Evaluation of Shortest Path Heuristics

by

ROHANN BOTHA
25029879

Submitted in partial fulfilment of the requirements for
the degree of

BACHELORS OF INDUSTRIAL ENGINEERING
in the
FACULTY OF ENGINEERING, BUILT ENVIRONMENT AND
INFORMATION TECHNOLOGY

UNIVERSITY OF PRETORIA

OCTOBER 2008

Contents

1 Introduction **3**
1.1 Research design . . . 3
1.2 Research methodology . . . 4
1.3 Structure of the document . . . 4

2 Literature Review **5**
2.1 Optimal strategies . . . 5
2.2 Heuristic shortest path algorithms . . . 7
2.2.1 Limit search area . . . 7
2.2.2 Decompose search area . . . 8
2.2.3 Limit links searched . . . 12

3 Pseudo code **13**
3.1 Detail discussion of pseudo code . . . 13
3.1.1 Node Class . . . 13
3.1.2 Edge Class . . . 15
3.1.3 Dijkstra's algorithm Class . . . 16
3.1.4 Branch Pruning method Class . . . 16
3.1.5 $A^{\star}$ method Class . . . 19
3.1.6 Sub-Goal method Class . . . 20
3.1.7 Bi-directional search method Class . . . 20
3.1.8 Hierarchical method Class . . . 22
3.1.9 Main Class . . . 22

4 Results **24**

5 Conclusion **27**

Abstract

In a time where everything needs to move faster, safer and more frequently the need for a quick accurate shortest path algorithm increases. The purpose of this document is to do research on the shortest path algorithms, do software development on them, test them, comment on all of the algorithms and finally state and motivate which one of the algorithms is though to be the best shortest path algorithm. This document will by and large be used by the operation research group of the University of Pretoria to determine which shortest path, if any, will be further developed by their programmers for the *MATSim* project.

Chapter 1

Introduction

Travelling is a part of most people's day-to-day lives. The road networks of a modern city is so complex that this seemingly simple task becomes very complicated; even more so as the population of a city increases and the sheer volume of people congest these networks even further. The Operations Research Group of the University of Pretoria is currently developing a computer based simulation program, *MATSim*, to create a model that will simulate commuter or traffic movement within Gauteng. The goal of the project is to be able to evaluate network changes in Gauteng so as to determine what steps can be taken to lessen or even alleviate the growing traffic problem faced by the citizens of Gauteng. In the traffic model all modes of transport will be included: rail, bus, private vehicle, non motorised transport as well as the paratransit mode in South Africa known as the mini taxi's. The traffic model requires numerous simulated "runs" before accurate statistics can be determined from the simulated data.
When the terminology of simulation modelling is used, the word *entity* refers to the object that will "move" around in the simulation model: people, cars, boats, planes, etc. *MATSim* requires that entities are able to automatically determine the shortest path from their Origin to their Destination, much like a human would determine the shortest path in the same situation. The need to "choose" the shortest path will arise numerous times per simulation run. For that reason the shortest path algorithm has to be as computationally efficient as possible. The goal of this project is to develop a computationally efficient Shortest Path Heuristic (SPH) algorithm which will enable an entity to determine its shortest, from Origin to Destination path in almost the same way a person would.

1.1 Research design

The desired end result is a software plug-in application that will be able to work with the software developed by the Operations Research group (*MATSim*). This plug in will be *Java* based in order to interface with *MATSim*. It will determine the shortest path between two points generated on a map, of Gauteng, as is needed by *MATSim*. The shortest path will be calculated in such a way that a good compromise is made between computational speed and optimality of the route. It is preferred that the entities in the decision making procedure appear autonomous in order to streamline the whole process. A new Shortest Path algorithm concept is not required and existing algorithms and heuristics may be

used in order to develop the software plug-in. The algorithms to be used are discussed in Chapter 2

1.2 Research methodology

The first step in the project will be to do research on currently available algorithms that find Shortest Paths between points in a network. A computer program called *ArcGIS* will be used to convert the map of the area needed by *MATSim* into a network of links and nodes in order to determine the shortest path between two points. The next step will be the development and testing of mathematical and logical shortest path algorithms. The algorithms are referred to as logical or mathematical due to the fact that some will be mostly mathematical, and some will be mostly logical. Each of the algorithms will be compared to an optimal algorithm, *Dijkstra's algorithm*, that was developed by Edsger Dijkstra. The comparison of the length of the shortest path calculated to that of Dijkstra as well as the total average time taken to do the calculations will be used to determine which of the algorithms is best suited to be used in the *MATSim* simulation.

1.3 Structure of the document

In chapter 2 the literature review comprising of all of the different algorithms to be used in this project will be discussed. Chapter 3 will contain all of the algorithms developed for this project as well as a discussion of how each of these algorithms work. The comparison of the algorithms will be done in Chapter 4. Finally Chapter 5 will contain the conclusion and discussion regarding which algorithm is considered to be the best.

Chapter 2

Literature Review

There has been a lot of work done in the field of Shortest Path Heuristics, especially for Route Guidance Systems commonly referred as Global Positioning Systems (GPS). When it is considered that the heuristic solution algorithm required, for the simulation model, will be run numerous times in *MATSim* and that a great number of different algorithms are capable of solving the shortest path problem; the challenge is not to merely develop a new Shortest Path Heuristic (SPH) algorithm, but to find a computationally efficient SPH algorithm or combination of SPH algorithms which will be the most efficient for use in the simulation model.
All of the SPH algorithms that are already developed work on the assumption that an area is already mapped with nodes, representing intersections, and links, representing roads. The links are assigned a “cost” primarily determined by the length of the road. When solving a shortest path problem there are two strategies that should be considered: the *Optimal strategy*, which determines the optimal shortest path within the given area, this strategy is very time consuming, and the *Heuristic shortest path algorithm* which aims to determine a shortest path, in significantly less time than is used by the optimal strategy, which will be the shortest possible path within a certain tolerance.

