planning. In Ahuja, R.K., Möhring, R.H. and Zaroliagis, C. (eds.), *Robust and Online Large-Scale Optimization, Lecture Notes in Computer Science*, Berlin, Heidelberg: Springer, pp. 207–230.

Denardo, E. and Fox, B. (1979). Shortest-Route Methods: 1. Reaching, Pruning, and Buckets. *Operations Research*, 27(1), pp.161-186.

Deo, N. and Pang, C. (1984). Shortest-path algorithms: Taxonomy and annotation. *Networks*, 14(2), pp.275-323.

Diestel, R. (2005). *Graph Theory*. 3rd ed. Berlin, New York: Springer-Verlag .

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

Ding, J.W., Wang, C.F., Meng, F.H. and Wu, T.Y. (2010) Real-time vehicle route guidance using vehicle-to-vehicle communication. *IET Communications* 4(7), pp. 870-883

Dreyfus, S. (1969). An Appraisal of Some Shortest-Path Algorithms. *Operations Research*, 17(3), pp.395-412.

Driving Directions and Maps (2014) Google Map [online] Available from: https://www.drivingdirectionsandmaps.com/ [Accessed 24 May 2015].

Fan, Y., Kalaba, R. and Moore, J. (2005). Shortest paths in stochastic networks with correlated link costs. *Computers & Mathematics with Applications*, 49(9-10), pp.1549-1564.

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

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

Frank, H. (1969). Shortest Paths in Probabilistic Graphs. *Operations Research*, 17(4), pp.583-599.

Frigioni, D., Marchetti-Spaccamela, A. and Nanni, U. (1998). Semidynamic Algorithms for Maintaining Single-Source Shortest Path Trees. *Algorithmic*, 22(3), pp.250-274.

Frigioni, D., Marchetti-Spaccamela, A. and Nanni, U. (2000). Fully Dynamic Algorithms for Maintaining Shortest Paths Trees. *Journal of Algorithms*, 34(2), pp.251-281.

Fu, L., Sun, D. and Rilett, L. (2006). Heuristic shortest path algorithms for transportation applications: State of the art. *Computers & Operations Research*, 33(11), pp.3324-3343.

George, B. and Kim, M. (2013). *Representation Graphs* [online] Available from: https://www.khanacademy.org/computing/computer-science/algorithms/graph-representation/a/representing-graphs [Accessed 24 May 2015].

George, B. (1962). *Linear Programming and Extensions*. Princeton University Press.

Geisberger, R., Sanders, P., Schultes, D. and Delling, D. (2008) Contraction Hierarchies : Faster and Simpler Hierarchical Routing in Road Networks. In *Proceedings of the 7th International Workshop on Experimental Algorithms*, (WEA'08), Provincetown, MA, May 30 - June 01, 2008. Berlin, Heidelberg: Springer-Verlag, pp.319–333.

Giacomo, N., Daniel, D., Leo, L. and Dominik, S. (2011) *Bidirectional A∗ Search for Time-Dependent Fast Paths*. France.

Giorgio, G. and Stefano, P, (1984). Shortest path methods in transportation methods. In Florian, M (ed.) *Transportation Planning Models*. Elsevier Science Publishers B.V.

Giorgio, G. and Stefano, P, (1986). Shortest path methods: A unifying approach. *Mathematical Programming Study*, 26, p. 38{64}.

Golden, B. (1976). Technical Note—Shortest-Path Algorithms:A Comparison. *Operations Research*, 24(6), pp.1164-1168.

Goodwin, P. (2004). *The Economic Cost of Road Traffic Congestion*. London: Rail Freight Group.

Google Maps™ (n.d.). *Driving Directions and Maps* [online] Available from: www.drivingdirectionsandmaps.com/ and www.Drivingdirectionsandmaps.com [Accessed 15 May 2018].

Google Maps. (n.d.). Transit – [online] Available from https://www.google.com/transit [Accessed 12 Feb 2018].

Hall, R. (1986). The Fastest Path through a Network with Random Time-Dependent Travel Times. *Transportation Science*, 20(3), pp.182-188.

Hart, P., Nilsson, N. and Raphael, B. (1968). A Formal Basis for the Heuristic

Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2), pp.100-107

Hribar, M and Taylor, V. (1995). Choosing a shortest Path Algorithm [online] Available from:http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.22.1097 [Accessed 24 May 2013].

Harrie, L. (2010). *Lecture Notes in GIS Algorithms*. GIS Centre and Department of Earth and Ecosystem Sciences, Lund University.

Huang, Y. W., Jing, N. and Rundensteiner, E.A. (1995*) Route guidance support in intelligent transportation systems: an encoded path view approach*. University of Michigan Technical Report

Ikeda, T., Hsu, M.-Y., Imai, H., Nishimura, S., Shimoura, H., Hashimoto, T., Tenmoku, K. and Mitoh, K. (1994). A fast algorithm for finding better routes by AI search techniques. In Proceedings of VNIS'94 -Vehicle Navigation and Information Systems Conference, Yokohama, Japan, 31 Aug.-2 Sept. 1994. IEEE. pp, 291–296

Jacob, R., Marathe, M. and Nagel, K. (1999). A computational study of routing algorithms for realistic transportation networks. *Journal of Experimental Algorithmic*, 4, p.6.

Jamali, M. (n.d). *Learning to Solve Stochastic Shortest Path Problems*. [online] Available from http://www.cs.sfu.ca/~sja25/personal/resources/SSP.pdf [Accessed 26 August 2013].

James, B.H. and Kwa, B.S. (1989). An admissible bidirectional staged heuristic search algorithm. *Artificial Intelligence*, 38, pp. 95-109.

Karaş, I.R. and Atila, U. (2011) A Genetic Algorithm Approach for Finding the Shortest Driving Time on Mobile Devices. *Scientific Research and Essays*, 6(2), pp. 394-405.

Kaufman, D. and Smith, R. (1993). Fastest Paths in Time-Dependent Networks for Intelligent Vehicle-Highway Systems Application. *I V H S Journal*, 1(1), pp.1-11.

Keyur, R. and Mukesh, Z. (2011). *A- star Algorithm for Energy Efficient Routing in Wireless Sensor Network*. Department of computer Engineering.

Klunder, G. and Post, H. (2006). The shortest path problem on large-scale real-road networks. *Networks*, 48(4), pp.182-194.

Koenig, S. (2001). *Incremental A**. Los Angeles: Computer Science Department, USC.

Koenig, S., Likhachev, M. and Furcy, D. (2004). Lifelong Planning A∗. *Artificial Intelligence*, 155(1-2), pp.93-146.

Kotusevski, G. and Hawick, K. (2009). *A Review of Traffic Simulation Software*. Aukland, New Zealand: Institute of Information and Mathematical Sciences, Massey University.

Kwa, J. (1989). BS∗: An admissible bidirectional staged heuristic search Algorithm. *Artificial Intelligence*, 38(1), pp.95-109.

Lanning, D., Harrell, G. and Wang, J. (2014). *Dijkstra's Algorithm and Google Maps.*Valdosta, Georgia: Department of Math and CS Valdosta State University.

Lawler, E. (2001). *Combinatorial Optimization Networks and Matroids*. United States of America: Dover.

Lester, P. (2005). A* *Path finding for Beginners*. United States of America.

Lucotte, M. and Sang Nguyen (eds.). (1998). E*quilibrium and Advanced Transportation Modelling.* Kluwer Academic Publishers Group.

Loui, R. (1983). Optimal paths in graphs with stochastic or multidimensional weights. *Communications of the ACM*, 26(9), pp.670-676.

Magzhan, K. and Jani, H.M. (2013). A Review and Evaluations of Shortest Path Algorithms. *International Journal of Scientific & Technology Research*, 2(6), pp. 99–104.

Martins, E. and Pascoal, M. (2003). A new implementation of Yens ranking loop less paths algorithm. *Quarterly Journal of the Belgian, French and Italian Operations Research Societies*, 1(2).

McDermott, J. (1980). Principles of artificial intelligence. *Artificial Intelligence*, 15(1-2).

Meena, S, and Geethanjali, N. (2010) A Survey on Shortest Path Routing Algorithms for Public Transport Travel. *Global Journal of Computer Science and Technology*, 9(75).

Mirchandani, P. (1976). Shortest distance and reliability of probabilistic networks. *Computers & Operations Research*, 3(4), pp.347-355.

Mirchandani, P. and Soroush, H. (1985). Optimal paths in probabilistic networks: A case with temporary preferences. *Computers & Operations Research*, 12(4), pp.365-381.

Mohr, A. (2007). Quantum Computing in Complexity Theory and Theory of Computation' [online] Available from: ttps://pdfs.semanticscholar.org/pdf . [Accessed: 7 June 2014].

Murthy, I. and Sarkar, S. (1996). A Relaxation-Based Pruning Technique for a Class of Stochastic Shortest Path Problems. *Transportation Science*, 30(3), pp.220-236.

Murthy, I. and Sarkar, S. (1997). Exact algorithms for the stochastic shortest path Problem with a decreasing deadline utility function. *European Journal of Operational Research,* 103(1), pp.209-229

Negnevitsky, M. (2011). *Artificial Intelligence: A Guide to Intelligent Systems*.3rd ed. Addison Wesley.

Nielsen, R. L. (2004). *Route Choice in Stochastic Time-Dependent Networks*. PhD Thesis, Department of Operations Research University of ARHUS.

Nilsson, N. (1980). *Principles of Artificial Intelligence*. San Francisco: Morgan Kaufmann.

Nottingham City Council (2016) Open Data Nottingham [online] Available from (http://www.opendatanottingham.org.uk/dataset.aspx?id=2.) [Accessed 11 May 2016]

Nordbeck, S. and Bengt, R. (1969). Computer Cartography Shortest Route Programs. Sweden: The Royal University of Lund.

Oliveir, R.L. and Chaimowicz, L. (2010). *A Parallel Bidirectional Heuristic Search Algorithm*. Brazil: Universidade Federal de Minas Gerais (UFMG).

