Improved Shortest Path Routing Based on A* Algorithm

A thesis submitted in partial fulfilment of the
requirements for the award of the degree

Bachelor of Engineering (Computer)

from

University of Wollongong

by

Shujian Zhou

School of Electrical, Computer and Telecommunications Engineering

June, 2014

Supervisor: Prof. Fazard Safaei

Abstract

Shortest path routing algorithm is widely used in many fields including network routing and navigation in electric maps. Most of the shortest path routing methods are based on Dijkstra's algorithm which has been suggested for many years and its efficiency cannot be ensured in large scale maps. In this paper, A* algorithm and some of its variants that would be helpful to the improvement of routing algorithm would be introduced and investigated. Both Dijkstras and those advanced versions of A* algorithm would be discussed and their features would be compared later on in this investigation.

Above all, a simulation implemented on real map will be presented in this thesis. It is expected to demonstrate the performance of all these routing approaches, as well as the differences between them. In summary, this paper is going to provide an insight of these shortest path routing algorithms.

Acknowledgements

Firstly, I would like to express my appreciation to my supervisor Prof. Farzad Safaei for his valuable guidance and giving me inspiration.

And then, I would like to say thanks to Ryan Pon, a programmer in U.S, for his helping me to finish the simulation.

Lastly, I would like to say thanks to my friend Fan Fei. He always helps me when I am confused and give me passions in studying computer programming.

Statement of Originality

I, Shujian Zhou, declare that this thesis, submitted as part of the requirements for the award of Bachelor of Engineering, in the School of Electrical, Computer and Telecommunications Engineering, University of Wollongong, is wholly my own work unless otherwise referenced or acknowledged. The document has not been submitted for qualifications or assessment at any other academic institution.

Signature: _____

Print Name: _____

Student ID Number: 4656933

Date: _____

Contents

List of Tables

List of Figures

Abbreviations and Symbols

$h()$	Heuristic function
$g()$	actual cost from start node to current node
$O(f(x))$	Timing complexity
$d()$	Actual cost between two nodes

List of Changes

Section	Statement of Changes	Page Number
Abstract	Slightly modified	ii
2	The explanation of Dijkstra's algorithm has been modified	3

Chapter 1

Introduction

This paper is organized as six chapters. The first chapter provided a brief introduction of shortest path routing and its utilizations. This is followed by chapter 2, in which a systemic review of Original Dijkstra's with its improved approaches and A* algorithm is presented here. In chapter 3, a series of advanced approaches based on A* algorithm will be discussed in detail. After this, in chapter 4, the simulation will be described which is established to illustrate the gap between all these algorithms. In chapter 5, the results of simulation will be analysed and discussed. Lastly, a conclusion of this entire project will be given in chapter 6.

1.1 An Introduction of Shortest Path Routing

With the development of electronic maps and embedded system, people are seeing an increasing number of navigation applications on smartphones or in-vehicle navigation systems. If someone was planning to drive from home to another place that he does not know before, he would probably require help from electronic maps to find where it is and the shortest path. In this case, a series of algorithms and techniques are introduced to solve this problem. Among all these algorithms, which including Dijkstra's algorithm and A* algorithm, Dijkstra's algorithm[5] is the most popular one

1.2 Introduction of Routing Algorithms

In network routing, Dijkstra's algorithm is the most common method and it is the foundation of OSPF. However, this is totally different from real road network routing. The most important point is that, in network, routing can be done completely only after the routers acquire complete topology of the network. In other words, in Dijkstra's algorithm, all nodes in network will be searched or visited. But in transport routing, people will only consider the path between start and destination. Once utilized in real road network routing, it would search all nodes within the

network, so it would waste a significant amount of time and computing resources. Consequently, Dijkstra's algorithm is modified to be applied in road network routing. During the procedure of modified Dijkstra's algorithm, the searching will stop immediately once the destination was found.

Being similar with Dijkstra's algorithm, A* algorithm, proposed in[6],is a type of BFS (Breadth-First-Search) algorithm. The difference is that heuristic function is introduced to determine the order in which the nodes are visited and which direction should the algorithm search along. In this way, shortest path routing can be implemented in a more efficient way and the cost of computing time and other resources would be reduced substantially.

Apart from these algorithms above, researchers have been working on improving A* since its birth. During these years, a number of algorithms were proposed to find the shortest path sooner and consume less memory which include IDA*, Bi-directional search, ALT (A* with Landmarks Triangle Inequality), Pathmax etc.

In this thesis, we are going to analyse all these algorithms and make a comparison among them via testing on a real map. Finally, our aim is to provide a description of their features select the best one.

Chapter 2

Literature Review

In this chapter, a series of basic approaches that can be utilized in shortest path routing are going to be illustrated. And this paper will concentrate on the A* algorithm, which is utilized in this project. In general, the literature review is consisted of three segments. Firstly, the original Dijkstras algorithm would be explained and its disadvantages would be presented. And then, two approaches based on Dijkstras algorithm would be introduced and analysed. Finally, a review of A* algorithm is given to explain it in detail.

2.1 Background

In the field of shortest path routing, it is widely admitted that Dijkstras algorithm was the most essential method to find a path between two nodes. And this approach has been applied in a series of areas, especially in path routing. However, Dijkstras algorithm is not a perfect solution since it is not efficient in speed and memory usage. People have been attempting to address these problems of Dijkstras algorithm for years. Then, Case based reasoning, Knowledge based reasoning, Hierarchal graph are proposed to modify Dijkstras algorithm. Above all, A* algorithm was also suggested to provide better performance than Dijkstras algorithm. The foundation of A* algorithm is based on Dijkstras algorithm. This method uses heuristic approach to minimize the searching scale so that it may address some of these problems and made useful improvements.

2.2 Dijkstra's Algorithm

As a type of greedy algorithm, Dijkstras algorithm is utilized to solve the single-source shortest path routing problem. During the process of Dijkstras algorithm, the cost of shortest path between source node and any other node would be calculated. In terms of the data modelling, the relation among all nodes is represented

by an N dimensional weighted adjacency matrix in which the value of every element demonstrates the costs between each two nodes. As explained in[7], Variable cost$[i, j]$ demonstrates that the weighted value of arc $< V_i, V_j >$, if the point V_i is not reachable from point V_j, $cost[i, j]$ would be set to infinity($cost[i, j] = \infty$). And a vector $Dist[i]$, which is stored in a priority queue, is established to present the minimum distance between start point and every other ones. Therefore, the vector is initialized to be infinity except the one for start node which is initialized to zero, and every time a smaller value is obtained, it will be updated as below:

$$Dist[i] = cost[s, i], Vi \in V$$

Firstly, the algorithm will start with the source node and generates its successors. Then, it will acquire all costs of these successors by adding the distance to current node and the cost of arc between current node and its successors, which is the operation of Dist[i]=Dist[s] +cost[s,i], and put them into the priority queue in increasing order. This operation will update the priority queue and make the algorithm operate in a best-first manner. During this process, if any of these successors is determined to be the goal node, the algorithm will terminates and the path from start to the current node will be returned as the shortest path. If the goal node is not found, the algorithm will proceed to next iteration. As mentioned above, next iteration will start with the node which is on the top of the priority queue. It also means that the best node ever found will be expanded firstly.

Pseudo-code:

```
function Dijkstra(Graph, source):
     dist[source]  := 0
     for each vertex v in Graph:
         if v  source
         dist[v]  := infinity
         previous[v]  := undefined
         end if
         add v to Q
```

```
    end for
    while Q is not empty:
      u := vertex in Q with min dist[u]
         remove u from Q
         for each neighbor v of u:
            alt := dist[u] + length(u, v)
            if alt < dist[v]:
                 dist[v]  := alt
                 previous[v]  := u
            end if
         end for
    end while
     return dist[], previous[]
end function
```

This procedure would be repeated totally for n-1 times, so that, with the purpose of obtaining the shortest path in a large scale graph, the cost of computing will increase exponentially according to the size of graph. Additionally, the algorithm searches from a given node to almost every other node in the network. In this case, a large amount of time would be spent on searching all nodes in the graph. We assume that if people are searching the path from city centre to a northern place, it will be unnecessary to consider roads that lead to the south.

2.2.1 Improved Dijkstra's Algorithm

Although Dijkstras algorithm is widely used in network routing, it is not efficient in road network. Because searching all nodes in a road map which may contains thousands of roads and junctions probably waste a lot of time and a large scale of memory and this would be useless. As an example, if someone is going from a central point to a northern point, he would not consider the roads that lead to the south. Though it is common-sense knowledge, in Dijkstras algorithm, all nodes in every direction would be searched.

In order to improve Dijkstras algorithm, by [1], case-based reasoning and knowledge-based reasoning is suggested. Firstly, case-based reasoning deals with new problems

based on the previous problems. Once a path between two nodes has been calculated, the path and the nodes on it will be stored to provide the system more case. In this way, future problems would be solved more quickly. This approach is similar to humans behaviour. Assuming that a person is going to somewhere he has been, he would go through the way which he took last time instead of taking a map and searching again.

These methods mentioned above could be integrated to solve problems and overcome the disadvantages of the other techniques. Firstly, the case-based reasoner would be used to deal with problems by storing previous shortest routes. If a previous path matches a new start and goal, it would be solved immediately. And there will be no need to implement Dijkstras algorithm. When there is no similar case that matches the problem, the case-based reasoner would try to find a path that provides a partial route and the remaining part will be left to Dijkstras algorithm.

2.2.2 Case Based Reasoning

The case-based reasoning system consists of two components which include case base and problem solver. Each case keeps a list that stored a series of nodes that are in solution paths. When a request come with a source node and a destination node, the system will try to find a similar case to solve the problem.