2.1 Optimal strategies

Fu et al. (2006) states that the majority of optimal shortest path algorithms are applications of dynamic programming which searches for the shortest path in a graph. There are two categories in which an optimal shortest path algorithm can be classified: *Label setting* and *Label correcting*. A feature of the *label setting* (LS) algorithm, often referred to as a one-to-one search method, is that the algorithm can be terminated as soon as the Destination node is reached. The *label correcting* (LC) algorithm, also known as a one-to-all search method, on the other hand cannot provide the shortest path between two nodes until the shortest path to every node in the network is identified.
Dijkstra’s algorithm, conceived by Dutch computer scientist Edsger Dijkstra in 1959, will be used as the optimal strategy in this project Dijkstra (1959). A test area will be generated on which all of the different algorithms and strategies will be tested. Firstly Dijkstra will be used to determine the optimal shortest path between several Origin and Destination nodes. The effectiveness of all of the other heuristics created for this project

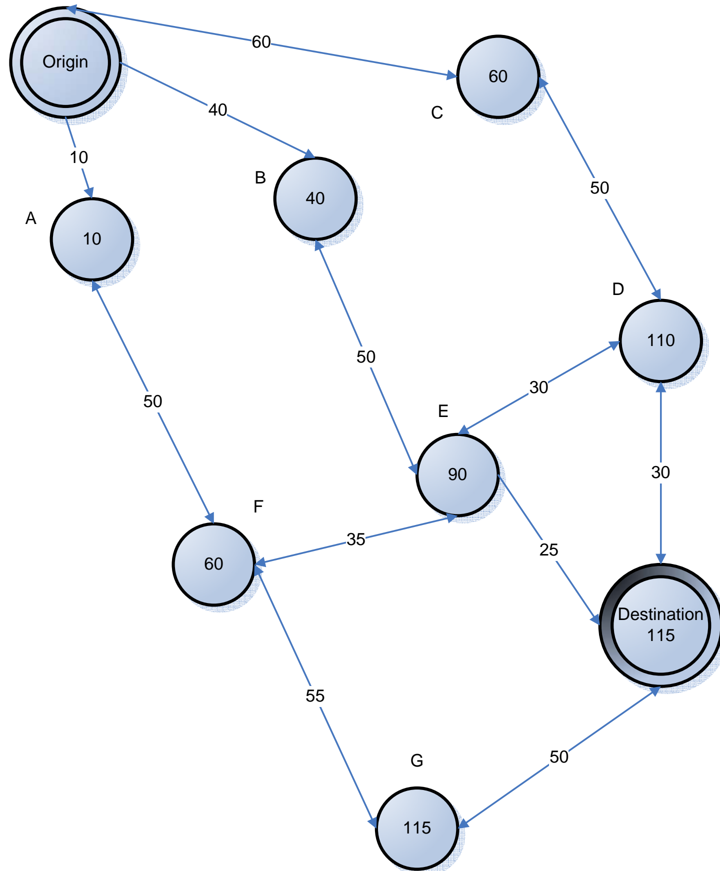

Figure 2.1: *A node-set that has been solved according to the principles of Dijkstra's algorithm. The process followed in each step to determine the shortest path is as follows: Starting at the Origin node determine the shortest path to all of the nodes directly connected to it, then calculate the distance of the second set of nodes. Continue in this fashion. Note that there are instances where nodes like node F has more than one path leading to it. For instance Origin-B-E-F, Origin-C-D-E-F, however, the shortest path with cost 60, displayed in node F, is Origin-A-F*

will then be determined according to how much longer their route is compared to the route determined by Dijkstra. To explain Dijkstra's algorithm in layman's terms it can be compared to a web of knotted strings lain down onto the floor. When a knot is chosen, this knot will signify the Origin, and is lifted off of the ground the other knots will also be lifted off of the ground in the sequence of closest first. Dijkstra's algorithm determines the shortest path between nodes in much the same way.

When using Dijkstra's algorithm three types of nodes will exist until the network has been solved completely: those with optimal shortest path already determined, those with shortest path in the process of being determined and those with shortest path not yet determined. Figure 2.1 shows a small network with all of the shortest paths already determined. To demonstrate the methodology used *node E* will be taken into consideration. The paths leading to *node E* from the Origin, illustrated as Origin-NextNode-NextNode to show the sequence of nodes, are: Origin-B-E; Origin-A-F-E; Origin-C-D-E; Origin-A-

F-G-Destination-D-E etc. As can be seen from Figure 2.1 the shortest path to *node E* is Origin-B-E. All of the shortest paths were calculated in this manner.

2.2 Heuristic shortest path algorithms

Although optimal shortest path strategies, like Dijkstra, will give the proven best possible shortest path between two nodes on a map there is usually not enough time to compute the shortest path using these strategies. Heuristic shortest path algorithms use three main strategies, *limit search area*, *decompose search area*, and *limit links searched*, to reduce the total time to compute the shortest path. In these algorithms the search area is reduced in some way or another. This has the effect that a shortest path may be found that is not as good as the optimal strategy's shortest path, however, this shortest path would be found in a fraction of the time that would have been needed for the optimal shortest path.

2.2.1 Limit search area

Fu et al. (2006) state that the main concept of the *Limit search area* strategy is to use some knowledge about the attributes of the shortest path(s) from the Origin node to the Destination node in order to be able to constrain the shortest path search within a specific area. In other words the knowledge of the position of the Destination node in relation to the Origin node will allow the search algorithm to limit the search area to only those nodes that incrementally decrease the distance from Origin to Destination. There are two methods specifically designed for that purpose, the *Branch pruning method* and the $A^\star$ *algorithm*.

Branch pruning method Developed by Fu (1996) and Karimi (1996) attempts to limit the search area by *pruning* away, or rather not considering, any and all nodes that have little or no possibility of being in the shortest path from the Origin to the Destination node. This is accomplished by making sure that the sum of the shortest path to the node under consideration, *L(i)*, and the estimated distance from the node under consideration to the Destination, *e(i,d)*, is less than the estimated distance from the Origin to the Destination,*E(o,d)*, multiplied by a factor K. Thus the equation $L(i) + e(i,d) \leq KE(o,d)$ will be used to filter out the unnecessary nodes. A graphical illustration of the *Branch Pruning method* is given in Figure 2.2.

$A^\star$ **algorithm** Does not cut away nodes that have a low probability of being on the shortest path; instead it assigns a low priority to them. Thus the algorithm does a *best first* search. It searches through the nodes that have the highest probability, of being on the shortest path, to those that have the lowest probability until it finds the shortest path.
This algorithm was first proposed by Hart et al. (1968) and further discussed and popularised by Nilsson (1971), Pohl (1971) and Pearl (1984). The final search area of the $A^\star$ algorithm and the Branch pruning method are more or less the same, except that with the $A^\star$ algorithm all of the nodes on the edge of the search area