Orda, A. and Rom, R. (1990). Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length. *Journal of the ACM*, 37(3), pp.607-625.

Padua, D. (ed.) (2011). *Encyclopaedia of Parallel Computing*, University of Illinois at Urbaba-Champaign, USA: Springer.

Pape, U. (1974). Implementation and efficiency of Moore-algorithms for the shortest route problem. *Mathematical Programming*, 7(1), pp.212-222.

Pijls, W. and Post, H. (2009a). A new bidirectional search algorithm with shortened postprocessing. *European Journal of Operational Research*, 198(2), pp.363-369.

Pijls, W. and Post, H. (2009b). *Yet another bidirectional algorithm for shortest paths*. Technical Report EI 2009-10, Erasmus University Rotterdam, Econometric Institute.

Pohl, I. (1969). *Bi-directional and heuristic search in path problems*. Technical Report 104, SLAC (Stanford Linear Accelerator Center), Stanford, California.

Pohl, I. (1971). Bi-directional Search. In Meltzer, B. and Michie, D. (eds.), *Machine Intelligence*, 6. Edinburgh: Edinburgh University Press, pp. 127-140.

Ramalingam, G. and Reps, T. (1996). An Incremental Algorithm for a Generalization of the Shortest-Path Problem. *Journal of Algorithms*, 21(2), pp.267-305.

Rios, L. and Chaimowicz, L. (2010). A survey and classification of A* based best-first heuristic search algorithms. SBIA'10 Proceedings of the 20th Brazilian conference on Advances in artificial intelligence. São Bernardo do Campo, Brazil, October 23 - 28, 2010. Berlin, Heidelberg: Springer-Verlag, pp. 253-262.

Russell, S. and Norvig, S. (2009). *Artificial Intelligence: A Modern Approach*. 3rd ed. Prentice Hall.

Sanders. P and Schultes. D. (2005). Highway Hierarchies Hasten Exact Shortest Path Queries. In *ESA'05 Proceedings of the 13th annual European conference on Algorithms*. Palma de Mallorca, Spain, October 03 - 06, 2005. Berlin, Heidelberg: Springer-Verlag, pp. 568-579.

Sedgewick, R. and Wayne, K. (2011). *Algorithms*. 4th ed. Addison Wesley.

Sigal, C., Pritsker, A. and Solberg, J. (1980). The Stochastic Shortest Route Problem. *Operations Research*, 28(5), pp.1122-1129.

Sloot, P. and Abramson, D. (2003). *Computational science-- ICCS 2003.* Berlin, Heidelberg: Springer.

Sommer, C. (2014). Shortest-path Queries in Static Networks. *ACM Computing Surveys*, 46(4), pp.1–31.

Spira, P. and Pan, A. (1975). On finding and updating spanning trees and shortest paths, *SIAM Journal on Computing* 4, pp. 375–380.

Stentz, A. (1994). The D* Algorithm for real-time planning of optimal traverses, Tech Report CMU-RI-TR-94-37, Robotics Institute, Carnegie Mellon University, October 1994 http://www.ri.cmu.edu/pubs/pub_356.html

Stentz, A. (1995). The Focussed D* Algorithm for real-time planning of optimal traverses. In *IJCAI'95 Proceedings of the 14th international joint conference on Artificial intelligence - Volume 2*. Montreal, Quebec, Canada, August 20 - 25, 1995. San Francisco: Morgan Kaufmann, pp. 1652-1659.

Sung, K., Bell, M., Seong, M. and Park, S. (2000). Shortest paths in a network with time-dependent flow speeds. *European Journal of Operational Research*, 121(1), pp. 32-39.

Sven, K, Likhachev, M, Liu, Y. and Furcy, D. (2005). *Life Planning A**. Los Angeles: Computer Science Department, USC.

Tang, F. (2013). *Analysis of Algorithms* [online] Available from **https://www.cpp.edu/~ftang/courses/CS240/lectures/analysis.htm** [Accessed 11 May 2018]

Terrovitis, M., Bakiras, S., Papadias, D. and Mouratidis, K. (2005). Constrained Shortest Path Computation. *Lecture Notes in Computer Science*, 3633, pp. 181-199.

VI Tran, N., Nha, V., Djahel, S. and Murphy, J.(2012). *A Comparative Study of Vehicles Routing for Route Planning in Smart Cities.* Ireland: School of Computer Science and Informatics.

Wegener, A., Piorkowski, M., Raya, M., Hellbruck, H., Fischer, S. and Hubaux, J.-P. (2008). TraCI. In *CNS '08 Proceedings of the 11th communications and networking simulation symposium*. Ottawa, Canada, April 14 - 17, 2008. New York: ACM Press, pp. 155-163

Weihong, C. (1995). *Research on Spatial Data Structure*. China: Science and Technology Press

Whangbo, T. K. (2007). Efficient Modified Bidirectional A* Algorithm for Optimal Route-Finding. In *IEA/AIE'07 Proceedings of the 20th international conference on Industrial, engineering, and other applications of applied intelligent systems*. Kyoto, Japan, June 26 - 29, 2007. Berlin, Heidelberg: Springer-Verlag, pp. 344-353.

Williams, S.G.D . (2008). *Using the A-Start Path Finding Algorithm for Solving General and Constrained Inverse Kinematics Problem.*

Wu, L., Xiao, X., Deng, D., Cong. G., Zhu, A. and Zhou, S. (2012). Shortest Path and Distance Queries on Road Networks: An Experimental Evaluation. *Proceedings of the VLDB Endowment*, 5(5), pp. 406–417.

Wu, Q. (2006). *Incremental Routing Algorithms for Dynamic Transportation Networks*. MsC Thesis, Calgary University.

Xi, C., Qi, F. and Wei, L. (2006). *A new Shortest Path Algorithm based on Heuristic Strategy*. Huazhong University of Science and Technology.

Xingxing, C. and Weihong, C. (1996). An Experimental Study of Rapid Reacting System of City. *Remote Sensing of Environment China,* 11(3), pp. 227-233.

Yen, J. (1971). Finding the Shortest Loop less Paths in a Network. *Management Science,* 17(11), pp.712-716.

Zhan, F. and Noon, C. (1998). Shortest Path Algorithms: An Evaluation Using Real Road Networks. *Transportation Science,* 32(1), pp.65-73.

Zhan, F. and Noon, C. (1997). Three faster shortest path algorithms on real mad networks: data structures and procedures, *Journal of Geographic Information and Decision Analysis,* 1(1), pp. 69-82.

Zhu, A., Ma, H., Xiao, X., Luo, S., Tang, Y. and Zhou, S. (2013). Shortest Path and Distance Queries on Road Networks: Towards Bridging Theory and Practice *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Dat.*
New York, USA, June 22 - 27, 2013. New York: ACM, pp. 857-868

APPENDIX

APPENDIX 1

Using Label setting algorithm to find K-Shortest Path Algorithm

To improve the effectiveness of travel information, there is a need to generate alternative routes for users of public transportation. For example, when the shortest path between the source nodes to the destination node is blocked, because there is a traffic jam or an incident, it is necessary to compute the second shortest path. If this shortest path is not available for some reason, a third shortest path may be needed. This set of alternative paths is known as the set of k-shortest paths (KSP) (Meena and Geethanjali, 2010).

The project starts to find K-Shortest Paths using bus timetable with two different networks, in a static network and time-dependent network. Two implementations have been tested based on Label Setting shortest path algorithm to produce the best possible performance. The suitability of each method was judged in term of minimum run-time and the quality of the obtainable solution

The overall approach and objective of this section are to evaluate two k-shortest paths (KSP) implementations. Two methods have been tested under Microsoft Visual C++ 2012 environment and based on k-shortest paths algorithm using a MySQL database, and standard template library to implement the algorithm. The first method used the k-shortest path algorithm on a static directed graph to compute the optimal route and used the real timetable data to further optimize the route. The second method used the k-shortest path algorithm directly on the timetable data. For a static network, a directed graph was stored in a text file, representing a real road network. For a time-dependent network, the public transportation network was stored in a database. The database contains bus timetables based on the real word transportation in Nottingham city.

In term of performance two methods are compared by used a batch mode command line executable, which executed several thousand times with a unique command line in each case

Network Model

A model describes how to create a graph from the timetables such that we can answer queries in this graph by shortest path computations. Both the road network and public transportation can be modelled as directed graphs. For road networks, each node corresponds to the bus-stop location, where two or more road links meet, and the edges of the graph correspond to the route links. The cost of the link is presented as travel time. The shortest route in the graph corresponds to the quickest path to get from source to destination.

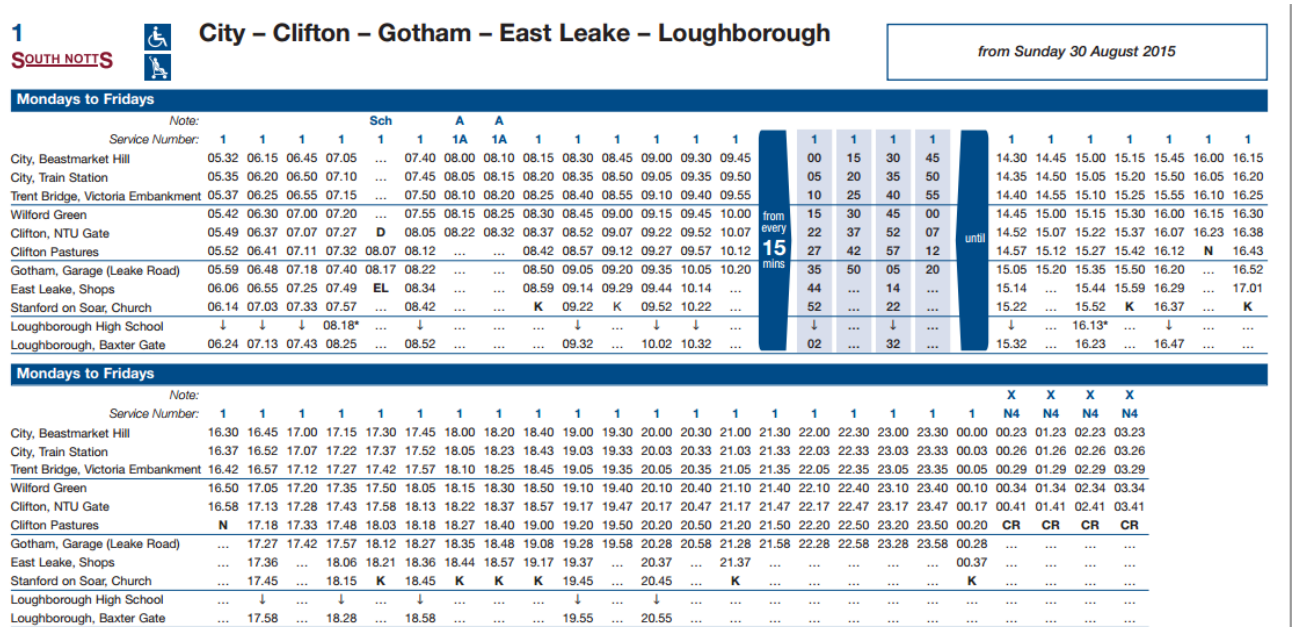
However, there are two ways to model public transportation networks. It can be modelled as a *time-expanded model*, where each node corresponds to a particular time event (arrival or departure), and each link has a constant travel time. To allow transfers with waiting, additional