Scenario 1:

If the source node and destination node is included in a previous solution path, the system would extract this segment of the path and present it as solution.For example, if a previous solution is

(12, 56, 78, 96, 152, 36, 45, 47)

When a user requests the shortest path between 56 and 36, the system matches the above case and provides the segment of it which is related to the new request:

(56, 78, 96, 152, 36)

Scenario 2:

However, in most situations, there is no previous case match the source and destination exactly. Thus, searching from a nearby junction is beneficial to issue that. For example, if someone requests the shortest path from 12 to 45, but the most related case are

(55, 12, 25,) and (95, 55, 45)

In this scenario, the system will search the neighbour nodes around 12, which means that the Case-based reasoning will implement on the nodes that located besides 12. In this way, the corresponding case of 55, (95, 55, 45), will be extracted to solve this problem.

2.2.3 Knowledge Based Reasoning:

If the routing fails in Case-based reasoning, it would be passed to Knowledge-based route finder. There is a serious problem about original Dijkstras algorithm in road map routing that every node has to be visited and stored during the process of searching. This would waste much time and memory capacity. In order to solve this, a new method is used to overcome these disadvantages.

Once source and destination are all acquired, the system would generate a rectangle in which both source and destination locate in. This rectangle would mark the roads and junctions inside it. This isolates a part of the whole road network for Dijkstras algorithm to be implemented. If the path cannot be found in this grid, which means source and destination are not connected in this rectangle, the system will enlarge it until these two points are connected.

In terms of a long distance travelling, a single large grid would not be that helpful because it contains too many roads and junctions to be calculated. In this situation, a combination of grids is proposed. Two grids will be established based on source node and destination node respectively. In these grids, both major roads and minor roads are considered. But only major roads are considered out of the grids. In this way, much useless searching can be reduced and Dijkstras algorithm can be done in a more efficient way.

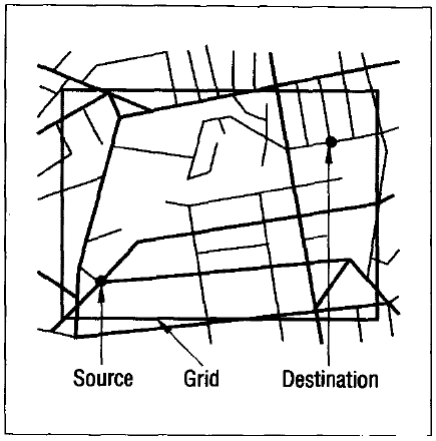


Figure 2.1: Grid Generation[1]

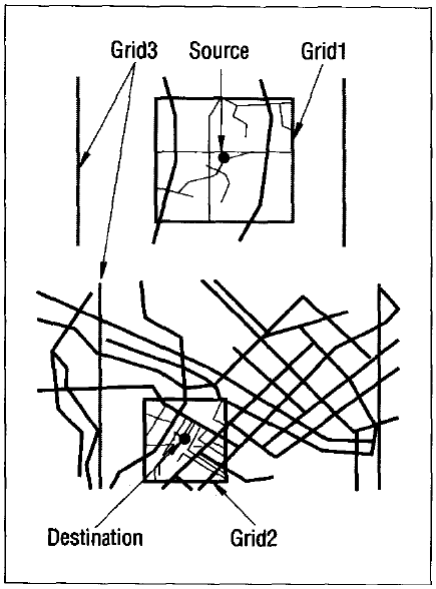


Figure 2.2: Grid Generation2[1]

The improved Dijkstras algorithm can actually reduce the scale of searching and provide a more intelligent way to implement it. But two problems would be the barrier of this approach. Firstly, as to store previous solution path, a device with very large capacity would be needed and it would increase the cost of server. Secondly, it might consume much time on related case retrieval in case-based reasoning.

2.3 Dijkstras Algorithm in Heuristic Approach

It is true that Dijkstras algorithm can be applied easily on a small map. But in a large dense graph, Dijkstras algorithm will pose a series of challenges on storage and I/O operations. With the purpose of overcoming these problems, many algorithms which based on portioning the graph into a group of segments are proposed.

In study given by Chan and Zhang[8], they build a hierarchal graph and pushed all border nodes and bodes which belong to multiple fragment to establish another level. As for fitting in the memory, an acceleration algorithm which called h-Dijkstra algorithm is suggested by Idwan and Etaiwi [3] to enhance the computation in large graph. By the help of h-Dijkstra algorithm, which divide large graph into a series of sub-graphs, the consumption of memory and the number of I/O operations can be reduced significantly.

In a traditional way, each edge in graph connects two nodes. hMetis certificates to specify a parameter K which represent the number of fragments. In this approach, the number of nodes in each segment is approximately the same.

This algorithm contains two steps. Firstly, hMetis would be used in pre-processing. Then, a two-level hierarchal graph which contains the source node and destination node is established. While the nodes are divided into different fragments, the edges are categorized into local and border ones. A local edge is one on which nodes located in the same fragment, while the nodes on a border edge belong to different fragments. A pair of fragments would be said to be adjacent once they occupy edges with endpoints located in different fragments. In the pre-processing, each node in each fragment would be specified in these two types.

Then, by inserting a border node at the middle of each border edge, a two-level hierarchical graph can be established. These pictures below describe the process.

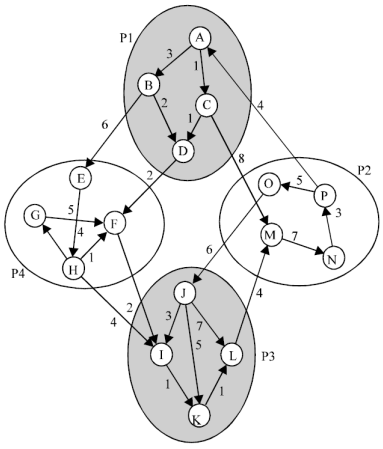


Figure 2.3: Divided Graphs[2]

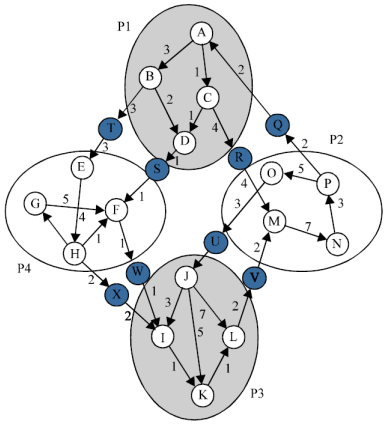


Figure 2.4: Top-level Graphs[2]

The process of building the Hierarchical graph is given below:

1. Fragment list=hMetis
2. For each gragment F in fragment list do
   (a) For each border node belongs to F
   (b) Path=compute the shortest path by using the edges in the flat graph

Having moved these border nodes to top-level graph, the shortest path computation can be implemented by using the top-level graph. There will be an edge created between source node and border node adjacent to the fragment which source node locates in. And another edge will be inserted between destination node and border node adjacent to the fragment in which destination node is in. Then, the shortest path is described in the following figure:

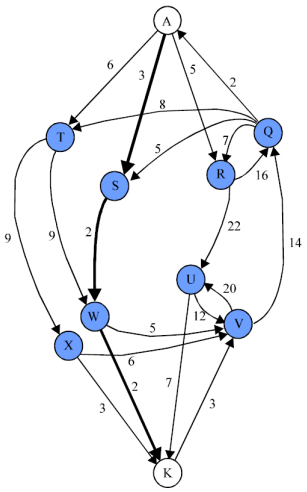


Figure 2.5: The Implementation on Hierarchal Graph[2]

As presented in this figure, point A and K are defined as source and destination. It can be seen from the figure that the shortest path between source node and destination node is calculated among these edges on top-level. By using Dijkstras algorithm, the shortest path can be computed as ASWK.

2.4 A* Algorithm:

The A* algorithm [9] is a generalization of Dijkstra's algorithm and it has been widely used in AI path-finding and graph traversal. In this algorithm, best-first search is used, and this is similar to Dijkstras algorithm. During the procedure of

traversing, A* algorithm will follow a path with lowest expected cost and keep a queue of alternative path on the road. Specifically, a cost function which combines known cost and heuristic value would be used to determine the next node to be expanded. The heuristic function is consisted of two functions:

1. The previous cost function, which indicates the distance between the source and current node.

2. The estimated cost function, which is expected to be an admissible heuristic estimate of the distance from current node to destination.

Being similar to most of the informed shortest path routing algorithms, it searches paths which are expected to be the best. This is also called best-first manner. The difference between A* and other best-first algorithms is that, by the help of heuristic function, A* will firstly extract a child node which is seems closer to destination.

Beginning with the source point, it holds a queue which called open table. It stores nodes that have been visited. This queue is sorted in increasing order which means that the node with least $f(x)$ value would be placed on the top. In each step, the point with the smallest $f(x)$ value will be expanded and its neighbours would be generated into open table. After this, the parent node would be moved to closed table. The procedure will continue until the goal node is obtained. The value of the destination node will be the length of the shortest path, while $h()$ at the goal is zero.