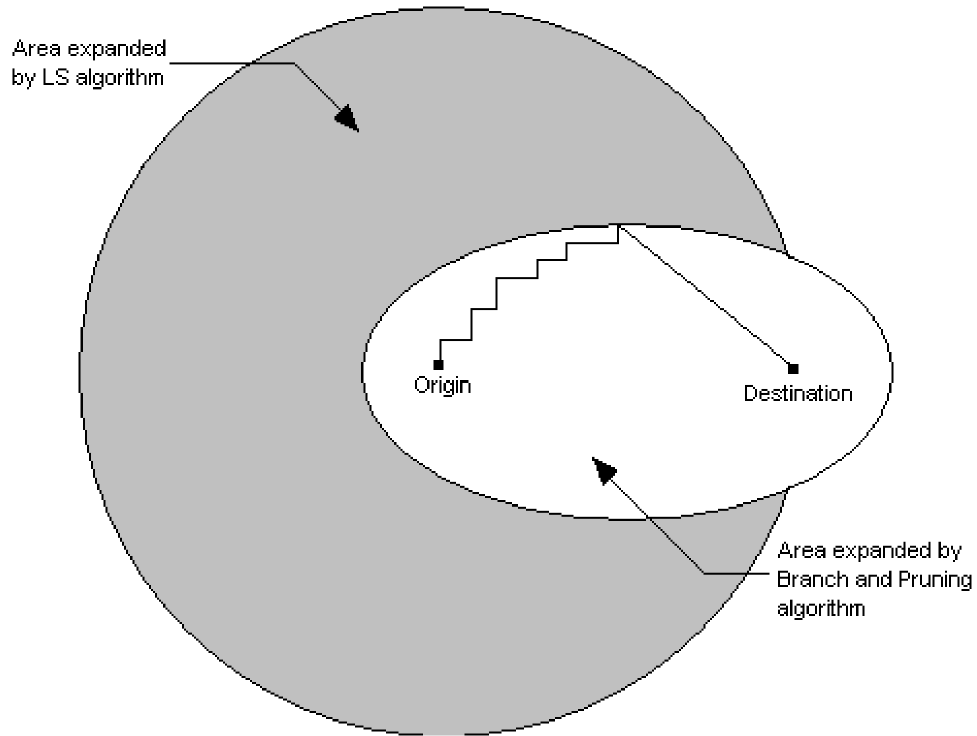

Figure 2.2: *The Branch pruning method reduces the search area from a large area that an optimal algorithm would have searched, to a significantly smaller area. (Source:Fu et al. (2006))*

also have to be evaluated during the search procedure whereas the Branch pruning method discards some of the edge nodes once it is proven to be outside of the feasible solution area, see Figure 2.3 for an illustration. This difference in the two methods has the effect that the *Branch pruning algorithm* maintains a smaller area needed to be scanned, than the $A^\star$ *algorithm*, and is expected to be computationally faster.

2.2.2 Decompose search area

The computational time required to determine the shortest path between the Origin and Destination node depends on the number of nodes searched before the Destination node is reached. As a result, if the Original problem can be decomposed into smaller sub-problems substantial computing time can be saved. Two methods are used:

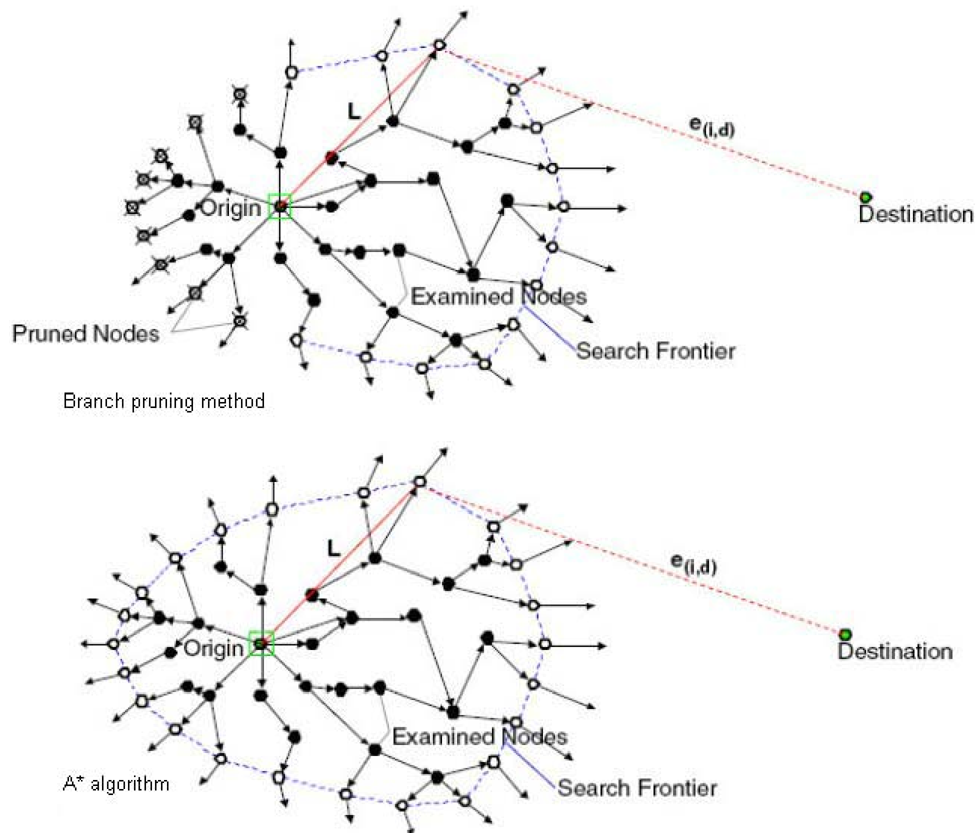

Figure 2.3: *A comparison between the Branch pruning method and the $A^\star$ algorithm, with $L_{(i)}$ being the distance travelled from the Origin to node i and $e_{(i,d)}$ being the estimated distance from node i to the Destination node. The figure shows how the $A^\star$ algorithm, unlike the Branch pruning method, takes low priority nodes into account. (Source: Fu et al. (2006))*