transfer nodes are added. By adding transfer links from arrival nodes only to transfer link after a minimum transfer duration passed, realistic transfers are ensured. The advantages of this model are its simplicity, as all links cost are a simple value, and Dijkstra algorithm can be used to compute shortest paths (Patrice Marcotte and Sang Nguyen, 1998 and Frank, Wagner, and Karsten Weihe, 2000, Matthias Müller–Hannemann and Karsten Weihe, 2001). Also, public transportation networks can be modelled by *time-dependent model.* The time-dependent model reduces the number of nodes in comparison to the *time –expanded modal*, which showed to be efficient. In *the time-dependent model,* the station graph model uses one node per bus-stop, where *time- expanded model* models multi nodes at each bus-stop (Ariel Orda and Raphael Rom 1990, and Ariel Orda and Raphael Rom 1991, and Karl Nachtigall, 1995 and Gerth Brodal and Riko Jacob, 2004).

In this research, Public transportation networks are modelled in a similar way to road networks, except that we have to deal with scheduled bus timetables. And considers the *time-dependent model* that stations graph model uses exactly one node per bus-stop, to avoid parallel edges which cause more complex edges weights. Each bus-stop except the last one, the timetable includes a departure time, and for each station, except the source location, the timetable includes an arrival time.

Modelling of Bus Timetables

To represent a bus transportation network, bus timetables need to be modelled.



1 SOUTH NOTTS

City – Clifton – Gotham – East Leake – Loughborough

from Sunday 30 August 2015

Mondays to Fridays

Note:					Sch		A	A																			
Service Number:	1	1	1	1	1	1	1A	1A	1	1	1	1	1	1		1	1	1	1		1	1	1	1	1	1	1
City, Beastmarket Hill	05.32	06.15	06.45	07.05	...	07.40	08.00	08.10	08.15	08.30	08.45	09.00	09.30	09.45		00	15	30	45		14.30	14.45	15.00	15.15	15.45	16.00	16.15
City, Train Station	05.35	06.20	06.50	07.10	...	07.45	08.05	08.15	08.20	08.35	08.50	09.05	09.35	09.50		05	20	35	50		14.35	14.50	15.05	15.20	15.50	16.05	16.20
Trent Bridge, Victoria Embankment	05.37	06.25	06.55	07.15	...	07.50	08.10	08.20	08.25	08.40	08.55	09.10	09.40	09.55		10	25	40	55		14.40	14.55	15.10	15.25	15.55	16.10	16.25
Wilford Green	05.42	06.30	07.00	07.20	...	07.55	08.15	08.25	08.30	08.45	09.00	09.15	09.45	10.00	from	15	30	45	00		14.45	15.00	15.15	15.30	16.00	16.15	16.30
Clifton, NTU Gate	05.49	06.37	07.07	07.27	D	08.05	08.22	08.32	08.37	08.52	09.07	09.22	09.52	10.07	every	22	37	52	07	until	14.52	15.07	15.22	15.37	16.07	16.23	16.38
Clifton Pastures	05.52	06.41	07.11	07.32	08.07	08.12	...	...	08.42	08.57	09.12	09.27	09.57	10.12	15	27	42	57	12		14.57	15.12	15.27	15.42	16.12	N	16.43
Gotham, Garage (Leake Road)	05.59	06.48	07.18	07.40	08.17	08.22	...	...	08.50	09.05	09.20	09.35	10.05	10.20	mins	35	50	05	20		15.05	15.20	15.35	15.50	16.20	...	16.52
East Leake, Shops	06.06	06.55	07.25	07.49	EL	08.34	...	...	08.59	09.14	09.29	09.44	10.14	...		44	...	14	...		15.14	...	15.44	15.59	16.29	...	17.01
Stanford on Soar, Church	06.14	07.03	07.33	07.57	...	08.42	...	...	K	09.22	K	09.52	10.22	...		52	...	22	...		15.22	...	15.52	K	16.37	...	K
Loughborough High School	↓	↓	↓	08.18*	...	↓	...	...	...	↓	...	↓	↓	...		↓	...	↓	...		↓	...	16.13*	...	↓	...	...
Loughborough, Baxter Gate	06.24	07.13	07.43	08.25	...	08.52	...	...	...	09.32	...	10.02	10.32	...		02	...	32	...		15.32	...	16.23	...	16.47	...	...

Mondays to Fridays

Note:																					X	X	X	X
Service Number:	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	N4	N4	N4	N4
City, Beastmarket Hill	16.30	16.45	17.00	17.15	17.30	17.45	18.00	18.20	18.40	19.00	19.30	20.00	20.30	21.00	21.30	22.00	22.30	23.00	23.30	00.00	00.23	01.23	02.23	03.23
City, Train Station	16.37	16.52	17.07	17.22	17.37	17.52	18.05	18.23	18.43	19.03	19.33	20.03	20.33	21.03	21.33	22.03	22.33	23.03	23.33	00.03	00.26	01.26	02.26	03.26
Trent Bridge, Victoria Embankment	16.42	16.57	17.12	17.27	17.42	17.57	18.10	18.25	18.45	19.05	19.35	20.05	20.35	21.05	21.35	22.05	22.35	23.05	23.35	00.05	00.29	01.29	02.29	03.29
Wilford Green	16.50	17.05	17.20	17.35	17.50	18.05	18.15	18.30	18.50	19.10	19.40	20.10	20.40	21.10	21.40	22.10	22.40	23.10	23.40	00.10	00.34	01.34	02.34	03.34
Clifton, NTU Gate	16.58	17.13	17.28	17.43	17.58	18.13	18.22	18.37	18.57	19.17	19.47	20.17	20.47	21.17	21.47	22.17	22.47	23.17	23.47	00.17	00.41	01.41	02.41	03.41
Clifton Pastures	N	17.18	17.33	17.48	18.03	18.18	18.27	18.40	19.00	19.20	19.50	20.20	20.50	21.20	21.50	22.20	22.50	23.20	23.50	00.20	CR	CR	CR	CR
Gotham, Garage (Leake Road)	...	17.27	17.42	17.57	18.12	18.27	18.35	18.48	19.08	19.28	19.58	20.28	20.58	21.28	21.58	22.28	22.58	23.28	23.58	00.28	...	...	...	...
East Leake, Shops	...	17.36	...	18.06	18.21	18.36	18.44	18.57	19.17	19.37	...	20.37	...	21.37	...	...	...	...	...	00.37	...	...	...	...
Stanford on Soar, Church	...	17.45	...	18.15	K	18.45	K	K	K	19.45	...	20.45	...	K	...	...	...	...	...	K	...	...	...	...
Loughborough High School	...	↓	...	↓	...	↓	...	...	...	↓	...	↓	...	...	...	...	...	...	...	...	...	...	...	...
Loughborough, Baxter Gate	...	17.58	...	18.28	...	18.58	...	...	...	19.55	...	20.55	...	...	...	...	...	...	...	...	...	...	...	...

Table 1 Timetable of Bus 1 (https://www.nct.co.uk/)

To represent the cost travel time that takes to travel between to stations, an edge is put into the directed graph to represent a connection between two bus stations.

Network Representation

A weighted directed graph *G (N, E)* is presented as a public transportation network, where *N* is a set of nodes and *E* is a set of edges. Each node in *G* corresponds to a certain transport station. In this search, we assume, for simplification, that there is only one kind of public transportation which the bus, so each node corresponds to a bus stop. We also assume that bus stops are represented with numbers from 1 to n. The directed edge $(i, j) \in E$ is an element of the set *E*, also has a cost or length $lij \geq 0$

Figure 1, shows a very simple example of the public transportation presented as a graph, where nodes showed as circles and edges are shown as lines with arrows connected between nodes.