With the aim of presenting the path, each node on the path maintains its predecessor. When the destination node is found, the goal node will point to its predecessor, and same does any other node on the path, until start node is pointed. If the heuristic function is an admissible one. It indicates that the minimal cost of reaching the target point will not be overestimated, then A* is optimal. If a closed table was utilized, the value of $h(x)$ must be constant as well so that A* algorithm can be optimal. This can be explained that for any two of adjacent nodes x and y, in which length of the edge between them is indicated by $d(x, y)$, it should be :

$$h(x) \leq d(x, y) + h(y)$$

As one of uniform-cost algorithms, Dijkstra's algorithm can be treated as a scenario of A* since that has an $h(x)$ of zero. In other words, the value of $h(x)$ of every node is equal to zero. Commonly, DFS (depth-first search) would be implemented by utilizing the A* by assuming that a global variable with value of infinity is existed. Every time a node is visited, its neighbour would be allocated in open table. During each calculation, the variable will decrease by one. The point that is discovered earlier would have the higher $h(x)$ value.

There are a spectrum of simple optimizations detail that can significantly influence the performance of an A* algorithm. The first detail to mention is that the way the priority queue deals with ties may have a substantial influence on performance in some scenario. Once ties are cracked, A* will be implemented like DFS among paths with equal cost.

At the end of the search, the path between source and destination would be required. It is common for every node to hold a pointer to its parent node. Finally, these pointers would be utilized to present the path. If these pointers exist in the priority queue, they cannot appear more than once. An admissible way is to check if the node is kept in closed table. If it does, the pointers will point to another path with lower cost. As for improving the performance in checking, many implementations of min-heap cost $O(n)$ time. Fredman and Tarjan[10] proposed that, by using heap with a hash table, the complexity of time can be reduced to constant time.

As for demonstrating the advantages of A* algorithm over original Dijkstras algorithm, these figures are given below to show that. In these figures, start, end and path are illustrated. Above all, the nodes which are visited and processed during the searching are presented in colours clearly. As can be seen in these pictures, in Dijkstras algorithm, almost every node in the graph is visited while only few nodes are in A* algorithm. Apparently such inefficient visiting would cost a large amount of memory and computing time. Thus, A* algorithm would be much more efficient than original Dijkstras algorithm.

As demonstrated in figure 2.7, both A* algorithm and Dijkstras algorithm are implemented in one graph in which source node and goal node locate in the same position. In Dijkstras algorithm, nearly all nodes in graph are visited and stored,

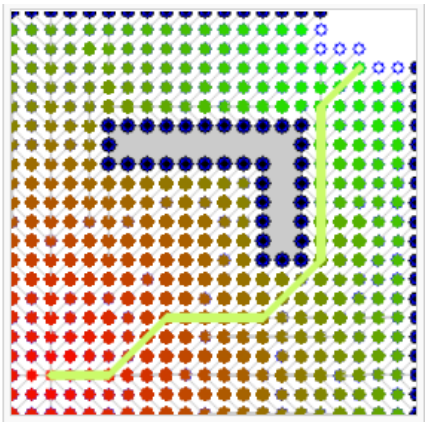


Figure 2.6: Implementation of Dijkstra's Algorithm[3]

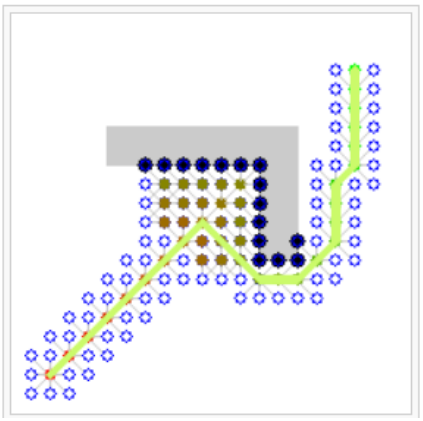


Figure 2.7: Implementation of A* Algorithm[3]

while only few nodes are visited in A*algorithm. It means that, in practical use, A* algorithm will probably reduce the time spending on computation and usage of memory.

Chapter 3

Advanced Algorithms based on A*

Although A* algorithm has improved the performance of routing substantially, it is still impractical on a large dense map. Consequently, researchers have been investigating on modifying and improving A* algorithm. In this chapter, a number of advanced searching approaches based on A* will be presented and discussed.

3.1 Bi-directional Search

As discussed above, the main reason for the inefficiency of Dijkstras algorithm is that, during the procedure, too many nodes would be visited and expanded. In this case, as an improved version of Dijkstra algorithm, bi-directional Dijkstras algorithm was proposed to reduce the scale of searching.

As the name suggests, this method runs the forward searching and reverse searching simultaneously. During the procedure of this algorithm, the forward search starts path-finding from start node while the reverse search starts from the end node. Additionally, in order to determine when the optimal solution will be returned and when searching can stop, the algorithm maintains a pair of global variable μ to retain the shortest path seen so far in both directions. At the beginning of searching, μ is set to ∞. If an edge (a, b) was found by the forward search while b has been visited in reverse search, we assume that the $d_s(a)$ and$d_t(b)$ denote the distance of paths $s-a$ and $b-t$ respectively. When the equation holds that $\mu > d_s(a)+c(a,b)+d_t(b)$, the value of μ along with its corresponding path will be updated. Similarly, this operation will be done in reverse search as well. The procedure will terminate when a new path is found to hold the equation $\mu \leq d_s(a) + c(a, b) + d_t(b)$. The reason for this stoping criterion is that, as a best-first manner algorithm, the path discovered later in dijkstras algorithm would not be the optimal solution. This means that the paths found afterwards would have a larger cost than previous ones, and it will be

unnecessary to proceed the algorithm. Consequently, the value of μ will be returned as the cost of the shortest path.

As demonstrated in figure 5.1 and 5.2, if the searching engine is requested to return a route from two nodes between which the distance is R, it will search an area of $\frac{\pi r^2}{2}$ until the optimal solution can be acquired. On the other hand, if bi-directional Dijkstras algorithm was implemented, the searching area is expected to drop to$\frac{\pi r^2}{4}$. In this way, it is able to reduce the number of visited nodes drastically and improve the performance of routing.

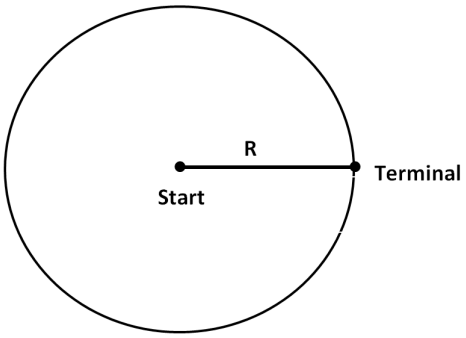


Figure 3.1: Searching Scale of Original Dijkstra's Algorithm

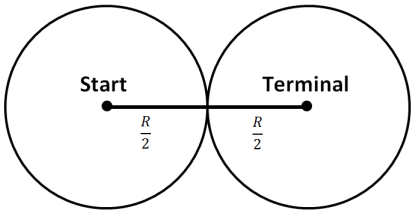


Figure 3.2: Searching Scale of Bi-directional Dijkstra's Algorithm

3.1.1 Bi-directional A* Search

As mentioned above, Bi-directional search throw a significant improvement on Dijkstras algorithm. Inspired by this result, an idea of combining A* searching and bi-directional searching was introduced. However, the combination is not that simple as Bi-directional version of Dijkstras algorithm. The stoping criteria should be determined thoroughly.

Initially, it should be ensured that searches from both directions should use the same estimation. Otherwise, the forward search and reverse search would use different heuristic values. In this case, when forward search and reverse search meet, it

cannot guarantee the best path found ever is optimal. With the intention to ensure the consistency, two strategies can be applied, one is having a new termination criteria and the other one is using a consistent heuristic function. The first method is called symmetric approach which can utilize tight lower bound but cannot terminate immediately as two searches meet. The second one is called consistent approach which stops as the two searches meet, but the consistency requirement restricts the lower bound option. These two approaches will be demonstrated in following sections.

3.1.2 Symmetric Approach

Being similar with Bi-directional Dijkstras algorithm, in symmetric approach, a variable μ is maintained to retain length of shortest path found ever. When an edge (v, w) is scanned by forward search while w has been processed by reverse search, the system will examine if the combination of shortest path of $s-v$ found by forward search and shortest path of $v-t$ found by reverse search has a smaller cost than the best path indicated by μ. This procedure is also implemented in reverse search. The algorithm will terminates when one of these searches scan a node v which denotes a path that has a higher cost than μ. Additionally, an improvement is suggested by [11] that when edge (v, w) is scanned by forward search while w has been processed by reverse one, the forward search will not operate on w, because the shortest path of $w - t$ has been computed by reverse search.

3.1.3 Consistent Approach

We assume that h_r and h_f denote the heuristic functions in reverse and forward search respectively. Ikeda[12] introduced the average function in which $P_t(v) = \frac{h_r(v)-h_f(v)}{2}$ present the potential function for forward search and $-P_t(v)$ for the reverse one. Although this potential function is admissible and consistent, it is not a tight lower bound. In other words, this kind of estimation cannot provide an accurate estimation between current node and the goal, it is actually much lower than the actual cost. As discussed above, a smaller potential function such as average would make more nodes be visited. Therefore, to make this algorithm more efficient, we modify the average function as below:

$$P_f(v) = \frac{h_f - h_r}{2} + \frac{h_r(t)}{2} \tag{3.1}$$

$$P_r(v) = \frac{h_r - \pi_f}{2} + \frac{h_f(s)}{2} \tag{3.2}$$

Since the added items are constant and $P(v)$ is not overestimated eventually, both consistency and admissibility will be ensured. Compared with Euclidean distance, this function still provides a worse bound.