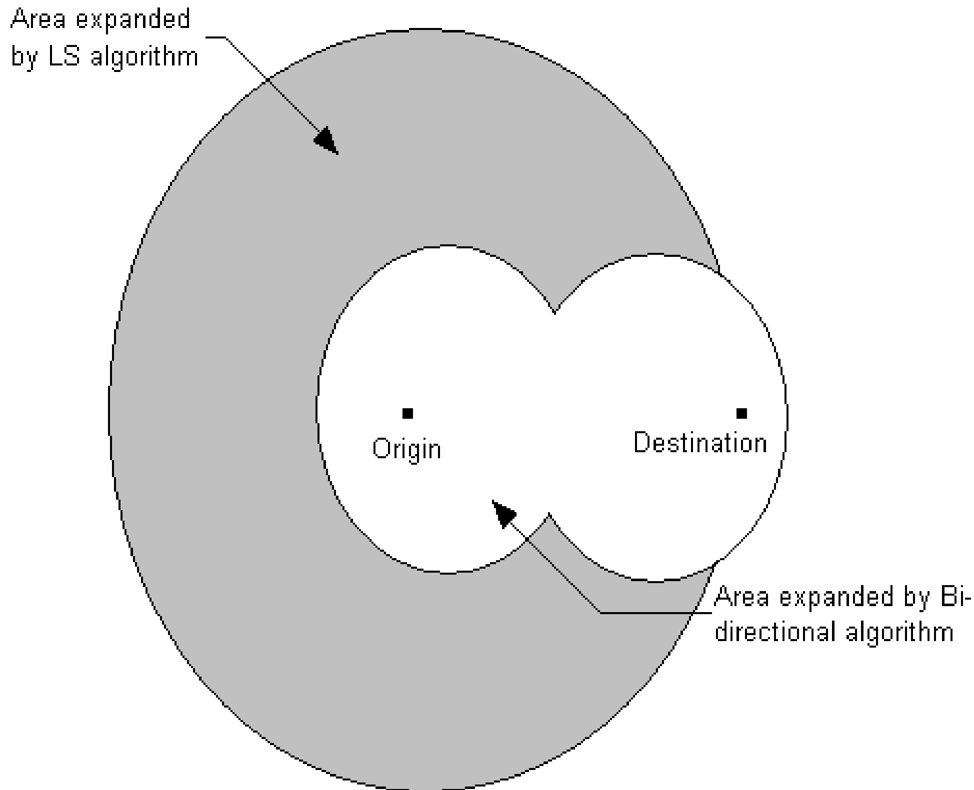

Figure 2.4: *This figure shows how the Bi-directional search method reduces the search area from a large area that an optimal algorithm would have used to a significantly smaller area just because both the Origin and Destination nodes are used in the search. (Source: Fu et al. (2006))*

Bi-directional search method First suggested by Dantzig (1960) and later proposed by Nicholson (1966). Two separate search algorithms are used in this method, one proceeds forward from the Origin node and the other proceeds backward from the Destination node: a solution is found when the two search procedures meet at some middle section. One problem found with the *Bi-Directional search method* is that there is a very high probability that the two search algorithms will miss each other resulting in two searches being done on the same area instead of one search done in two directions. Figure 2.4 illustrates the concept of this method compared to the optimal Label setting algorithm.

Sub-goal algorithm Defines a sub-goal as an intermediate state of the optimal solution to a problem. If one was in the possession of advanced knowledge of where a sub goal should be placed a shortest path problem can be broken up into two or more smaller shortest path problems. In theory, the more sub-goal nodes used and the more uniformly they are distributed the smaller the total search area should be. Figure 2.5 represents the sub-goal method where the search will be from the Origin node to a sub goal node to the Destination node.

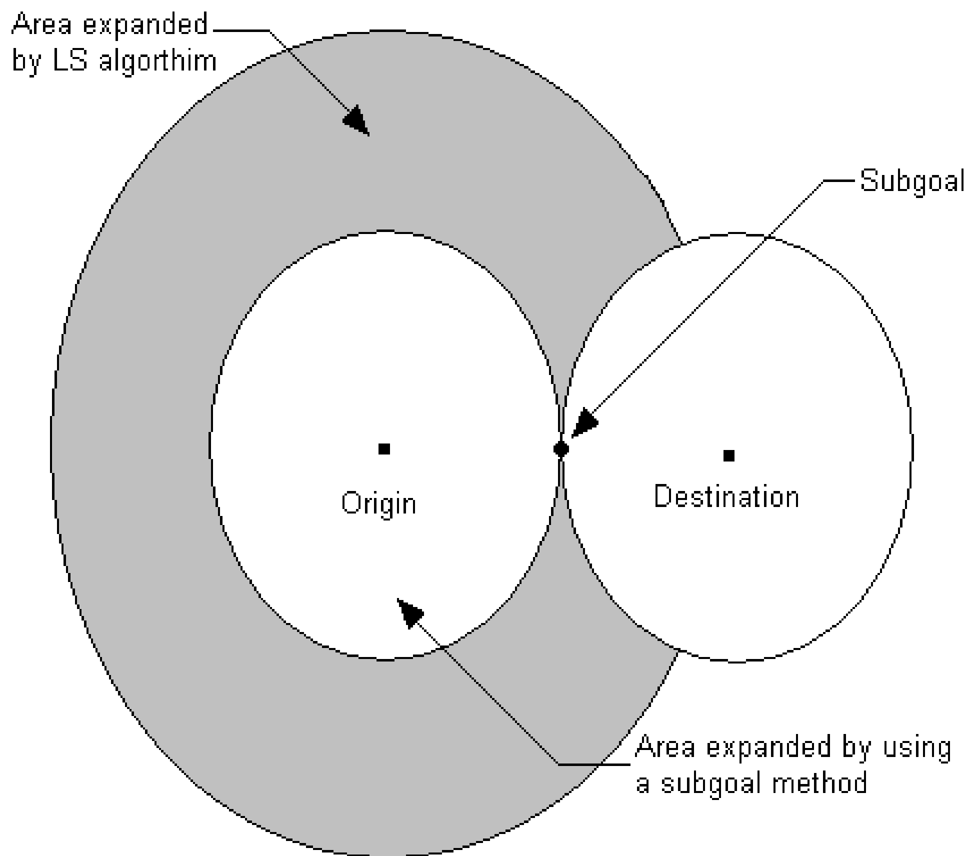

Figure 2.5: *This figure shows how the Sub-goal method reduces the search area from a large area that an optimal algorithm would have used to a significantly smaller area when a sub-goal is defined and both the Origin and Destination nodes are used in the search. (Source: Fu et al. (2006))*

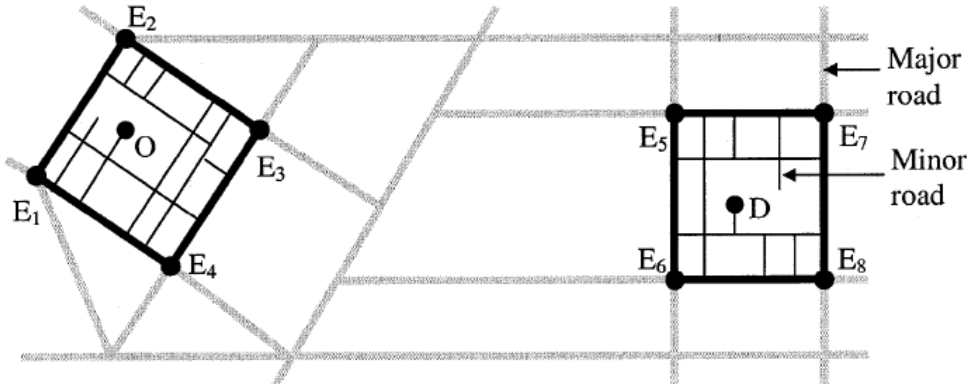

Figure 2.6: *This figure shows how the Hierarchical method reduces the search area from a large area that an optimal algorithm would have searched to a significantly smaller area because the different traffic layers split the shortest path algorithm into three or more sections. (Source: Fu et al. (2006))*

2.2.3 Limit links searched

The *Limit links searched* strategy systematically skips the examination of the links that have a low probability of being on the shortest path. This concept is implemented effectively by the *hierarchical search method*. The idea behind the hierarchical search method is that in order to find a solution to a problem, the search procedure should firstly give attention to the important features of the problem without considering the lower level details, and then complete all of the smaller details later. In a typical traffic network for instance the roads will be classified into major and minor or higher and lower level roads. Shapiro et al. (1992) and Car and Frank (1993) proposed an approach that starts by first finding two shortest paths from the Origin and Destination nodes to the nearest entry points to the next higher level and then calculating the shortest path between these entry points using an ordinary Label setting algorithm, this is illustrated in Figure 2.6.