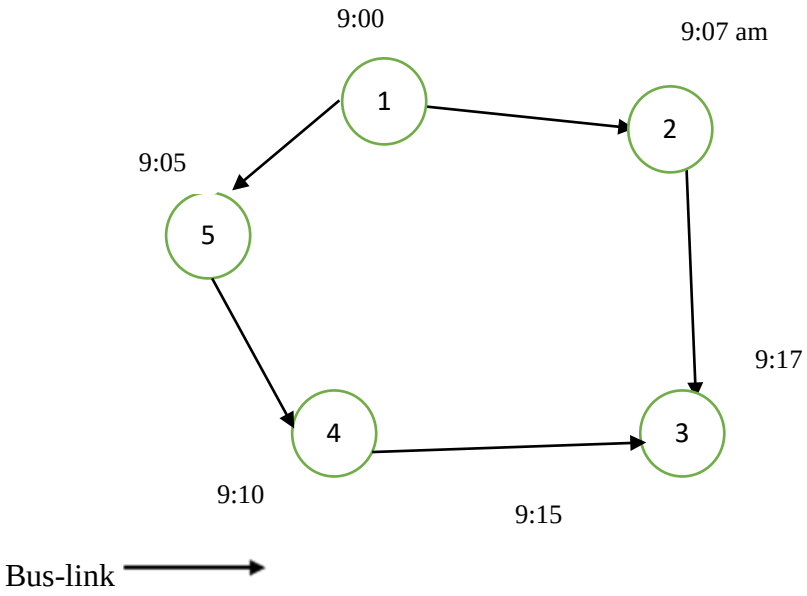
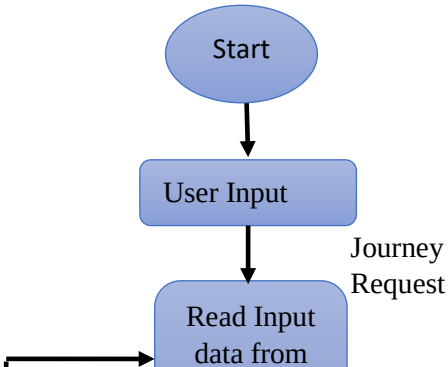


Figure 1: Representation of a simple transportation network

Experimentation

The first implementation has two parts. The first part finds the KSP in static networks. A directed graph stored in a text file was used to represent a real road network. The second part used the real timetable data to further optimize the route. (Figure 2)



The second implementation finds the KSP in a time-dependent (dynamic) network. The public transportation network is stored in a database, which contains bus timetables based on the real world transportation network of Nottingham city. So, this program will find the KSP directly on the timetable data.

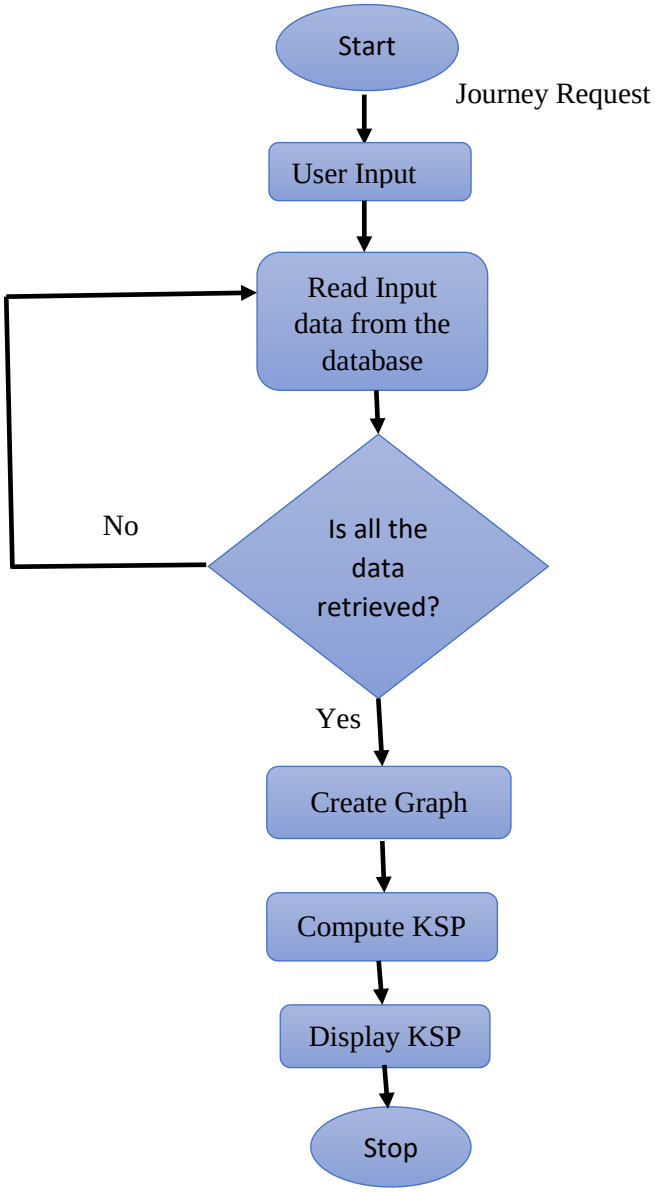


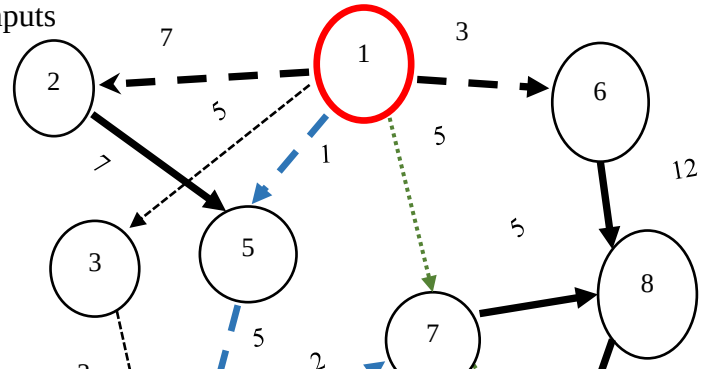
Figure 3: Methodology flow chart of dynamic network implementation

- Journey Request include (Source Location / Destination Location / No of Route / Travel Time)

Evaluation Results

Compute K Shortest Path

Based on the following example inputs



- ✓ Enter Source location : 1
- ✓ Enter Destination location: 10
- ✓ Enter Travel time : 09:48
- ✓ Enter Number of route(s) (K): 3

First Implementation:

KSP for Static and Dynamic Network;

Firstly, the program will display

KSP for Static Network:

1 KSP

2 KSP

3KSP

Figure 4: Directed Graph

And then will read these paths on the bus timetable database and display the result.

1 KSP

2 KSP

3KSP

The second Implementation

KSP for Time-Dependent Network using timetable data directly

1 KSP

2 KSP

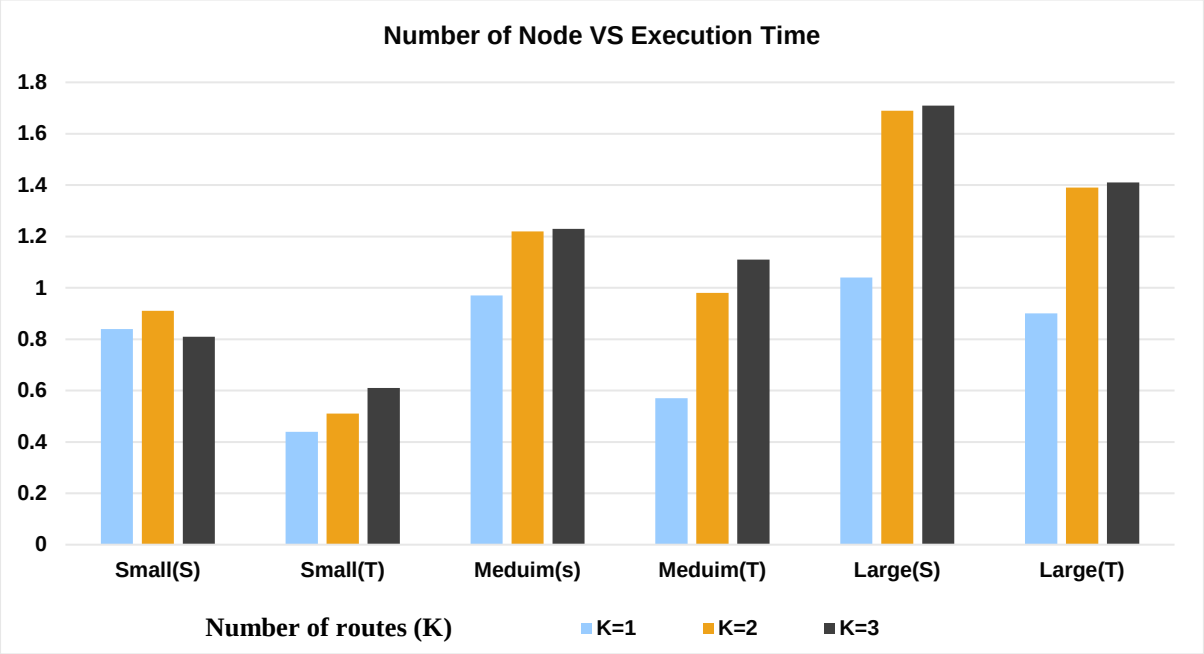
3KSP

Compute the Execution Time of the Program

Experimental results are reported in this section, comparing the performance of the two implementations KSP for static and dynamic network and KSP for the dynamic network using timetable data directly, based on the public transportation network of Nottingham City. The programs used a batch mode command line executable, which is executed several thousand

times with a unique command line in each case. And using the network with 20 nodes the average runtime for the KSP for the static and time-dependent network is 62 second while the average runtime for the KSP for the dynamic network using timetable data is 61 second for finding 3 to 4 shortest paths.

The last comparison that we have conducted was the evaluation of computational runtime between different network size in both static and time-dependent network and the time-dependent network can be found in figure 5.



(S) Static and Time-Dependent Network / (T) Time-Dependent Network

Figure 5: The Execution time of the program

Performance of the KSP Algorithm

Experimental Results for the KSP Algorithm

The KSP algorithm using static and time-dependent network takes a longer execution time, and the execution time increases significantly with the increase of the length of the journey and the value of K. Taking the travel plan in Figure 3 as an example again, the performance for the KSP algorithm using different networks as follow:

1. When K=2, finding two shortest path from bus stop 1 to bus stop 10, the developed KSP algorithm (using the time-dependent network) spends 1.39 seconds with the large network. And compared with the results of the KSP algorithm using the static time-dependent network, the execution time using large network takes 1.69 seconds.

2. To find three shortest paths (K=3) from stop 1 to bus stop 10, the KSP algorithm (using the time-dependent network) spends 1.41 seconds. Again, compared with the results of using static and time-dependent network takes 1.71 seconds.