As for the stoping criterion, Goldberg[13] introduced reduced cost of paths. For forward and reverse searches, the reduced cost is defined as below respectively:

$$cost_f(v, w) = cost(v, w) - h_f(v) + h_f(w) \tag{3.3}$$

$$cost_r(w, v) = cost(v, w) - h_r(w) + h_r(v) \tag{3.4}$$

Based on this, assuming that v and w are currently visiting in forward and reverse respectively, the shortest paths of $s-v$ and $w-t$ are indicated by following equations:

$$d_f(v) + P_f(v) - P_f(s) = top_f - P_f(s) \tag{3.5}$$

$$d_r(w) + P_r(w) - P_r(t) = top_r - P_r(t) \tag{3.6}$$

The algorithm stops when following equation holds:

$$[top_f - P_f(s)] + [top_r - P_r(t)] \geq [\mu - P_f(s) + P_f(t)] \tag{3.7}$$

$$d_f(v) + P_f(v) + d_r(w) + P_r(w) \geq \mu + P_r(t) \tag{3.8}$$

3.2 Pruning

During the procedure of searching a path, more than one route will be computed. However, only one path is necessary in general. Therefore, an advanced approach is proposed by L.Poole [14] to prune redundant nodes or routes which are not expected to be on the optimal path. Furthermore, the admissibility of this method is guaranteed though many nodes and paths are abandoned. And through some mathematical derivation, this can be proved to be practical. The details of this

algorithm are provided below.

In order to guarantee that the shortest path would not be abandoned while the optimal solution can be found, one of the following can be done:

1. Ensure that the path found firstly is the lowest-cost path to that node, and remove all subsequent paths found to this node.
2. If a newly found path has a lower-cost than one already found, pruning can be implemented on the path with higher-cost, because this path has less possibility to be on the optimal path.
3. When the searching finds a lower-cost path to a node than a path found previously, it can incorporate a new initial section on the paths.

Although pruning will bring a number of benefits which includes the decrease of memory consumption and the scale of searching, there is a certain risk that some nodes which may be on the optimal path may be abandoned. For the purpose to see when to remove subsequent path would cause dangers, we assume that the algorithm has select a path p to node n when there exists a path that has a lower-cost than p but has not discovered yet. And then, there should be a path $p\prime$ which is an initial part of the lower-cost path, and it is included in path p. Assuming that the last node of path $p\prime$ is $n\prime$, and here should holds an equation that $f(p\prime) \geq f(p)$. And we have:

$$cost(p) + h(n) \leq cost(p\prime) + h(n\prime)$$

As mentioned above, the actual cost of $p\prime$ should be lower than path p:

$$cost(p\prime) + d(n\prime, n) < cost(p)$$

Where d(n,n) denotes the actual distance between n and n. Combine these two equations, we will have

$$D(n, n) < cost(p) - cost(p\prime) \leq h(p\prime) - h(p) = h(n\prime) - h(n)$$

In this way, we can confirm that the first path found is a part of the optimal when the equation $|h(n\prime) - h(n)| <= d(n\prime, n)$ holds. This is also called monotone restriction. This means that the difference of heuristic value between two nodes should be less or equal to their actual distance. In summary, we can realize that the heuristic function is required to meet this restriction. Additionally, we can also say that the

heuristic function should be a lower-bound of the distance between current node and destination node. Thus, Euclidean distance, also known as straight line distance, is an appropriate choice.

Suppose that node v is the parent of node w and t denotes the destination, we will have:

$$h(w) \geq h(v) - d(v, w)$$

It is obviously that this equation is equivalent to $|h(n\prime) - h(n)| <= d(n\prime, n)$. To understand this in an easier way, it can be treated as a form of triangle inequality. The heuristic value of v and t is the long side, and the difference between $h(v)$ and $h(w)$ should be equal to or smaller than the actual distance between v and w. Consequently, it can be proved that Euclidean distance meets the monotone restriction and it is able to be applied in pruning.

Although this approach seems perfectly since it can prune all other path but the shortest one, it is not able to be applied in path routing. First of all, when the path forwards to a dead end, without other options in open table, it cannot continue anymore. Furthermore, during the proof of multi-path pruning, cost of the path between n and $n\prime$, which consists multiple nodes, is assumed to be known when $n\prime$ was expanded. The problem is that when a parent node is expanded, only the cost between children and itself can be obtained. Therefore, path pruning cannot be done. However, this discussion on paths pruning is still valuable for further research. As suggest above, if n is the parent of $n\prime$, the monotonic restriction will be valid in this way since their distance can be obtained during expansion. Then, with monotonic restriction, we are able to confirm that the solution path returned in every iteration is optimal.

3.2.1 The Problem of Decreasing Order

As discussed above, it can be seen that monotone restriction ensures the optimality of searching. But this also causes an issue that will enlarge the searching scale. As the equation given above, we can introduce an equation:

$$f(n\prime) = g(n\prime) + h(n\prime) \geq g(n) + d(n, n\prime) + h(n) = f(n) \tag{3.9}$$

This equation provides a fact that the $f - value$ is in a non-decreasing order as the

algorithm runs. From this conclusion, the later expanded nodes are expected to have higher estimated cost which means less possibility to be considered. In this way, the algorithm will intend to return to the previously visited nodes which are not expected to be on the optimal path. In an intuitive way, as shown in figure3.3, the colour of nodes changes from green to red gradually as searching proceeds. To be more exactly, the red nodes were visited later while those green ones were searched earlier. It can be seen that, as the process of routing, the algorithm repeatedly searches periphery of the searching space. Even when the algorithm almost finishes, the algorithm still searches the area besides the start node. Therefore, while monotonic restriction ensures the optimality, the searching scale may be relative large.

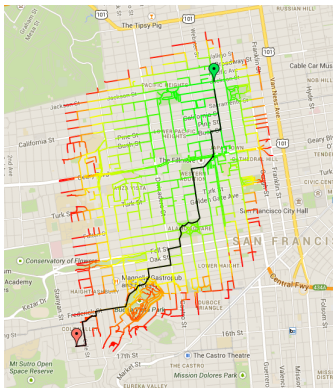


Figure 3.3: Searching Orders of A* Algorithm

3.2.2 The Problem of Ties

As described above, during the searching, a newly expanded node may possess the same or even larger $f - value$ than many earlier visited nodes. This issue is also called tie-breaking problem. However, it is apparently that the newly found one is more likely to be on the optimal path. Hence, we intend to modify the calculation of $f - cost$ to make them differ so than the algorithm can operate in a more intelligent way.

A very important point is that A* algorithm sorts all open table nodes in increasing order, by the help of this method, only one element among them would be extracted and considered. Under this premise, there is a solution to break the tie that we can modify the value of $h()$ slightly in every step. If we decrease the value of $h()$, it can be seen from the previous result that the algorithm will intend to expand a previously expanded node. On the other hand, if we increase the value of $h()$ in

every step, the nodes the algorithm will be more likely to visit a node which is locate on the straight line between start node and end node.

In this way, in each step, we increase the heuristic value slightly:

$$Heuristic* = (1 + \alpha)$$

By using this approach, the algorithm would prefer to expand nodes that are

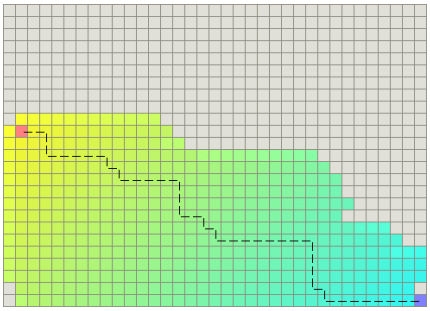


Figure 3.4: Searching Scale of Normal A*

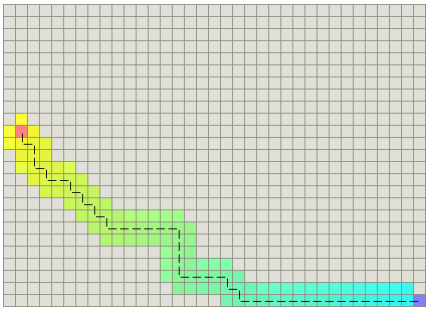


Figure 3.5: Searching Scale of A* Algorithm with Convergence

generated in current stage instead of returning back to visit those nodes which is near start. And it is expected to minimize the scale of searching. We call this method A* algorithm with convergence. It is demonstrated in figure 3.5 that it actually searches in a smaller scale.

However, there exists a problem that the growth of heuristic function will possibly threat the admissibility.Another way to solve this problem, as shown in figure 3.6, is to expand the nodes which are along the straight line from start to goal. In this method, heuristic function will be modified as below:

$$dx1 = current.x - goal.x$$
$$dy1 = current.y - goal.y$$
$$dx2 = start.x - goal.x$$
$$dy2 = start.y - goal.y$$
$$cross = abs(dx1 \times dy2 - dx2 \times dy1)$$
$$heuristic+ = cross \times 0.001$$

This code calculates the vector cross-product from start to destination and current node to destination. If this vector do not line up, the cross-product will increase. Consequently, this feature will provide a preference to the algorithm to visit those

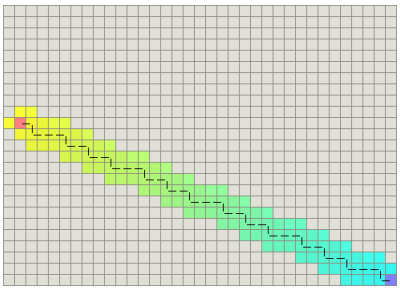


Figure 3.6: A* Restricted on Line

nodes that lies along the straight line from source to destination.

3.3 IDA* and Fringe Search

3.3.1 IDA*

It cannot be denied that, by using heuristic function, the number of visited nodes is hugely reduced. Thus, A* algorithm made a significant improvements than those common greedy algorithms including Dijkstras algorithm on memory usage and computing time as well.