Chapter 3

Pseudo code

In this chapter the pseudo codes for the shortest path algorithms will be given and discussed in detail. The program was coded in *Java* and was broken up into many different pieces called *classes*. A class can be described as a building block of sorts and in this case the different classes were specifically designed to function as individual, yet interrelated, programs. This was done in order to make the code reusable and thus save time when constructing the final program.

3.1 Detail discussion of pseudo code

The different classes developed are the Node, Edge, Dijkstra's algorithm, Branch Pruning method, $A^\star$ algorithm, Sub-goal method, Bi-directional search method, Hierarchical method, Hierarchical-setup, and the Main class. They will each be discussed in detail in the following subsections.

3.1.1 Node Class

Algorithm 1 shows the pseudo code of the Node Class. This class is the first step in the combination of the all of the classes. The number of nodes in the network as well as each node's name is defined in this class. All of the other classes rely on this information as the size of the network is determined here.
The necessary variables, that is the variables that will be carried by each node of the Node class, are classified in *lines 2-5*, variable *Name* stores the name of each node and is a string variable of the type *ArrayList* which enables it to handle varying number of entries. All of the variables are initialised in *lines 7-13*. The *Getter* and *Setter* methods are set up in *lines 15-24*. *Getter* and *Setter* methods are methods specifically designed to set up or return the value of specified variables, e.g. when *lines 19-20* are executed the current node will receive its name.

```
    Input: Manually entered node names
    Output: A list of nodes
 1  Classification of variables:
 2  Name type String ArrayList;
 3  nodeName type String;
 4  nodeID type Integer;
 5  numberOfEntries type Integer;
 6  Initialization of variables:
 7  Name.add(Name of first node);
 8  Name.add(name of second node) ;
 9  ⇓
10  Name.add(name of last node) ;
11  numberOfEntries = Size of Name Variable;
12  nodeName = "Default" ;
13  nodeID = 0 ;
14  Set up "Getter" and "Setter" methods:
15  getName(i):
16  return Name.get(i);
17  getNodeName:
18  return nodeName;
19  setNodeName:
20  this.nodeName = nodeName;
21  getNodeID:
22  return nodeID;
23  setNodeID:
24  this.nodeID = nodeID;
```

Algorithm 1: *The Node class sets up an array list of the names of the different nodes that will be used in the shortest path algorithms.*

3.1.2 Edge Class

The Edge class shown in Algorithm 2 specifies whether there are links between certain nodes, and if so what distance they are away from each other. Furthermore it also contains the coordinates of the nodes which are used to calculate the distance between nodes that are connected.

The variables of the Edge class are classified in *lines 2-4* an instance of the Node class is called in *line 2* in order for the Edge class to assign the edge distances and the coordinates to the nodes. The Edge matrix classified in *line 3* has rows and columns equal in number to the number of nodes that are in the Node class.

The calculations done in *lines 11-20* are to determine the edge lengths, the standard formula to determine distance on a graph is used.

Input: Manually entered edge values
Output: A list of coordinates and distances for each node

```
1  Classification of variables:
2  Call an instance of the Node class;
3  Edge type matrix with numberOfEntries rows and columns;
4  Coord type matrix with 2 rows and numberOfEntries columns;
5  Initialization of variables:
6  Initialize edge(from)(to);
7  Set up "Getter" and "Setter" methods:
8  getEdge
9  return edge;
10 setEdge
11 List of edges that connect certain nodes;
12 Call setCoord function;
13 for i = 0 to numberOfEntries do
14 |   for j = 0 to numberOfEntries do
15 |   |   if edge(i)(j) connects node i and j then
16 |   |   |   edge(i)(j) =
   |   |   |   edge(i)(j) × √([coord(0)(i) − coord(0)(j)]² + [coord(1)(i) − coord(1)(j)]²);
17 |   |   |   edge(j)(i) = edge(i)(j);
18 |   |   end
19 |   end
20 end
21 setCoord
22 List of coordinates for each node;
```

Algorithm 2: *The Edge class sets up a square matrix for all nodes that defines the distance between them as well as whether or not it is possible to travel between certain nodes. It also specifies the coordinates of the nodes.*

3.1.3 Dijkstra's algorithm Class

Dijkstra's algorithm class, shown in Algorithm 3, will serve as the benchmark for the other algorithm classes to compete against. All of the necessary classification and initiation of variables are done in *lines 5-9*.
The first while loop is used to stop the execution of the program once the Destination node has been reached and all edges leading to or from it have been evaluated for the shortest path. It also has a list, *line 18*, that keeps track of the nodes visited and examined in order to ensure that no duplicate work will be done. Furthermore, the list is used to determine the next node from which *Dijkstra* will continue its search, this node is referred to as the *From* node.
Once the *From* node is determined, the algorithm uses a for loop to search through every available node for a connection between the *From* node and any and all other nodes, referred to as *To* nodes. In this fashion all of the nodes will be evaluated for a possible part in the final shortest path. This process does take a long time, but the thoroughness of the search is exactly what gives the *Dijkstra* algorithm its accuracy.
Line 14 ensures that no invalid node indexes are considered in the class, this is only to prevent errors. The key calculation of Dijkstra's algorithm are contained in *lines 15-19*, this states that if the distance from the Origin node to the *To* node, *afs[to]*, is greater than the sum of the distance from the Origin node to the *From* node, *afs[indexFromNode]*, and the distance from the *From* node to the *To* node; THEN set *afs[to]* equal to that sum.
Note that the way the run-time, which will be essential for evaluation of the different algorithms, of a class is determined is also shown in Algorithm 3.