Analysis of the KSP Algorithm

The time complexity of KSP algorithms is higher than shortest path algorithms, and it differs with different algorithms in different networks. However, labelling KSP algorithms using time-dependent network take longer execution time but the time complexity can be reduced by an advanced data structure. There are few data structures that give the best theoretical performance. Fibonacci heap (1984) is the best for a general network: O (e + n log n). Balanced binary trees (1998) also have good complexity: O (e log n + n log n) (Giorgio, and Stefano (1984), Giorgio and Stefano (1986), Bruce (1976) and Pape (1974)).

It is clear to see that the program which used a time-dependent network (read bus timetable travel time directly) gives the accurate optimal route and presents the result in shortest time for different network's size.

The KSP algorithm developed in this work is based on the label setting algorithm, so the computational complexity is similar to the generalized Dijkstra's algorithm.

The main advantages of the algorithm are that if only one route from a source location to the single destination location is required, the algorithm can be completed when the label of that destination location is set. Therefore, the label setting algorithms are very suitable for applications such as route guidance system where the objective is to find KSP between two specific locations. (Fu and Rilett, 2005). The time complexity of the algorithm is O (n^2) for n iterations (Hribar, et al, 1995).

Summary

This section started to find K-Shortest Paths using two different networks, in the static network and time-dependent network. Two implementations have been tested based on Label Setting shortest path algorithm to produce the best possible performance. The suitability of each method was judged in term of minimum run-time and the quality of the obtainable solution.

There is another problem arises in public transportation networks. Route planning on time-dependent networks is the fact that traffic conditions are not static. In the next section, we will consider the traffic condition. Since the traffic condition changes continually over time, the optimal route will change based on time traffic conditions that provided.

The next step collects traffic data and implements the developed algorithm to Nottingham Public Transportation Network

APPENDIX 2

Finding k-Shortest Path Algorithm by using Traffic Data

1. Introduction

Section 1: has shown that a suitable method to find the shortest path in the real world bus networks is k- shortest path algorithm using the time-dependent network.

In these days, due to failure, maintenance or other reasons related to traffic or weather condition, different kind of uncertainties are frequently encountered in practice and must be taken into account. Based on the real world transportations networks in Nottingham city. There is a developing need for the improvement of the efficiency of urban traffic data in order to ensure the sustainability of modern cities. The requirement for the development of any traffic telematics application is the availability of real-time traffic data. In this research will use traffic data underlying the operation of SCOOT-Urban Traffic Control System at Nottingham city. The SCOOT is an adaptive system optimising the split, cycle and offset times of traffic signals [1996].

In this section, the different types of traffic data are available in the context of Nottingham city network are collected using SCOOT Urban Traffic Control System and implemented for the shortest path algorithm. The main task is to find the optimal route from starting place to a destination on a roadmap. As road traffic conditions may change during the bus journey such as increase of the congestion level or random incidents etc. The best route should be re-evaluated as soon as an update in traffic conditions available.

Nha and his group (2012) considered that “choosing an appropriate route planning algorithm among the existing algorithms in the literature to apply it in real road networks is an important task for any transportation application”. This section firstly implements shortest path algorithm using traffic data (traffic congestion and random incidents).

SCOOT (Split Cycle and Offset Optimization Technique)

SCOOT data is the main source of traffic information in Nottingham city. SCOOT is an Urban Traffic Control system developed in the UK for optimizing network traffic performance, and SCOOT utilizes a model of the controlled network in which traffic flows from the online inductive loop detectors are fed. From this, SCOOT produces a large variety of traffic status messages at regular time intervals.

M02 SCOOT message

This research proposed to use M02 class message which associated with the model information. The M02 message allows the user to assess the situation of the traffic network. M02 provides a straightforward way for link's travel time estimation. Anderson (1997), Carden et al (1989). The following format message of M02

M02 Link <LINK> PERIOD aaaa STP bbbbbb DLY*10 ccccccc

FLO dddddd CONG eeeeee RAW ffffff FLTS gggggg

PERIOD is the time in seconds over which the figures were collected

STP is the approximate number of vehicle stops per hour

DLY is the approximate delay in vehicle hours per hour

FLO is the approximate flow of vehicles per hour

CONG is SCOOT congestion in intervals/hour

RAW is congestion at the detector in intervals/hour

FLTS is number of faulty links in period

This research used M02 message to collect traffic data (traffic congestion and random incidents) of link **N10161E** in Nottingham city network as an example which can be found in figure 6.

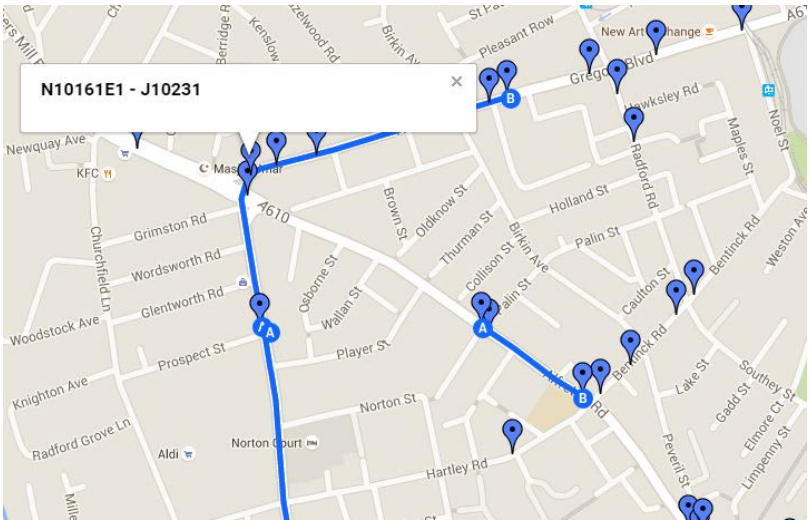


Figure 6: link **N10161E** in Nottingham city map

Example parameters that reported by M02, it shows the congestion level of this link in exact time see the example below

Fr 10:03:30 M02 N10161E PERIOD 300 STP 302 DLY*10 24 FLO 345 CONG 12 RAW 0 FLTS 0

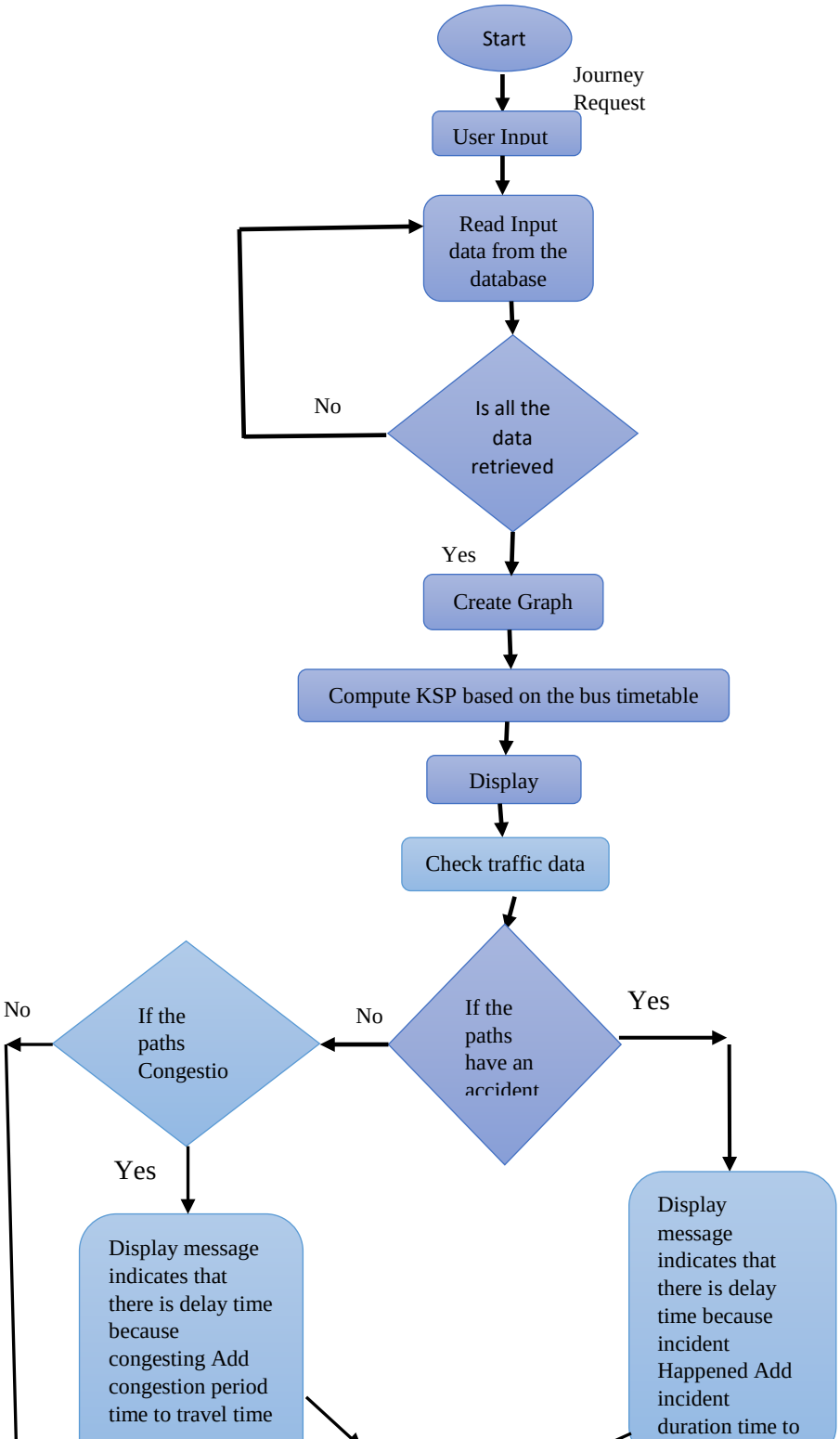
Random incident refers to anything that stops the traffic flow. Such as in the real world could be roadworks or just bad weather conditions causing very slow speeds. Example parameters that reported by M02, it shows a random incident that happened in this link see figure 7. a

Fr 10:34:28 I04 N10161E ---- Incident CLEARED……
Fr 10:34:28 I08 N10161E 1 Duration 16 minutes……

Figure 7, a: format of a random incident message

2. Experimentation

This part implements available traffic data for the k-shortest path algorithm. The program will find k-shortest path based on bus timetable and dynamic parameters (traffic congestion, random incident). The program has been tested under Microsoft Visual C++ 2013 environment and based on k-shortest paths algorithm using a MySQL database, and standard template library to implement the algorithm. Figure 7,b: is a flowchart describing how the program works.