However, when it implements on a very large graph and the distance between start and end is very long, there is still a huge number of nodes would be visited and stored in Open table and Closed table. Furthermore, during the procedure, a sorted Open table is maintained to present the best node which is expected to be expanded in next iteration. In this way, sorting operations would be executed in every iteration. Moreover, as progress of the algorithm, the length of open table would increase rapidly. This means that the consumption of sorting would increase with the progress. Except the open table, the whole closed table will be traversed to examine whether the current node has been expanded or not.

With the purpose of reducing the consumption of storage and sorting, IDA*[15] algorithm, a space-efficient version of A* algorithm, was proposed which uses storage that is linear in the length of the shortest path length.

Prior to the introduction of IDA* algorithm, the original algorithm Iterative Deepening which IDA* was derived from would be introduced.

For the purpose of combining the space efficiency and optimality in depth-first search

and breadth-first search separately, iterative deepening was proposed to fulfil this achievement. As mentioned in[3], the main idea of this method is to recompute the elements of the frontier instead of saving them in a sorted order. In each iteration, searching will be implemented to a limited depth. In the initial phase, it will firstly implement a depth-first limited to 1. Then the limit will gradually increase as the progress of iterative deepening.

When the algorithm is implementing, these should be distinguished:

1. Failure caused by reaching of the depth-bound.
2. Failure that is not related to reaching of the depth bound.

As for the first case, the procedure will proceed with a deeper limit. For the second case, the problem cannot be solved by simply repeating the process with larger depth. The first case is called failure unnaturally while the second one is called failure naturally.
The Iterative deepening search fails whenever the breadth-first would fail. Therefore, it can be ensured to be admissible and return an optimal solution. And the condition of halting is listed below:

1. The truncation of search by reaching the depth bound gives an increase on depth bound.
2. The search returns a route if it would not have been reported in previous iteration.

It seems that iterative deepening would be a waste of computation for its recomputing of each node that has been processed in previous iteration. Assuming that the branching factor is b and the search bound is k. It can be obtained that, in last iteration, b^k nodes are expected to be visited until the bound is reached. Thus, the total number of nodes generated is

$$\sum_{i=1}^{k} ib^{1-i}$$
$$\leq b^k(\sum_{i=1}^{\infty} ib^{1-i})$$
$$= b^k(\frac{b}{b-1})^2$$

As demonstrated above, $(\frac{b}{b-1})^2$ is constant, so we can conclude that this algorithm has the complexity of $O(b^k)$. This conclusion indicates that the scale of computing will increase explosively with the increasing number of nodes and it would not be acceptable in a large graph.

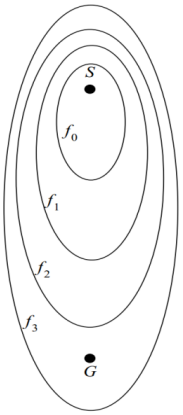


Figure 3.7: $f - cost$ in IDA* [4]

IDA* can be considered as a combination of Iterative deepening and A* algorithm, since the bound in IDA* is $f - cost$ instead of the number of edges. Therefore, it takes the advantage of heuristic function. The IDA* algorithm is demonstrated below 3.8

Pseudo-code:

```
procedure ida_star(root, cost(), is_goal(), h())
  bound := h(root)
  loop
    t := search(root, 0, bound)
    if t = FOUND then return FOUND
    if t =  then return NOT_FOUND
```

```
    bound := t
  end loop
end procedure

function search(node, g, bound)
  f := g + h(node)
  if f > bound then return f
  if is_goal(node) then return FOUND
  min :=
  for succ in successors(node) do
    t := search(succ, g + cost(node, succ), bound)
    if t = FOUND then return FOUND
    if t < min then min := t
  end for
  return min
end function
```

The figure 3.7 presents the $f-cost$ boundaries around initial node. In this diagram, all nodes between boundaries f_i and f_{i+1} have $f(n) = f_{i+1}$. This is similar to the contour of a topographic map.
Instead of expanding and retaining every successor of the current node and proceed to next iteration, IDA* will repeat the procedure with a larger $f-cost$ limit in next loop. It means that, similar to iterative deepening, all nodes that have been visited in previous iteration will be searched again. Thus, in IDA* algorithm, there is no retention for the nodes which have been visited and so it consumes less memory than A*.

However, the price of using less storage is suffering from re-computation and re-expansion. Assuming that, in the worst case, every node has a different $f-cost$ so that only one node can be searched in every iteration. While A* searches k nodes until the solution is acquired, IDA* will examine $O(k^2)$ nodes. Furthermore, the gap between these two method will enlarge with the growth of graph and the slowdown compared to A* will be unacceptable.

3.3.2 Fringe Search

As mentioned before, A* algorithm is a best-first search that will find the shortest path sooner while it consumes a large amount of capacity to retain the previous information. Therefore, IDA* algorithm was proposed to solve the searching issues in a memory-bounded environment and it cost less memory capacity. But this low storage solution does not come for free because it will visit a node for many times that consume more time on computing.

Thus, a new approach is proposed as a combination of A* and IDA* that is able to spans the space/time trade-off between them. Actually, this algorithm is inspired by the problem of eliminating the inefficiencies with IDA*. This approach is called Fringe search which proposed by[4].

Firstly, it should be noticed that IDA* which consumes less capacity has a number of drawbacks compared to A*:

1. As IDA* algorithm cannot detect repeated status, it has to reconstructs the frontier in each cycle which causes repeatedly visiting. While in A*, the retention of previous information is used to avoid repeating.

2. In IDA*, Left-to-right traversal is utilized in searching frontier. It means that it regress to blind search in a depth-first manner. While A* maintain an open table in sorted order so that it will expand the best node firstly and complete the searching earlier.

By introducing the fringe search, these problems above are expected to be solved. As a combination of A* and IDA*, at one extreme, fringe search algorithm will expand nodes in frontier in left-to-right order as in IDA* which is similar to pre-order traversal. At the other extreme, it will maintain a sorted order list to expand nodes in a best-first manner.

3.3.3 Implementation of Fringe Search

Firstly, in fringe search, not all previous nodes would be visited in each iteration, because the algorithm iterates on the fringe of the search tree. In this way, re-expanding can be significantly reduced. Moreover, a data structure will be established in this

procedure which contain *now* list (nodes will be processed in this iteration) and *later* list (nodes will be processed in next iteration). At the beginning, the *now* list starts off with the start node while the *later* list is empty. Then, in each iteration, the head of *now* list will be examined:

1. If $f(head)$ is greater than the searching bound, then it will be move to the tail of later list. This denotes that those nodes which have a greater $f - cost$ than searching limit will be processed in next phase.

2. If $f(head)$ is not greater than the limit, then we insert its successor to the front of *now* list and discard itself from this set.

If the goal has not been found till the end of iteration, the threshold will be increased to the minimal $f - cost$ among the searched nodes.

For the purpose of avoiding repeatedly expanding in a memory-bounded situation, sorting is implemented by establishing a number of buckets. Bucket is utilized to implement the storage of nodes which would be processed in current iteration. The difference between bucket and open table is that every bucket corresponds to a possible range of $f - cost$. By using a number of buckets, Fringe Search would probably expand nodes in an approximately same order with A* and generate the most promising node in priority. Furthermore, the number of buckets could have great impact on the efficiency of this algorithm. On one hand, a large number of buckets, which means a smaller granularity, can make it more possible to expand the best node firstly. In other words, by the help of buckets, Fringe Search can acquire the solution sooner. On the other hand, a small number of buckets mean that Fringe Search will implement as the order in IDA*. In between, using an appropriate number of buckets and get partial ordering is able to profit from best-first search benefits without the overhead of sorting a large table.

3.4 Pathmax

In shortest path routing algorithms, successors of a given node would be generated to implement searching. This procedure is called node expansion. It was stated by Laszlo Mero[16] that node expansion is the most time consuming operation compared

```
Initialize:
    Fringe F ← (s)
    Cache C[start] ← (0, null),
    C[n] ← null for n ≠ start
    f_limit ← h(start)
    found ← false
Repeat until found = true or F empty
    f_min ← ∞
    Iterate over nodes n ∈ F from left to right:
        (g, parent) ← C[n]
        f ← g + h(n)
        If f > f_limit
            f_min ← min(f, f_min)
            continue
        If n = goal
            found ← true
            break
        Iterate over s ∈ successors(n) from right to left:
            g_s ← g + cost(n, s)
            If C[s] ≠ null
                (g', parent) ← C[s]
                If g_s ≥ g'
                    continue
            If F contains s
                Remove s from F
            Insert s into F after n
            C[s] ← (g_s, n)
        Remove n from F
    f_limit ← f_min
If found = true
    Construct path from parent nodes in cache
```

Figure 3.8: Fringe Search

with other administrative activities of the search algorithm. With the intention to reduce node expansion, these approaches were introduced.

As demonstrated in A* algorithm, each time a node has been visited and expanded, the algorithm will firstly search the open table and closed table to determine whether it has been visited or not. Once the current node is known to have been visited, the algorithm will compare the $f - cost$ of this loop with the value that exists in open table. If a better $f - cost$ is obtained in this loop, then the corresponding node along with its $f - cost$ would be placed at the top of open table. Followed by this step, the algorithm will traversal closed table to examine if this node has been expanded. Once it is determined to have been expanded, the system will check if the new iteration could give a better $f - cost$ and update this node with this value. Therefore, closed table would be visited repeatedly. For the purpose of reducing access to closed table, following methods are proposed.