3.1.4 Branch Pruning method Class

The method used by the *Branch Pruning* algorithm for searching through the different nodes in a traffic network is almost the same as that used by *Dijkstra's* algorithm. The only difference is that an additional filter is used in the *Branch Pruning* algorithm. This filter can be seen in *line 13* of Algorithm 4. This filter reduces the processing time of the algorithm by limiting the nodes searched through. The criteria of the filter used, could be more clearly understood if put into words:
The node will be pruned away if $afs_{[to]} + e_{(i,d)} \geq E_{(o,d)}$ in other words, if the sum of the distance from the *Origin* node to the *to* node, written as $afs_{[to]}$, and the estimated distance from the *to* node to the *Destination* node, written as $e_{(i,d)}$, is greater than the estimated distance from the *Origin* node to the *Destination* node, written as $E_{(o,d)}$, the node will be pruned away.

```
    Input: values received from the main class
 1  Output: The shortest path distance, route and time taken to calculate
 2
 3  Current time recorded;
 4  Initialization and Classification of variables:
 5  Initialize afs matrix to ∞;
 6  initiate path matrix to name of the start node;
 7  Initialize orde matrix to -1;
 8  Initialize inleesPlek, leesPlek and afs(startNode);
 9  Initialize stopping criteria for while loop;
10  while (≠ stop) AND (safety < nodeSize) do
11  |   inc safety;
12  |   set initial value for to variable;
13  |   for to = 0 to numberOfEntries do
14  |   |   if indexFromNode ≠ −1 then
15  |   |   |   if (afs(to) > (afs(indexFromNode) + edge(indexFromNode)(to)))
    |   |   |   then
16  |   |   |   |   afs(to) = afs(indexFromNode) + edge(indexFromNode)(to);
17  |   |   |   |   path(to) = path(indeFromNode) + " " + nodeName;
18  |   |   |   |   Ensure that the correct node will be examined next;
19  |   |   |   end
20  |   |   end
21  |   end
22  end
23  Current time recorded;
24  Determine difference in times recorded;
```

Algorithm 3: *Class Dijkstra uses the Dijkstra algorithm to solve for the shortest path in a transportation network*

```
    Input: values received from the main class
 1  Output: The shortest path distance, route and time taken to calculate
 2
 3  Current time recorded;
 4  Initialization and Classification of variables:
 5  Initialize afs matrix to ∞;
 6  initiate path matrix to name of the start node;
 7  Initialize orde matrix to -1;
 8  Initialize er, inleesPlek, leesPlek and afs(startNode);
 9  Initialize stopping criteria for while loop;
10  while (≠ stop) AND (safety < nodeSize) do
11  |   inc safety;
12  |   Initialize fak, estimateFromItoD, factor and estimateFromOtoD;
13  |   if (afs[er] + estimateFromItoD) ≤ estimateFromOtoD then
14  |   |   Initialize to;
15  |   |   for to = 0 to numberOfEntries do
16  |   |   |   if indexFromNode ≠ −1 then
17  |   |   |   |   if
    |   |   |   |   (afs(to) > (afs(indexFromNode) + edge(indexFromNode)(to)))
    |   |   |   |   then
18  |   |   |   |   |   afs(to) = afs(indexFromNode) + edge(indexFromNode)(to);
19  |   |   |   |   |   path(to) = path(indeFromNode) + " " + nodeName;
20  |   |   |   |   |   Ensure that the correct node will be examined next;
21  |   |   |   |   |   er is set equal to the next From node ;
22  |   |   |   |   end
23  |   |   |   end
24  |   |   end
25  |   end
26  end
27  Current time recorded;
28  Determine difference in times recorded;
```

Algorithm 4: *The Branch Pruning class uses the Branch Pruning method to solve for the shortest path in a transportation network*

3.1.5 $A^\star$ method Class

The difference between the $A^\star$ method class, shown in Algorithm 5 and *Dijkstra's* algorithm class is that the order in which the nodes are examined differ from each other. The $A^\star$ method class uses $F_{(to)}$, defined as $F_{(to)} = afs_{(to)} + e_{(i,j)} + e_{(i,d)}$, which is the sum of the distance already travelled, $afs_{(to)}$, the distance to the next node, $e_{(i,j)}$, and an estimate of the remaining distance to the Destination node, $e_{(i,d)}$, thus the $A^\star$ method class does a best first search of the nodes in the traffic problem. *Dijkstra's* algorithm class however, does not use any methods to do a best first search and just searches through each of the nodes until the shortest path to the Destination is found.

Input: values received from the main class
1 **Output**: The shortest path distance, route and time taken to calculate
2
3 *Current time recorded*;
4 *Initialization and Classification of variables:*
5 Initialize afs matrix to ∞;
6 initiate path matrix to name of the start node;
7 Initialize orde matrix to -1;
8 Initialize inleesPlek, leesPlek and afs(startNode);
9 Initialize stopping criteria for while loop;
10 **while** $(\neq stop)$ AND $(safety < nodeSize)$ **do**
11   inc safety;
12   **for** $to = 0$ **to** $numberOfEntries$ **do**
13   **if** $indexFromNode \neq -1$ **then**
14   **if** $afs_{(to)} + e_{(i,j)} + e_{(i,d)} \leq F_{(to)}$ **then**
15                 $afs_{(to)} = afs_{(From)} + edge_{(From)(to)}$;
16                 $path_{(to)} = path_{(FromNode)} +$ " " + nodeName;
17                 $F_{(to)} = afs_{(to)} + e_{(i,j)} + e_{(i,d)}$;
18   Ensure that the correct node will be examined next;
19   **end**
20   **end**
21   **end**
22 **end**
23 *Current time recorded*;
24 Determine difference in times recorded;

Algorithm 5: *The $A^\star$ Class uses the $A^\star$ algorithm to solve for the shortest path in a transportation network*

3.1.6 Sub-Goal method Class

The Sub-Goal method class, shown in Algorithm 6, is an implementation of the Dijkstra algorithm class. The reason the Sub-goal method algorithm is effective is that it has additional knowledge of points through which the algorithm will have to search in order to find the final shortest path. The Sub-goal algorithm calls up a number of instances of the *Dijkstra* algorithm, in this case three, and then determines three separate shortest path which if combined form the total shortest path from the Origin to the Destination node.

Input: values received from the main class
1 **Output**: The shortest path distance, route and time taken to calculate
2
3 *Current time recorded*;
4 Call an instance of Dijkstra's class;
5 Run Dijkstra's class from the Origin node to sub-goal1;
6 Run Dijkstra's class from sub-goal1 node to sub-goal2;
7 Run Dijkstra's class from sub-goal1 node to Destination node;
8 *Current time recorded*;
9 Determine difference in times recorded;

Algorithm 6: *The Sub-goal method uses the Dijkstra algorithm to solve for the shortest path in a transportation network, using knowledge about the network to reduce the total search area*