First of all, the program works on the Nottingham city transportation network, and MPhil work works with the network has around 100 nodes (bus stops) and 1276 edges (routes).

Step 1: the program will read entered data in MySQL, and finding first k-shortest paths based on the developed label-setting algorithm using bus timetable.

Step 2: after presented the k-shortest paths, the program will check traffic data (traffic congestion and random incident) for each path. If the path is blocked because traffic jam or incident the program will display reorder of the paths based on changes in travel time and present a new arrival time because the bus will stop for the fixed period of time. This time will add to origin arrival time in bus timetable. So, for example, if there is an incident happened then the duration time if the incident will add to travel time to generate a new arrival time. See figure 7 which shows that there is an incident in the link E10161E at 10:34:28 and the duration time is 16 minute.

Evaluation Results

The program will use a user interface called Navigation System for Nottingham City and will ask the user to enter travel information, see the following example.

Based on the following example inputs

Enter Source location: Canning Circus, Canning Circus (Stop CC08)

Enter Destination location: Whitemoor (Nottingham), Aspley Lane (Stop WM18)

Enter Travel time: 09:00

Enter Number of route(s) (K): 3

The program will compute k shortest path based on the entered data above, and display the result.

The first shortest path is

Canning Circus, Canning Circus (Stop CC08) ---Bus78→ Radford (Nottingham), Boden Street (Stop RA33) ---Bus78→ Radford (Nottingham), Peveril Street (Stop RA19) ---Bus77→ Radford (Nottingham), Hartley Road (Stop RA49) ---Bus78→ Radford (Nottingham), Player Street (Stop RA30) ---Bus78→ Radford (Nottingham), Gregory Boulevard (Stop RA31) ---Bus78→ Bobbers Mill, Chadwick Road (Stop RA32) ---Bus78→Whitemoor (Nottingham), Aspley Lane (Stop WM18)

Departure time is 09:05

Arrival time is 09:47

The second shortest path is

Canning Circus, Canning Circus (Stop CC08) ---Bus77→ Radford (Nottingham), Boden Street (Stop RA33) ---Bus77→ Radford (Nottingham), Peveril Street (Stop RA19) ---Bus77→ Radford (Nottingham), Hartley Road (Stop RA49) ---Bus77→ Radford (Nottingham), Player Street (Stop RA30) ---Bus77→ Radford (Nottingham), Gregory Boulevard (Stop RA31) ---Bus77→ Bobbers Mill, Chadwick Road (Stop RA32) ---Bus78→Whitemoor (Nottingham), Aspley Lane (Stop WM18)

Departure time is 09:08

Arrival time is 09:50

The third shortest path is

Canning Circus, Canning Circus (Stop CC08) ---Bus78→ Radford (Nottingham), Boden Street (Stop RA33) ---Bus78→ Radford (Nottingham), Peveril Street (Stop RA19) ---Bus77→ Radford (Nottingham), Hartley Road (Stop RA49) ---Bus78→ Radford (Nottingham), Player Street (Stop RA30) ---Bus78→ Radford (Nottingham), Gregory Boulevard (Stop RA31) ---Bus78→ Bobbers Mill, Chadwick Road (Stop RA32) ---Bus78→Whitemoor (Nottingham), Aspley Lane (Stop WM18)

Departure time is 09:15

Arrival time is 10:00

After checking traffic data

The first shortest path is

Canning Circus, Canning Circus (Stop CC08) ---Bus78→ Radford (Nottingham), Boden Street (Stop RA33) ---Bus78→ Radford (Nottingham), Peveril Street (Stop RA19) ---Bus77→ Radford (Nottingham), Hartley Road (Stop RA49) ---Bus78→ Radford (Nottingham), Player Street (Stop RA30)

---Bus78→ Radford (Nottingham), Gregory Boulevard (Stop RA31) ---Bus78→ Bobbers Mill, Chadwick Road (Stop RA32) ---Bus78→Whitemoor (Nottingham), Aspley Lane (Stop WM18)

Departure time is 09:05

Delay time due to congestion

Arrival time is 09:47 + 5 min= 09:52

The second shortest path is

Canning Circus, Canning Circus (Stop CC08) ---Bus78→ Radford (Nottingham), Boden Street (Stop RA33)---Bus78→ Radford (Nottingham), Peveril Street (Stop RA19) ---Bus77→ Radford (Nottingham), Hartley Road (Stop RA49)---Bus78→ Radford (Nottingham), Player Street (Stop RA30)---Bus78→ Radford (Nottingham), Gregory Boulevard (Stop RA31) ---Bus78→ Bobbers Mill, Chadwick Road (Stop RA32) ---Bus78→Whitemoor (Nottingham), Aspley Lane (Stop WM18)

Departure time is 09:15

Arrival time is 10:00

The third shortest path is

Canning Circus, Canning Circus (Stop CC08) ---Bus77→ Radford (Nottingham), Boden Street (Stop RA33) ---Bus77→ Radford (Nottingham), Peveril Street (Stop RA19) ---Bus77→ Radford (Nottingham), Hartley Road (Stop RA49) ---Bus77→ Radford (Nottingham), Player Street (Stop RA30) ---Bus77→ Radford (Nottingham), Gregory Boulevard (Stop RA31) ---Bus77→ Bobbers Mill, Chadwick Road (Stop RA32) ---Bus78→Whitemoor (Nottingham), Aspley Lane (Stop WM18)

Departure time is 09:08

There is 16 min delay time due to the incident in the path

Arrival time is 09:50 + 16 min= 10:05

After running program, there is a delay incurred by the occurrence of traffic congestion and random incidents on the paths. The traffic data impact on the overall increase in travel time. The example shows that the k shortest paths is reorder based on traffic changes in the paths.

Simulation of Urban Mobility (SUMO) is chosen for simulating the studied algorithms in this research since it offers an open source package, which is highly portable, applicable for microscopic road traffic to manage each vehicle in the network (Kotusevski and Hawick (2009)). Moreover, SUMO supports Traci interface which provides the way to change vehicle route during runtime (Wegener and et. at 2008).

The German Aerospace Centre began developing SUMO in 2001 and then it has developed and evolved into a suite of traffic modelling utilities which includes a road network. SUMO was improved as the aim that its prospective users will suggest and implement developments to the simulator helping to build a realistic model. SUMO is not a traffic simulator only, but a suite of applications that allows the user to export/import a road network and define its corresponding traffic demand. (Smith, Djahel, and Murphy (n. d)). There are three modules in the SUMO package:

Firstly, SUMO, which reads the input information, processes the simulation and produces Output files. It's called graphical interface SUMO –GUI.

Secondly, NETCONVERT, it reads the input data, and computes it for SUMO and writes the result into different output formats, like XML, or VISUM- networks. Moreover, it's responsible for creating traffic light phases.

Thirdly, DUAROUTER, which is a command line application that, given the departure time, source and destination, computes the routes through the map network itself using Dijkstra's algorithm.

This research focuses on the use of a simulator to find the shortest path and simulate Nottingham city network based on the collected data. It uses "NETCONVERT" to import Nottingham city map from Open Street Map and converted it to be in an appropriate format see figure 8 and figure 9. Traffic demand and routes for each vehicle should be also created. These routing tools take the network and trips to produce route file that contains the routing information for each for each bus and their stops see figure 10.

Figure 8: Nottingham city map in open street map

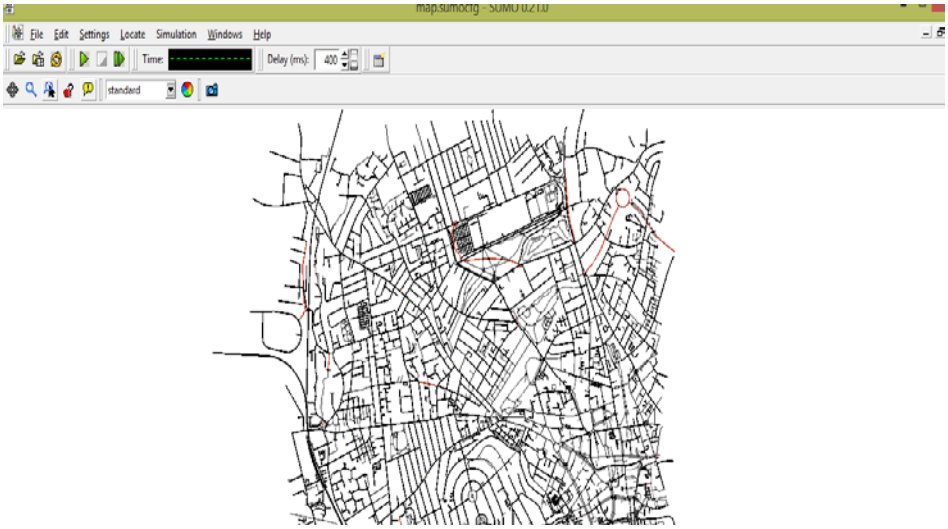


Figure 9: Nottingham city map after converting to SUMO network

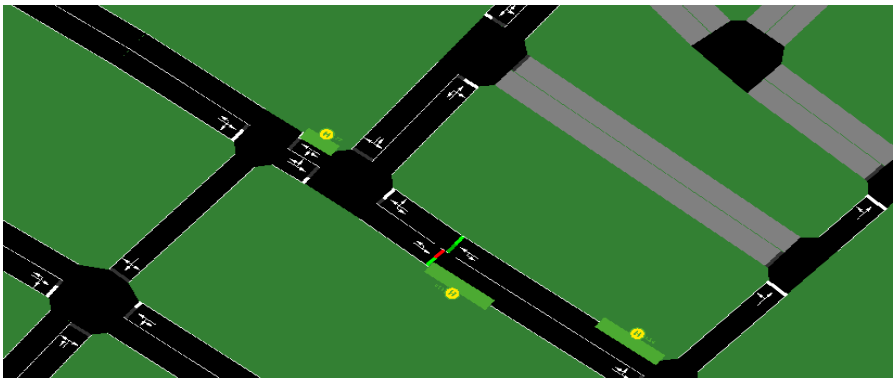


Figure 10: Buses Routes with their stops

Performance Evaluation

The flowchart in figure 11 describes how k shortest path algorithm interacts with the dynamic environment. This research used Nottingham city network and adapt the best routes assigned to a bus according to the updates in congestion level and random incidents collected from the SCOOT system. The program has three main steps which are explained as follows.