3.4.1 B Algorithm

Algorithm B is an improved version of A* algorithm which makes use of the heuristic information only when it is in accordance with the sub-graph established by search until the give stage.

In order to provide a better performance on A* routing algorithm, Martelli [17] introduced B algorithm by modifying A* algorithm as follows:

1. Insert the initial node into open table and set
$$g(s) = 0, f(s) = h(s), F = 0$$
2. If some nodes in open table are found to have a lower f value than F, then select among them and obtain the node n with the least $g - value$. Otherwise, the algorithm will select the node n in open table with least $f - cost$ and set the global variable $F = f(n)$ and remove n from open table then put it on closed table.

It has been presented above that the crucial part of this algorithm is the global variable F which delivers the greatest $f - cost$ in each phase. F is used to control whether the remaining costs promised by heuristic estimates are in accordance with the propagation for algorithm so far. In other words, F is responsible to inform the current progress of the procedure.

3.4.2 B' Algorithm

Having observed that generating a new node would possibly improve the heuristic estimate of its successors, B algorithm was suggested. In other words, once the information of parent node was acquired, it can be used to modify the heuristic value of its children.

This approach can be considered as a variant of algorithm B. It can be obtained by adding two steps as follows:

1. For each successor m of the currently expanded node n, if
$$h(m) < h(n)c(n, m)$$
then set
$$h(m) = h(n)c(n, m)$$
2. Assuming that m is the successor of n for which $h(n) + c(n, m) = h_m(n)$ is minimal. If $h_m(n) > h(n)$ then set $h(n) = h_m(n)$.

In this way, those inconsistent heuristic functions can be modified to meet the monotonic restriction. Thus, the case of $f < F$ would never appear in this algorithm.

3.4.3 Pathmax

In addition, with the intention to provide a better performance for consistent search, pathmax was suggested. We assume that $h()$ is the estimated distance between current node as above and $h^*()$ denotes the actual cost of the shortest path between these two nodes. Additionally, the heuristic function is assumed to be Euclidean distance which never overestimates the cost. Therefore, we can conclude that for every node n, the expression $h(n) <= h^*(n)$ holds. Furthermore, via triangle inequality there holds an equation for node n and its successor m that

$$h(n) \leq c(n,m) + h(m) \leq c(n,m) + h_*(m) = h_*(n)$$

It can be seen apparently that the heuristic value of parent node can be modified by employing the combination of $c(m,n)$ and $h(m)$. Finally, the $f - cost$ of these two node will be following value and the $f - cost$ of parent nodes will be enlarged.

$$F(n) = h(n) + g(n) = c(n,m) + h(m) + g(n)$$
$$F(m) = h(n) + g(m) = c(n,m) + h(m) + g(n)$$

In this way, the algorithm will follow the consistency assumption. In addition, the $f - costs$ are non-decreasing along any path which means that the nodes visited previously would not have better $f - cost$ than those visited later. Consequently, A* algorithm will never need to reopen the closed table. Otherwise, A* may reopen the closed table many times and the number of visiting closed table would increase exponentially with the number of nodes in the graph. Thus, it may provide an improvement on performance

3.5 ALT(A* with Landmarks Triangle Inequality)

In those previous version of A* algorithms, the lower bounds were calculated by using information implicit in the domain which includes Euclidean metrics, Manhattan distance and Chebyshev distance [18]. Here we have a new version of A* algorithm which is called ALT and proposed in[19], A* with Landmarks and Triangle Inequality. It uses a new approach to estimate the distance between current node and destination. As the name suggests, ALT is based on A* algorithm and employees landmarks and triangle inequality to compute lower bound.

The mechanism of ALT involves several simple steps. Firstly, it will select a number of nodes as landmarks. Then the algorithm will estimate the distance between current visiting node and end node by applying triangle inequality. In detail, assuming that $d()$ denotes the distance from L and any two node in this graph is presented by v and w, by the triangle inequality, the equation $|d(v) - d(w)| \leq dist(w, v)$ holds. After this, the left hand side of this equation will be used as the lower bound. Additionally, in order to obtain a tight lower bound, the system will utilize the maximum one among all landmarks. Next, the lower bound calculated before will be used as the heuristic function which indicates the distance between current node and end node.

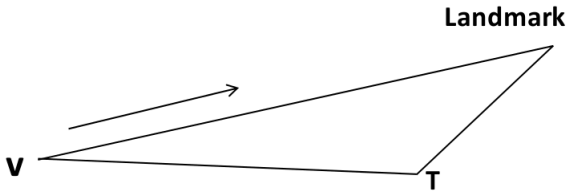


Figure 3.9: Landmark with Triangle inequality

With the intention to improve the performance of ALT, some optimizations can be implemented. For a pair of $s-t$, select a fixed-size subnet of landmarks that provide the higher value of lower bound than others. Thus, during the pre-computing of lower bounds, the searching can be limited in the range of this subnet. However, the

Theorem given by Goldberg demonstrates the fact that using less landmarks would probably result in more nodes scanning, because less landmarks may not provide an accurate, or tight, lower bound. For an example, this theorem is fulfilled in common A* algorithm in which searching with Euclidean distance would visit much more nodes than the one with Manhattan distance, since Manhattan distance is always equal or larger than Euclidean distance between two nodes. Fortunately, this increase is small relative to the improved performance on landmarks computation.

Theorem. *We assume that $h'_t(v)$ and $h_t(v)$ are two heuristic function. While for any node v, $h'_t(v) > h_t(v)$, then the set of nodes scanned by A* which uses $h'_t(v)$ is a subnet of those searched by A* with $h_t(v)$.*

In theory, ALT is a very ingenious method. Firstly, it was mentioned above that it gives a lower bound that never overestimate the heuristic distance. In this way, it is ensured to return an optimal solution. Furthermore, as shown in this figure, by setting a landmark which provides a lower bound and leads the direction of searching, this algorithm will tend to proceed along the straight line between current node and the landmark. Compared with original A* algorithms which search more than one directions, ALT is expected to reduce the searching space significantly.

3.5.1 Landmarks selection:

Finding appropriate landmarks is of vital importance to the performance of ALT, because good landmarks could provide a tight lower bound and minimize the scale of searching. Theoretically, as mentioned in[20],a perfect landmark should locate before the start or after the end. Thus, the method will be able to provide a very tight lower bound and search space can be reduced. For the purpose of selecting appropriate landmarks, those following selecting algorithms are introduced:

1. Random Select:

   It is the simplest way to extract a set of landmarks that just select a number of landmarks randomly.

2. Farthest Landmarks Selection:

   Pick up a node and find another node which is farthest away from the previous

node, then insert these two nodes in the set of landmarks. In each iteration, select and add a landmark that is farthest away from the current set of landmarks. In this way, the minimal distance between each pair of landmarks can be maximized.

3. Planer Landmarks Selection:

   Initially, select a node c that locates at the centre of map. Then, divide the map into a number of pie-slice sectors, each of these sectors contains roughly the same number of nodes. For each sector, extract a landmark that is farthest from c.

   During this procedure, in order to enlarge the distance between each two landmarks, detection is implemented and it works as follows. Assuming that the algorithm is processing sector y and has processed sector y, if the landmark of y locates beside the border of x and y, we will skip the processing on sector y.

Among all these selection algorithms, planer landmarks selection, although consume more pre-computation time, it is expected to give the best performance.

Chapter 4

Routing Algorithms Simulation and Designing

4.1 Simulation Objective

The simulation would be implemented on Original Dijkstras algorithm, Bi-directional dijkstras algorithm, improved A* algorithm and ALT algorithm. During this process, the data including number of expanded nodes, length of resulting path and elapsed time would be demonstrated to present the gap between those algorithms above. Additionally, with the intention to obtain a result that is closer to the real situation, this simulation is implemented on a real map.

On the other hand, for the purpose of investigating the impact of heuristic function on performance and optimality, A* algorithm has been implemented in various weight of heuristic function. This method is expected to provide an insight of heuristic function.

Lastly, all the parameters in this simulation have been processed to be presented in an intuitive way. Firstly, nodes in closed table are demonstrated in colours so that it will be easily to tell the searching scale by simply watching the graph. Secondly, in order to show the process of searching in details, the simulator is designed to display the procedure gradually in form of animation. In summary, this simulation will probably give a simple way to discover the results.

4.2 Evaluation Methodology

Based on previous investigation, which was announced by Laszlo that the time consumed by nodes expansions is much more than that consumed by framework of the algorithm, thus the measurement of nodes expansions is more convincing than running time. Therefore, in this thesis, we focus on the number of expanded nodes and that is going to bring additional benefits of being machine-independent. This

method is expected to give an insight of performances among these algorithms.

On the other hand, since optimality ensures that an optimal solution can be returned, this is a crucial part of shortest path routing. In this project, we firstly implemented Dijkstras algorithm in each scenario and this algorithm will return the actual shortest paths. Then, if the solution path acquired by any other algorithm is longer than Dijkstras solution, we can confirm that this algorithm is not optimal.

4.3 Simulation Construction

Consequently, in this simulation, performance of these algorithms would be measured by the number of nodes in closed table mainly.

For the purpose of realizing the shortest path routing in a real map, this project employees a number of approaches and tools. First of all, this project is written by Python which is a relatively simple object-oriented programming language. And then, in order to acquire the real map data, Google map API is applied.For the visualization, a user interface written by JavaScript is established.