3.1.7 Bi-directional search method Class

The Bi-directional search method class, shown in Algorithm 7, is an implementation of the Dijkstra algorithm class. The computational time saving factor of the Bi-Directional search method is that it reduces the number of nodes visited during a shortest path search. As can be seen in Algorithm 7 *Dijkstra's* algorithm is called twice, each time in its own section. The first section starts a normal *Dijkstra* search from the Origin to the Destination node, while this search is going on the code on *Lines 16-20* continuously check whether or not the halfway point has been reached. If the halfway point has been reached the current section is stopped, and the next section is started, which then determines the shortest path from the Destination to the Origin node. While the second section is running, *Lines 35-40* continuously check whether or not the current path under evaluation contains the same node as one of the paths from the previous section. If there is such a node then the second section is stopped and the results are displayed on the screen. If no node exists that is the same for both sections section 2 will run to completion and a *Dijkstra* search will have been done from the Destination to the Origin node.

```
   Input: values received from the main class
1   Output: The shortest path distance, route and time taken to calculate
2
3  Initialization and Classification of variables Section1:
4  Initialize all variables exactly like in Dijkstra's algorithm
5  while (≠ stop) AND (safety < nodeSize) do
6      inc safety;
7      set initial value for to variable;
8      for to = 0 to numberOfEntries do
9          if indexFromNode ≠ −1 then
10             if (afs(to) > (afs(indexFromNode) + edge(indexFromNode)(to))) then
11                 afs(to) = afs(indexFromNode) + edge(indexFromNode)(to);
12                 path(to) = path(indeFromNode) + " " + nodeName;
13                 Ensure that the correct node will be examined next;
14             end
15         end
16         for k = last node checked to new nodes checked do
17             if search ≥ halfway done then
18                 start Section2 search;
19                 end this search;
20             end
21         end
22     end
23 end
24 Initialization and Classification of variables Section2:
25 Initialize all variables exactly like in Dijkstra's algorithm
26 while (≠ stop) AND (safety < nodeSize) do
27     inc safety;
28     set initial value for to variable;
29     for to = 0 to numberOfEntries do
30         if indexFromNode ≠ −1 then
31             if (afs(to) > (afs(indexFromNode) + edge(indexFromNode)(to))) then
32                 afs(to) = afs(indexFromNode) + edge(indexFromNode)(to);
33                 path(to) = path(indeFromNode) + " " + nodeName;
34                 Ensure that the correct node will be examined next;
35                 for k = last node checked to new nodes checked do
36                     if last node of pathOrigin[k] = last node of pathDest[k] then
37                         remember both paths;
38                         end class;
39                     end
40                 end
41             end
42         end
43     end
44 end
45 Current time recorded;
46 Determine difference in times recorded;
```

Algorithm 7: *The Bi-directional search method uses the Dijkstra algorithm to solve for the shortest path in a transportation network, by searching from the Origin node to the Destination node and from the Destination node to the Origin node and then combining the two paths as they meet, thus searching over a much smaller area than*

3.1.8 Hierarchical method Class

The Hierarchical method class, shown in Algorithm 9, is an implementation of the Dijkstra algorithm class. Before the three instances of *Dijkstra's* algorithm are called in the Hierarchical method class, the Hierarchical method searches through the nodes from the Origin node to the closest *Main Road* node and from the Destination node to the closest *Main Road* node, done in *Lines 9 - 38*. Once this preliminary search is done *Dijkstra's* algorithm is run as shown in *Lines 39 - 42*.

3.1.9 Main Class

The Main class, shown in Algorithm 8, is used to call all of the different classes, in any order, to be able to utilise them. Although this class does not add any value to the algorithms it is necessary in order to tie up all of the different classes

Input: values received from Node and Edge classes

1

2 *Initialization and Classification of variables:*
3 Call an instance of the Edge class;
4 Call an instance of the Node class;
5 Initialize indexFromNode and indexDestinationNode;
6 Initialize startNode and DestinationNode
7 Call and run a shortest path algorithm;

Algorithm 8: *The Main method is used to call and run all of the other algorithms*

```
    Input: values received from the main class
 1  Output: The shortest path distance, route and time taken to calculate
 2
 3  Current time recorded;
 4  Initialisation and Classification of variables:
 5  Initialise certain nodes as type Main Road;
 6  Initialise SubFrom and SubDest to -1;
 7  Initialise From and Dest to false;
 8  Initialise count to 0;
 9  while ≠ From do
10      if FromNode + count < number of Nodes then
11          if NodeType[FromNode+count] = Main Road then
12              SubFrom = NodeID[FromNode+count];
13              From = true;
14          end
15      end
16      if FromNode - count > 0 then
17          if NodeType[FromNode−count] = Main Road then
18              SubFrom = NodeID[FromNode−count];
19              From = true;
20          end
21      end
22      int count;
23  end
24  while ≠ Dest do
25      if DestNode + count < number of Nodes then
26          if NodeType[DestNode+count] = Main Road then
27              SubDest = NodeID[DestNode+count];
28              Dest = true;
29          end
30      end
31      if DestNode - count > 0 then
32          if NodeType[DestNode−count] = Main Road then
33              SubDest = NodeID[DestNode−count];
34              Dest = true;
35          end
36      end
37      int count;
38  end
39  Call an instance of Dijkstra's class;
40  Run Dijkstra's class from the Origin node to SubFrom;
41  Run Dijkstra's class from SubDest node to Destination node;
42  Run Dijkstra's class from SubFrom to SubDest;
43  Current time recorded;
44  Determine difference in times recorded;
```


Algorithm 9: *The Hierarchical method uses the Dijkstra algorithm to solve for the shortest path in a transportation network, by using highway systems. Much like the Sub-goal method the Hierarchical method uses sub-goals, the difference is, that the Hierarchical method finds the sub-goal itself, and these sub-goals are located on a main road.*

Chapter 4

Results

In order to test which one of the algorithms is the best a set of test data had to be set up. The digital mapping software known as *ArcGIS* was used to plot 1200 random points across a map of Gauteng. These points were then sorted and became the nodes used in this project.
Each one of the nodes were assigned a value if they were connected to another node and the coordinates, retrieved from *ArcGIS* in decimal degrees, were used to determine the distance between connecting nodes. The final step in the processing of the data into a useful form was to classify some nodes as *main roads*.

To determine how well each of the different algorithms performed several tests were run, each with a different number of nodes to be utilised by the algorithms. For each of these tests four Origin and Destination nodes were chosen at random in order to asses how well the algorithms performed at different Origin and Destination distances. Each one of the tests were run 80 times in order to form an overall perspective of the average performance of each algorithm. The results obtained have been tabulated below.

In table 4.1 it was surprising to see that the Branch Pruning algorithm and the $A^\star$ algorithm did not perform as was theoretically stated. The shortest path distances generated by these algorithms was exactly as great as that obtained by *Dijkstra*. The Branch Pruning algorithm did however find the shortest path in an average time of **94.22%** of what was required by *Dijkstra*, the $A^\star$ needed **116.96%**.

A rather different, and expected, picture was found in table 4.2, where the Bi-directional search method needed, on average, **47.71%** of the time required by *Dijkstra* but yielded shortest path that were on average twice, **206.23%**, as long as that yielded by *Dijkstra*. The Sub-goal method delivered very poor results, yielding on average shortest paths **107.95%** the length of *Dijkstra* and taking **158.00%** as long.