The timetable data and traffic data are stored in MySQL database, congestion level and the random incident will represent as any number that indicates that link has congestion except number 0 that indicates that is no congestion. Also, in a random incident.

Step 1:

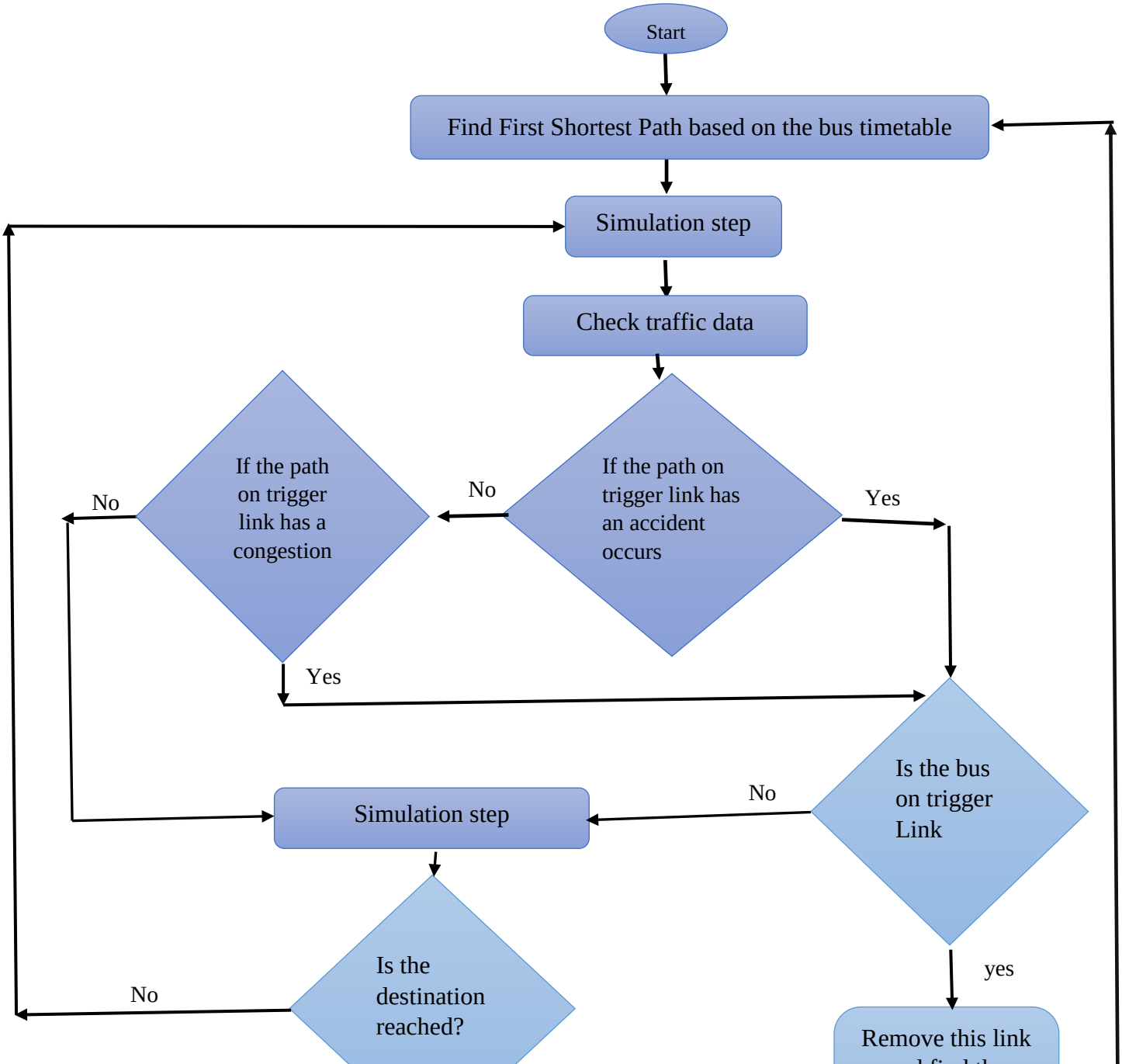
Compute first best shortest path from a source location to destination location according to the developed k-shortest path algorithm.

Step 2:

Re-calculate the best route due to an update in traffic condition. In this case, traffic conditions are checked for an update. If there is an update impacting it will check if the bus on the trigger link then, the program will remove this link and re-apply the algorithm again and compute another shortest path from the current location. And if the bus is not on the trigger link the bus will carry in its trip.

Step 3:

Has the bus arrived at the destination? If no, step 2 is repeated until the Bus arrived its goal location.


Start
Find First Shortest Path based on the bus timetable
Simulation step
Check traffic data
If the path on trigger link has a congestion
No
If the path on trigger link has an accident occurs
No
Yes
Yes
Is the bus on trigger Link
No
Simulation step
Is the destination reached?
No
yes
Remove this link

In our simulation and based on the location information, we define routes and positions of bus stops and let vehicles ("busses") stop at these positions for a pre-given time in the configuration files named map.sumocf.xml, map.rou.xml and map.add.xml.

Firstly, we find the shortest path from origin bus stop to the requested destination bus stop. The route is then assigned for the bus which will follow it as it is simulated by SUMO. The program checks the events variable if any road (link) in the shortest path is blocked because of traffic congestion or random incident. In this case, the program checks if the bus on the trigger link. The K-shortest path algorithm is applied again to re-compute the path from the current location of the bus to the destination. This process will have repeated until the bus arrives at its destination. To do so, our program uses TRACI interface to inform SUMO to change the route which is an already assigned to a bus. In this way, we can update the bus route during runtime.

The program is written in Python. Below figure 12 is part of the code in which label set is a function to find the shortest path from node (start) to the node (target) in the map.

Algorithm 2 in figure 13 calls the label-setting algorithm to compute the shortest path from start node to end node. After finding the route, this route is created in SUMO and is assigned to a bus using TRACI API.

```
def labelsetting(aGraph, start, target):
    # Set the distance for the start node to zero
    start.set_distance(0)

    # Put tuple pair into the priority queue
    unvisited_queue = [(v.get_distance(),v) for v in aGraph]
    heapq.heapify(unvisited_queue)

    while len(unvisited_queue):
        # Pops a vertex with the smallest distance
        uv = heapq.heappop(unvisited_queue)
        current = uv[1]
        current.set_visited()

    #for next in v.adjacent:
    for next in current.adjacent:
        # if visited, skip
        if next.visited:
            continue
        new_dist = current.get_distance() + current.get_weight(next)

        if new_dist < next.get_distance():
            next.set_distance(new_dist)
            next.set_previous(current)

    # Rebuild heap
    # 1. Pop every item
    while len(unvisited_queue):
        heapq.heappop(unvisited_queue)
    # 2. Put all vertices not visited into the queue
    unvisited_queue = [(v.get_distance(),v) for v in aGraph if not v.visited]
    heapq.heapify(unvisited_queue)
```

Figure 12: Algorithm 1 main Function

```
bus_time_query = ("SELECT bus_times FROM bus_time")
link_query = ("SELECT link_a, link_b FROM bus_time")
traffic_query = ("SELECT  link, congestion, accident FROM traffic_data WHERE time = '%s'")

bus_time_cursor.execute(bus_time_query)
link_cursor.execute(link_query)

#trip_time =datetime.datetime.strptime(bus_time_cursor.fetchone()[0], "%H:%m:%S")
trip_time = bus_time_cursor.fetchone()[0]

# Create the link graph
g = Graph()

i = 0
#for vertexs in link_cursor:
vertexs = link_cursor.fetchone()
while(vertexs is not None):
    if i == 0:
        start = vertexs[0]
    finish = vertexs[0]
    g.add_vertex(vertexs[0])
    i = i + 1
    vertexs = link_cursor.fetchone()
print("Nmber of vertexs:", i)

print("Start", g.get_vertex(start), "Finish", g.get_vertex(finish))

link_cursor.execute(link_query)
bus_time_cursor.execute(bus_time_query)
i=0
while (i<bus_time_cursor.rowcount) :
    lc = link_cursor.fetchone()
    #g.add_edge(link_cursor[i].link_a, link_cursor[i].link_b, bus_time_query[i])
    bc = bus_time_cursor.fetchone()
    if bc is None:
        print("BC is none")
        i = i + 1
        break
    g.add_edge(lc[0], lc[1], bc[0])
    i = i + 1

labelsetting(g, g.get_vertex(start), g.get_vertex(finish))
```

Figure 13: Algorithm 2 calculating the shortest path based on the data stored in MySQL

Summary

This chapter implemented k-shortest path algorithms in dynamic road network using available traffic data (traffic congestion and random incident). The simulation tool we used to simulate a road network is SUMO. To update the bus route during simulation runtime to avoid the delay incurred by the occurrence of traffic congestion or random incidents on the roads, we proposed to use TRACI (Traffic Control Interface) API in Python to alter the state of buses during simulation run time.

APPENDIX 3

Data Structure that used in the thesis.

Python Dictionary

In OLPA* and BiOLPA* algorithms, we used a dictionary of a set, because in the open set there is no key is provided, and we don't need to associate values with keys such as in LPA* algorithm. Dictionaries are another example of a data structure and it is used to associate things you want to store to keys you need to get them. A dictionary (or dict) is for matching some items (called "keys") to other items (called "values"). In bidirectional, I use the term "key" to denote the key (1 or -1) of the direction of search (forward, or backward) to decide the direction in which to process the search.