Firstly, there is a fact that the map data from Openstreetmap is based on spherical coordinates. However, the distance estimation of heuristic function and the length of the shortest path should be presented in Plane coordinate system. In this case, we use Harversine formula in[21] to obtain great-circle distance between two points on a sphere from latitudes and longitudes. This method is recommended for calculation of shortest straight-line distance by NASAs Jet Propulsion Laboratory. And this formula is described as below:

$$dlon = lon2 - lon1$$

$$dlat = lat2lat1$$

$$\alpha = (sin(\frac{dlat}{2}))^2 + cos(lat2) \times cos(lat1) \times (\frac{sin(dlon)}{2})^2$$

$$c = 2 \times tan^1(\sqrt{\alpha}, \sqrt{1-\alpha})$$

$$d = R \times c(R\ denotes\ the\ radius\ of\ the\ earth)$$

By using these series of equations, the planer distance, also known as Euclidean distance, can be attained.

4.4 Configuration of Algorithms

In ALT approach, we use *planerlandmarksselections*, which is expected to be the best among all selecting methods, to generate landmarks. Additionally, the value of k is set to 16 which means that there will be 16 landmarks in the graph. For the Bi-directional A* algorithm, we utilize Euclidean distance as heuristic function, and consistent approach is applied as the strategy of stoping. Finally, as for A* with convergence, we set the α to be 0.005.

Chapter 5

Simulation Results and Analysis

5.1 Comparison on Searching Scale

Initially, we implemented these algorithms between two points which are not very far away. It can be seen that, as expected before, Dijkstras algorithm searches much more nodes than others, while ALT searches least nodes. For Bi-directional search, it gives a significant improvement on both dijkstras algorithm and A* algorithm. When it comes to a long distance searching, the gap between these algorithms becomes larger as demonstrated in figure 5.4 to figure 5.21. In fact, for scenario 1 in which the distance between start and end is relatively short, the advantage of ALT is not apparent. Furthermore, as demonstrated in next two scenarios, the difference between ALT and other algorithms increases drastically. In terms of elapsed time, it can be noticed that ALT consumes more time on expanding one node compared to others. This would probably be caused by the process of calculating estimated distance in which the algorithm will consider a number of landmarks and extract one with largest lower bound. However this cost is small relative to the improvements on reducing expansion and ALT always consumes less time than any other algorithm.

5.2 Problem Caused by Poor Landmarks

However, it should be mentioned that, in some situations, the results on two nodes, which are very close, can be hugely different. For example, as illustrated in figure5.1,

Algorithms	closed nodes	Elapsed time(ms)	Length of route(km)
Dijkstra's algorithm	4912	124	2.33
A* algorithm	857	31	2.33
A* with convergence	525	19	2.35
Bi-dijkstra's algorithm	1914	70	2.33
Bi-A* algorithm	452	25	2.33
ALT	449	30	2.33

Table 5.1: Testing on Scenario.1

Algorithms	closed nodes	Elapsed time(ms)	Length of route(km)
Dijkstra's algorithm	14297	299	5.32
A* algorithm	3351	123	5.32
A* with convergence	2004	77	5.47
Bi-dijkstra's algorithm	9987	137	5.32
Bi-A* algorithm	1375	48	5.32
ALT	524	35	5.32

Table 5.2: Testing on Scenario.2

Algorithms	closed nodes	Elapsed time(ms)	Length of route(km)
Dijkstra's algorithm	35105	804	8.87
A* algorithm	5811	212	8.87
A* with convergence	3401	124	9.29
Bi-dijkstra's algorithm	21442	447	8.87
Bi-A* algorithm	3434	99	8.87
ALT	1337	55	8.87

Table 5.3: Testing on Scenario.3

although these two end points are very close, one query expanded 4020 nodes while the other one expanded only 2137. This kind of hug difference would not appear on other approaches. The reason for this phenomenon may be related to landmarks. For the purpose of explaining this phenomenon intuitively, we use a geometry way.

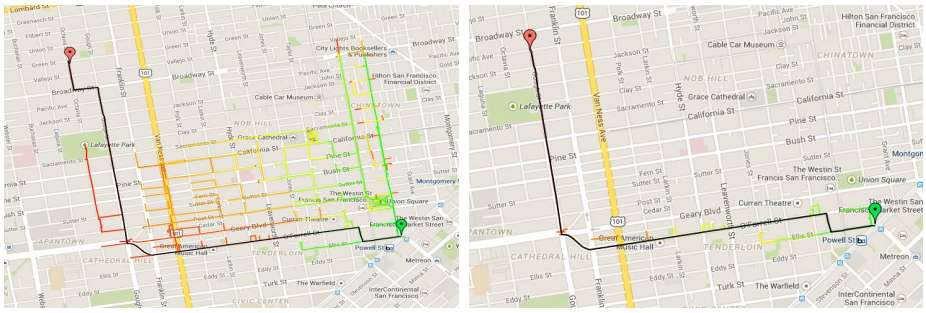


Figure 5.1: Problem Caused by Poor Landmark

In figure5.2, we assume that v is the current node while there are two child nodes y and w for it. Additionally, the equation holds that $c(v, w) = c(v, y)$ which means the cost from node v are same in these two children and we also assume that the distance between landmark and T is larger than that between landmark and node v. During the procedure of expansion, the system will calculate their $f - cost$ and extract the one with least value. By using the knowledge of geometry, we can conclude that the

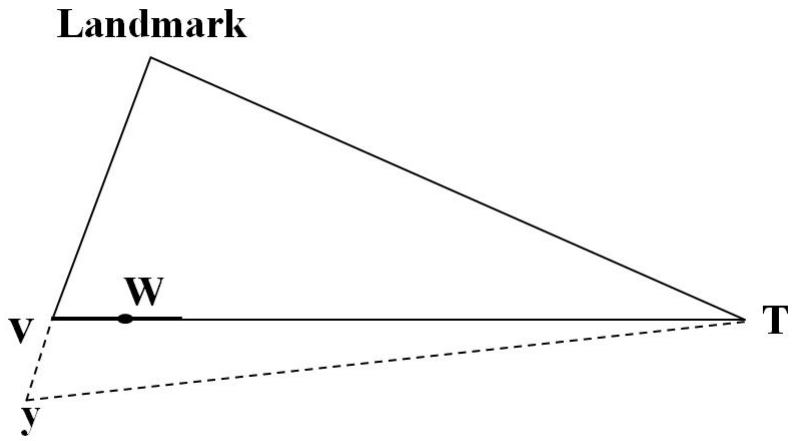


Figure 5.2: Searching with a Poor Landmark

searching will firstly expand node y instead of w, since node y has a lower $h()$ value than the other one. Consequently, it will be lead to a wrong way by poor landmark.

Initially, we define that there is a critical point in which the equation holds that $d(c, L) = d(L, t)$. It means that the best landmark for this point is the same close to current node and end node. In combination of this situation, we infer that these two nodes are located in different sides of the landmark, and the critical point may locate between these two nodes. As demonstrated in figure5.3, on the left side of critical point, searching will be lead along the path between node v and landmark, while on the other side searching will forward to a opposite direction. By analysing this phenomenon, we conclude that a good landmark should be located beside the end node. In other words, the landmark selection algorithm can be implemented in a circular area with the center of end node, and the radius is the distance between current node and end node. Otherwise, if there is not an appropriate landmark for current node, it will search along a wrong way.

5.3 Optimality

Apart from the performance, the admissibility which indicates whether the optimal solution can be returned would be considered in this project. As can be seen from the data, in Dijkstras algorithm, A* algorithm with Euclidean metrics and ALT would ensure the optimality. And we have already predicted this result in

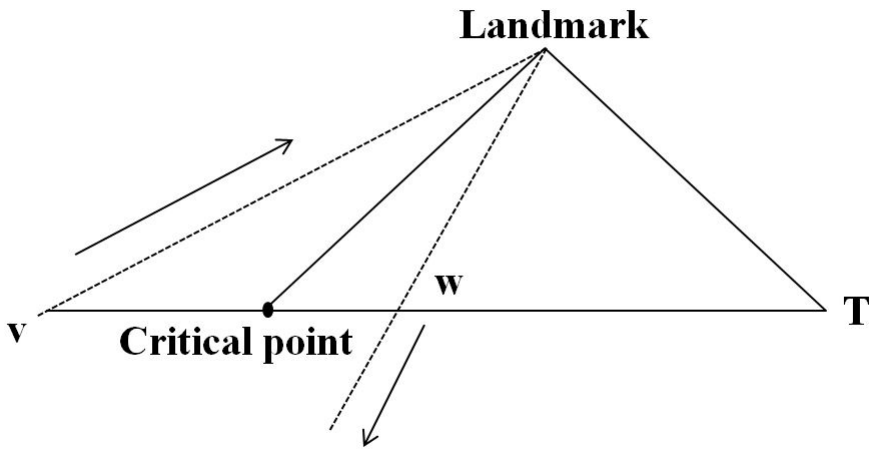


Figure 5.3: Landmark for Different Nodes

previous theoretical analysis. Actually, in a small searching space, the loss of optimality is not unacceptable since the algorithm would spend less time and storage on computing. However, when it comes to a long distance routing, the gap between optimal methods and un-optimal methods enlarges. As presented in scenario 3, the length of route returned by A* with Convergence is almost 500m longer than the optimal one. It seems that the gap increases as with the growth of searching scale. Consequently,those un-optimal approaches are not acceptable in real map routing, especially for long distance routing.