Finally the Hierarchical method in table 4.3 gave the best results in the region where it operated. It needed **41.92%** of the time *Dijkstra* did and yielded shortest path distances **108.59%** the length of *Dijkstra*. The Hierarchical method could not find a shortest path when there was less that 300 nodes available, and thus the data for it is only up to the

Number of nodes and search range	Dijkstra	Branch Pruning	$A^\star$	Distance
	Time(ms)	Time(ms)	Time(ms)	Distance(m)
1200 nodes: from 0 to 1199	23.64	22.65	27.55	185446.61
1200 nodes: from 200 to 900	16.03	15.64	18.35	206794.63
1200 nodes: from 500 to 520	1.95	1.36	2.33	78415.93
1200 nodes: from 53 to 977	17.59	17.96	21.1	199768.59
900 nodes: from 0 to 899	12.48	11.9	15.63	178803.77
900 nodes: from 100 to 800	10.35	10.15	13.48	195646.85
900 nodes: from 60 to 750	10.16	10.15	13.48	244341.85
900 nodes: from 320 to 340	1.39	1.58	2.15	262830.07
600 nodes: from 0 to 599	6.44	10.55	10.94	229613.19
600 nodes: from 480 to 500	1.38	0.98	0.96	77255.67
600 nodes: from 200 to 599	8.99	9	11.13	186488.7
600 nodes: from 98 to 464	4.5	4.29	6.26	56455.14
300 nodes: from 0 to 299	3.91	4.68	1.56	349770.99
300 nodes: from 150 to 170	1.38	1.39	1.56	111904.67
300 nodes: from 50 to 200	3.9	3.33	3.91	108366.99
300 nodes: from 63 to 139	1.39	0.99	1.56	67693.16
100 nodes: from 0 to 99	0.96	0.79	0.98	46838.13
100 nodes: from 10 to 20	0.58	0.6	0.78	9720.58
100 nodes: from 40 to 80	1.75	0.59	2.15	243533.2
100 nodes: from 3 to 69	1.58	1.35	1.56	124009.64

Table 4.1: This table shows the data generated by the Branch Pruning, $A^\star$ and Dijkstra's algorithm. Only one column is used to give the travelling distance as the travelling distance is exactly the same for all three of the algorithms.

runs with 600 nodes.

Number of nodes and search range	Bi-directional		Sub-goal	
	Time(ms)	Distance(m)	Time(ms)	Distance(m)
1200 nodes: from 0 to 1199	11.51	241527.28	24.03	185446.97
1200 nodes: from 200 to 900	3.51	206794.63	11.54	224576.34
1200 nodes: from 500 to 520	0.59	91456.48	17.58	196291.57
1200 nodes: from 53 to 977	9.76	199768.59	10.94	199768.59
900 nodes: from 0 to 899	5.49	178803.77	9.18	178803.77
900 nodes: from 100 to 800	3.3	310459.18	6.44	195646.85
900 nodes: from 60 to 750	5.08	244341.85	7.21	244341.85
900 nodes: from 320 to 340	0.39	319964.67	0.39	262830.07
600 nodes: from 0 to 599	4.69	229613.19	8.98	229613.19
600 nodes: from 480 to 500	0.79	77255.67	0.78	77255.67
600 nodes: from 200 to 599	1.35	186488.7	3.55	186488.7
600 nodes: from 98 to 464	0.99	56455.14	2.35	56455.14
300 nodes: from 0 to 299	1.75	349770.99	2.73	349770.99
300 nodes: from 150 to 170	0.78	232337.82	1.19	111904.67
300 nodes: from 50 to 200	0.99	156580.27	2.34	108366.99
300 nodes: from 63 to 139	0.58	67693.16	1.19	67693.16
100 nodes: from 0 to 99	0.98	46838.13	7.63	46838.13
100 nodes: from 10 to 20	0.59	171955.47	0.78	9720.58
100 nodes: from 40 to 80	1.19	401428.65	2.34	243533.2
100 nodes: from 3 to 69	0.58	262084.01	2.14	124009.64

Table 4.2: This table shows the data generated by the Bi-directional search and Sub-goal algorithms.

Number of nodes and search range	Hierarchical	
	Time(ms)	Distance(m)
1200 nodes: from 0 to 1199	2.13	269455.5
1200 nodes: from 200 to 900	2.34	276612.02
1200 nodes: from 500 to 520	0.79	79961.89
1200 nodes: from 53 to 977	1.19	199768.59
900 nodes: from 0 to 899	1.94	262812.65
900 nodes: from 100 to 800	1.38	195646.85
900 nodes: from 60 to 750	0.58	207443.7
900 nodes: from 320 to 340	0.56	175684.51
600 nodes: from 0 to 599	1.74	294439.88
600 nodes: from 480 to 500	0.59	77255.67
600 nodes: from 200 to 599	0.95	177253.6
600 nodes: from 98 to 464	12.46	56455.14

Table 4.3: This table shows the data generated by the Hierarchical algorithm.

Chapter 5

Conclusion

It would appear that the best algorithm to use would be the Hierarchical method. This method takes less than half the time to determine the shortest path between an Origin and Destination node than the optimal algorithm developed by Edsgar Dijkstra, further the shortest path distance found is extremely close to that yielded by *Dijkstra*.
Even though the Hierarchical method does not function properly in a traffic network where the number of nodes is below a certain number, *Dijkstra* could be used instead, as the increased computation time that will be experienced will not be very dramatic.
In this project the developing of a multi-threaded program, a program that runs parts of itself in parallel, was contemplated for the Hierarchical method, Sub-goal method and the Bi-directional search algorithm. As the processor of the simulation computer that this project was developed on does not have dual processors it would have been futile to attempt such a task. However if one of the above mentioned algorithms were to be used in a much greater network than the network used to test the functionality of the algorithms, it would be wise to consider developing them with a multi-threading capacity as this will further increase the effectiveness of the algorithms.

Bibliography

Car, A. and Frank, A. (1993). General principles of hierarchical spatial reasoningthe case of way-finding. *Proceedings of the sixth.*

Dantzig, G. (1960). On the shortest route through a network. *Managemnentm science*, 6:187–190.

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

Fu, L. (1996). Real-time vehicle routing and scheduling in dynamic and stochastic traffic networks. *Computer aided Civil engineering*, 14:309–319.

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

Hart, E., Nilsson, N., and Raphael, B. (1968). A fromal basis for the heuristic determination of minimum cost paths. *IEEE Transaction, system science and cybernetics*, 4:100–107.

Karimi, H. (1996). Real-time optimal route computation: a heuristic approach. *Journal of intelligent transportation systems*, 3:111–127.

Nicholson, J. (1966). Finding the shortest route between two points in a network. *Computer Journal*, 9:275–280.

Nilsson, J. (1971). *Problem-solving methods in arteficial inteliligence*. New-York; McGraw-Hill.

Pearl, J. (1984). *Heuristics: Intelligent search strategies for computer problem solving.* Addison-Wesley Publishing Company.

Pohl, I. (1971). Bidirectional search. *Machine intelligence*, 6:127–140.

Shapiro, J., Waxman, J., and D., N. (1992). Level graphs and approximate shortest path algorithms. *Networks*.