The key, $k(d,s)$, of node s is a vector with two components, while d (direction process) the best selection of either the forward or backward direction by expanding most promising node from both searches.

Key $k(d,s) = [k1(d,s);\ k2(d,s)]$,

Where $k1(d,s) = Min\ (g(d,s), rhs(d,s)) + h(d,s)$

And $k2(d,s) = Min(g(d,s), rhs(d,s))$.

Basic dict operations

Add or change an item in dict x

X [‘key’] = 20.5

if this key is already existing in the dict then change this key value to the new value 20.5. If this key is not existing in the dict then add this new key value to that dict.

Remove item from dict x

del x [‘key’]

Get a length of dict x

Len (x)

Check membership in x (only looks in keys, not values)

Item in x

Item not in x

Delete all items from dict x

X . clear ()

Delete dict x

del x

So now how to access keys and values in a dict

X . keys () # returns list of keys in x

X . values () # returns list of values in x

X . items () # returns list of key-value tuple pairs in x

Python Class Implementation

The key components of any object-oriented software design are the utilization of organized, efficient data structures, referred to as objects. These objects are implemented using class definitions. The main benefit of using the objects to model the problem domain is that they reflect the physical reality. In this project's case, 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*, OLPA*, BLPA*and BiOLPA* algorithms. The example below describes a part of a route class. It contains all variables that are used to support the different algorithms.

```
class Router:
    def Euclidean (self, v1, v2):
        return sqrt (sum ((c1 - c2) ** 2 for (c1, c2) in zip (v1, v2)))
    def reconstruct path (
        start,
        goal,
        pivot):
```

Creating a Bus network

```
from collections import defaultdict
from math import sqrt, floor, ceil
from functools import partial
from random import randrange
from pqdict import pqdict
import colorsys
import operator
import pyqtree
import pickle
import timeit
import glob
import copy
import csv
import re

def getBusGraph(self, buses_lines_csv_path, bus_stops):
        graph = {}
        for bus_csv_file in glob.glob(buses_lines_csv_path):
            with open(bus_csv_file, 'r') as f_bus_csv:
                bus_csv = list(csv.reader(f_bus_csv, delimiter='\t',
                               quotechar='"'))
                for j in range(2, len(bus_csv[0])):
                    prv_time = None
                    prv_stop = None
                    prv_loc = None
                    for i in range(1, len(bus_csv)):
                        cur_stop = bus_csv[i][0]
                        if bus_csv[i][j] != '-' and cur_stop \
                            in bus_stops:
                            cur_time = self.parseTime(bus_csv[i][j])
                            cur_loc = bus_stops[cur_stop]['location']
                            cur_bus = bus_csv[0][j]
                            if prv_stop:
                                if not cur_stop in graph:
                                    graph[cur_stop] = \
    {'location': cur_loc, 'neighbors': {}}
                                if not prv_stop in graph:
                                    graph[prv_stop] = \
    {'location': prv_loc, 'neighbors': {}}
                                if not cur_stop \
                                    in graph[prv_stop]['neighbors']:
                                    graph[prv_stop]['neighbors'
        ][cur_stop] = set()
                                graph[prv_stop]['neighbors'
                                        ][cur_stop].add((cur_bus,
                                        prv_time, cur_time))
                            prv_time = cur_time
                            prv_stop = cur_stop
                            prv_loc = cur_loc
        return graph
```

Figure 14: Method to import CSV file and building bus network

Get Max Speed

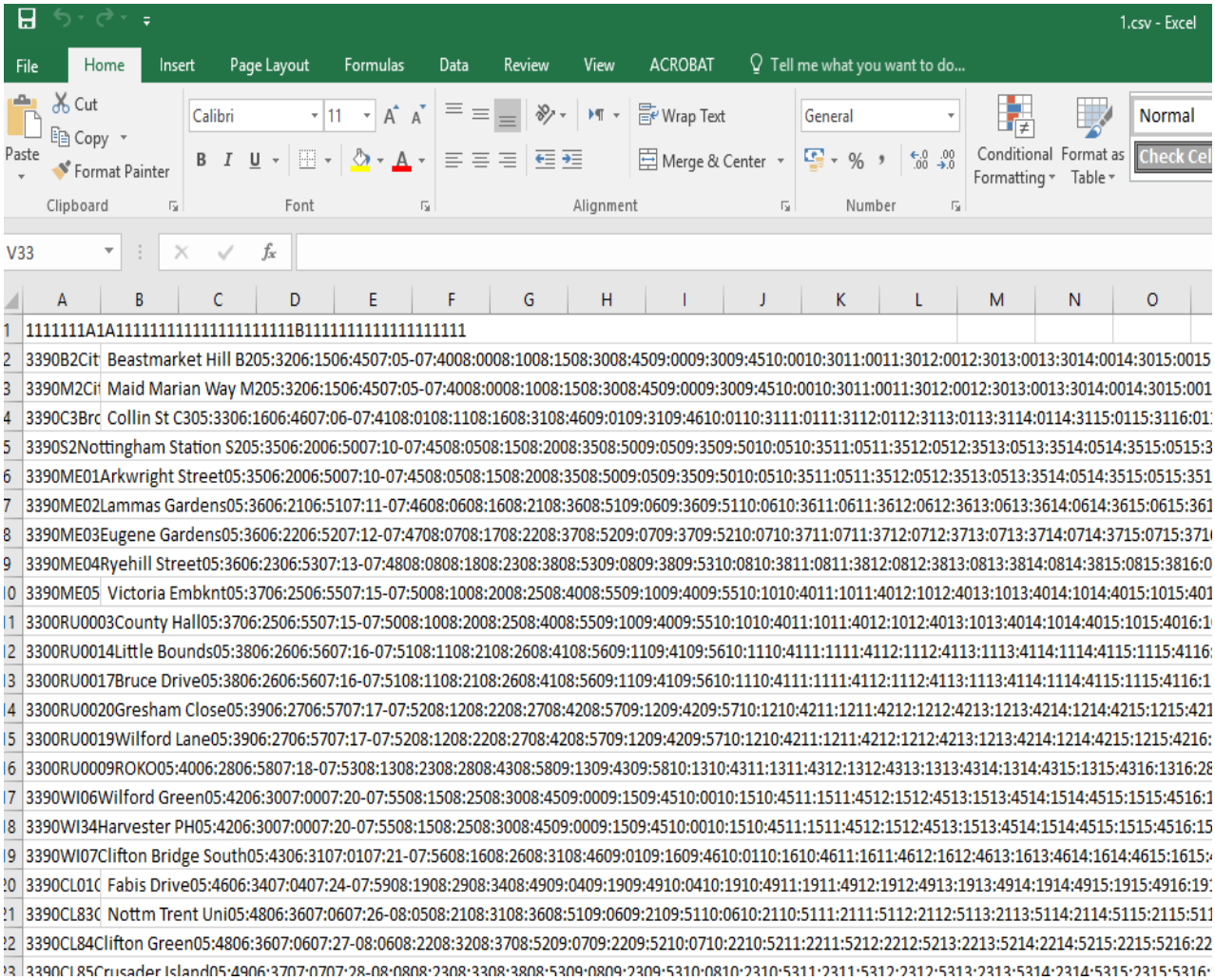
```
def addWalking(
    self,
    graph,
    WALK_LIMIT,
    WALK_SPEED,
    ):

    idx = pyqtree.Index(bbox = [444191.00, 322088.00, 488004.00, 398444.00])
    for stop in graph:
        l = graph[stop]['location']
        idx.insert(stop, l + l)
    for prv_stop in graph:
        prv_loc = graph[prv_stop]['location']
        for cur_stop in idx.intersect((prv_loc[0] - WALK_LIMIT,
                prv_loc[1] - WALK_LIMIT, prv_loc[0] + WALK_LIMIT,
                prv_loc[1] + WALK_LIMIT)):
            cur_loc = graph[cur_stop]['location']
            walk_distance = self.euclidean(cur_loc, prv_loc)
            if walk_distance <= WALK_LIMIT and prv_stop != cur_stop:
                if not cur_stop in graph[prv_stop]['neighbors']:
                    graph[prv_stop]['neighbors'][cur_stop] = set()
                graph[prv_stop]['neighbors'][cur_stop].add(('Walk',
                        0, ceil(walk_distance / WALK_SPEED)))

def getMaxSpeed(self, graph, WALK_SPEED):
    max_speed = WALK_SPEED
    for u in graph:
        for v in graph[u]['neighbors']:
            for m in graph[u]['neighbors'][v]:
                max_speed = max(max_speed,
                                self.euclidean(graph[u]['location'
                                ], graph[v]['location'])
                                / max(self.timeInterval(m[1],
                                m[2]), 1))
    return max_speed
```

Bus 1 Lines

Stop code, stop name, time



1. List of Publications

A) Conference Presentation

Alhoula, W (2014) *Static and Time-Dependent Shortest Path through an Urban Environment Time-Dependent Shortest Path.* Science and Information Technology Conference 2014 May 7-8, 2014 | NTU, UK (**Top Ten Poster)**

B) Conference Publication

- Hartley, J. and Alhoula, W (2014) *Static and Time-Dependent Shortest Path through an Urban Environment Time-Dependent Shortest Path.* Science and Information Conference 2014 August 27-29, 2014 | London, UK

- Hartley, J. and Alhoula, W (2018) *Fast Replanning Incremental Shortest Path algorithm for Dynamic Transportation Networks.* Computing Conference 2019, to be held from 16-17 July 2019 in London, United Kingdom. **(acceptance enclosed)**

C) Ready papers for submission

- Hartley, J. and Alhoula, W (2018) *Improved Fast Replanning Incremental Shortest Path algorithm for Dynamic Networks.* **(Ready for submission).**

- Hartley, J. and Alhoula, W (2018) *A Review Of Shortest Path Algorithms for Public Transportation Network* **(Ready for submission**).