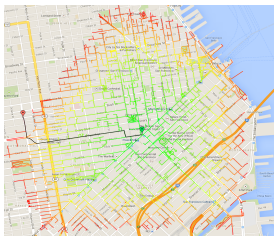


Figure 5.4: Dijkstra's in S.1

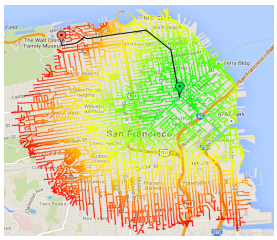


Figure 5.5: Dijkstra's in S.2

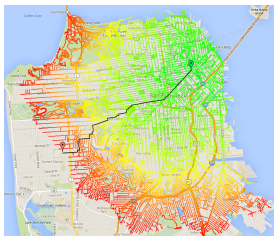


Figure 5.6: Dijkstra's in S.3

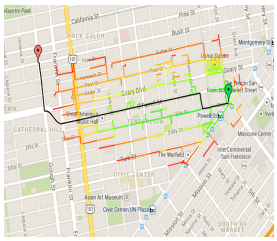


Figure 5.7: A* in S.1

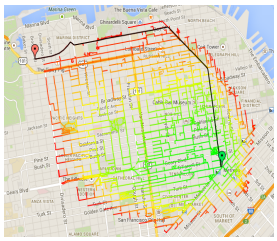


Figure 5.8: A* in S.2

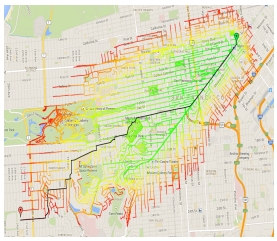


Figure 5.9: A* in S.3

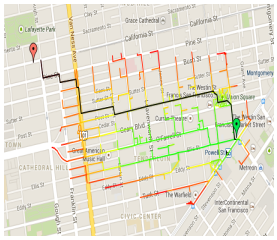


Figure 5.10: convergence in S.1

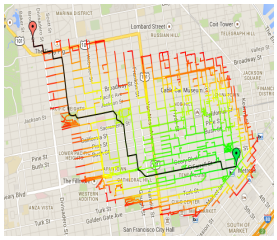


Figure 5.11: convergence in S.2

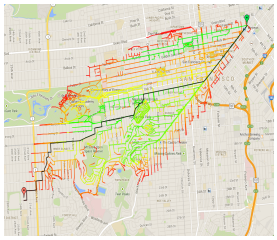


Figure 5.12: convergence in S.3

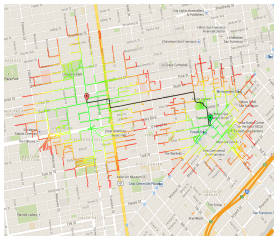


Figure 5.13: Bi-Dijkstra in S.1

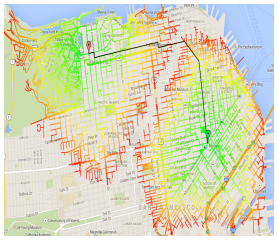


Figure 5.14: Bi-Dijkstra in S.2

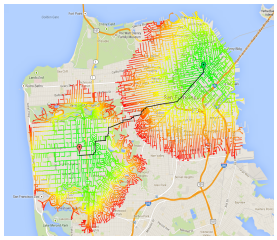


Figure 5.15: Bi-Dijkstra in S.3

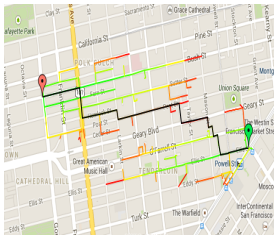


Figure 5.16: Bi-A* in S.1

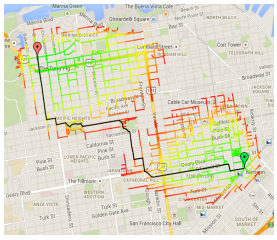


Figure 5.17: Bi-A* in S.2

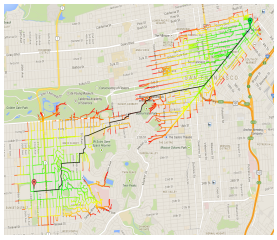


Figure 5.18: Bi-A* in S.3

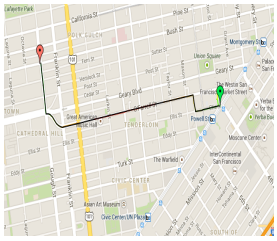


Figure 5.19: ALT in S.1

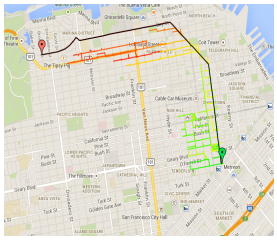


Figure 5.20: ALT in S.2

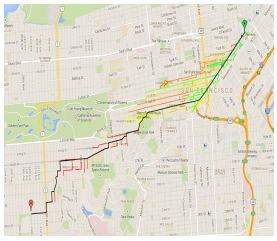


Figure 5.21: ALT in S.3

Chapter 6

Summary and Conclusion

Based on these results in simulations, we could obtain following conclusions.

Firstly, it is shown that dijkstras algorithm is not valuable any more in road network shortest path routing since it consumes too much time and memory. However, it is the foundation of path routing algorithms and it will play a key role in teaching. And then, A* algorithm made a huge improvements on dijkstras algorithm since it introduced heuristic approach which minimize the searching scale.

Based on these algorithms above, Bi-directional search brings a significant benefit on performance of routing while it does not damage the admissibility. As can be seen from the results, Bi-directional searches improves both Dijkstras algorithm and A* algorithm significantly. This consequence corresponds to the theoretical analysis given before. Additionally, for Bi-directional A* algorithm, consistent approach ensures optimality as we expected before. Next, the method of A* with convergence also improve the speed of routing by increase heuristic gradually while it cannot ensure optimality. Especially, when it serves for a long distance routing, the weight of heuristic will increase sharply, and it will turn un-optimal.

Above all, in terms of ALT algorithm, it searched least nodes and consumes least time among all the algorithms. By employing landmarks, it searches in a very small region and also gives the optimality which seems to be a perfect solution to shortest path routing problem. However, the issue of landmarks selection is of vital importance since it will impact the performance of ALT. By testing on a real map, we discovered some problems caused by this issue. In some situations, it will search along a wrong path, but it will find the correct direction later. We made a conclusion that this issue is related to the relative position among current node, landmark and end node. With the intention to improve this method, the landmarks selection algorithm can be modified to select landmarks beside the end node instead of within a large area. In this manner, landmarks are expected to lead the searching a more efficient way.

In summary, ALT is the best method among these algorithms since it searches less node and quicker. However, there are possibilities to improve this method by applying an advance landmarks selection algorithm.

References

[1] B. Liu, S.-H. Choo, S.-L. Lok, S.-M. Leong, S.-C. Lee, F.-P. Poon, and H.-H. Tan, "Integrating case-based reasoning, knowledge-based approach and dijkstra algorithm for route finding," in *Artificial Intelligence for Applications, 1994., Proceedings of the Tenth Conference on*, pp. 149–155, IEEE, 1994.

[2] S. Idwan and W. Etaiwi, "Dijkstra algorithm heuristic approach for large graph," *Journal of Applied Sciences*, vol. 11, pp. 2255–2259, 2011.

[3] S. Russell, P. Norvig, and A. Intelligence, "A modern approach," *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, vol. 25, 1995.

[4] Y. Björnsson, M. Enzenberger, R. C. Holte, and J. Schaeffer, "Fringe search: Beating a* at pathfinding on game maps.," *CIG*, vol. 5, pp. 125–132, 2005.

[5] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[6] I. Pohl, "Heuristic search viewed as path finding in a graph," *Artificial Intelligence*, vol. 1, no. 3, pp. 193–204, 1970.

[7] D. Xie, H. Zhu, L. Yan, S. Yuan, and J. Zhang, "An improved dijkstra algorithm in gis application," in *World Automation Congress (WAC), 2012*, pp. 167–169, IEEE, 2012.

[8] W. Zeng and R. Church, "Finding shortest paths on real road networks: the case for a*," *International Journal of Geographical Information Science*, vol. 23, no. 4, pp. 531–543, 2009.

[9] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *Systems Science and Cybernetics, IEEE Transactions on*, vol. 4, no. 2, pp. 100–107, 1968.

[10] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the ACM (JACM)*, vol. 34, no. 3, pp. 596–615, 1987.

[11] J. B. Kwa, "Bs¡sup¿¡/sup¿: An admissible bidirectional staged heuristic search algorithm," *Artificial Intelligence*, vol. 38, no. 1, pp. 95–109, 1989.

[12] T. Ikeda, M.-Y. Hsu, H. Imai, S. Nishimura, H. Shimoura, T. Hashimoto, K. Tenmoku, and K. Mitoh, "A fast algorithm for finding better routes by ai search techniques," in *Vehicle Navigation and Information Systems Conference, 1994. Proceedings., 1994*, pp. 291–296, IEEE, 1994.

[13] A. V. Goldberg and C. Harrelson, "Computing the shortest path: A search meets graph theory," in *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pp. 156–165, Society for Industrial and Applied Mathematics, 2005.

[14] D. L. Poole and A. K. Mackworth, *Artificial Intelligence: foundations of computational agents*. Cambridge University Press, 2010.

[15] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," *Artificial intelligence*, vol. 27, no. 1, pp. 97–109, 1985.

[16] L. Mero, "A heuristic search algorithm with modifiable estimate," *Artificial Intelligence*,

vol. 23, no. 1, pp. 13–27, 1984.

[17] A. Martelli, "On the complexity of admissible search algorithms," *Artificial Intelligence*, vol. 8, no. 1, pp. 1–13, 1977.

[18] R. J. Gutman, "Reach-based routing: A new approach to shortest path algorithms optimized for road networks.," in *ALENEX/ANALC*, pp. 100–111, 2004.

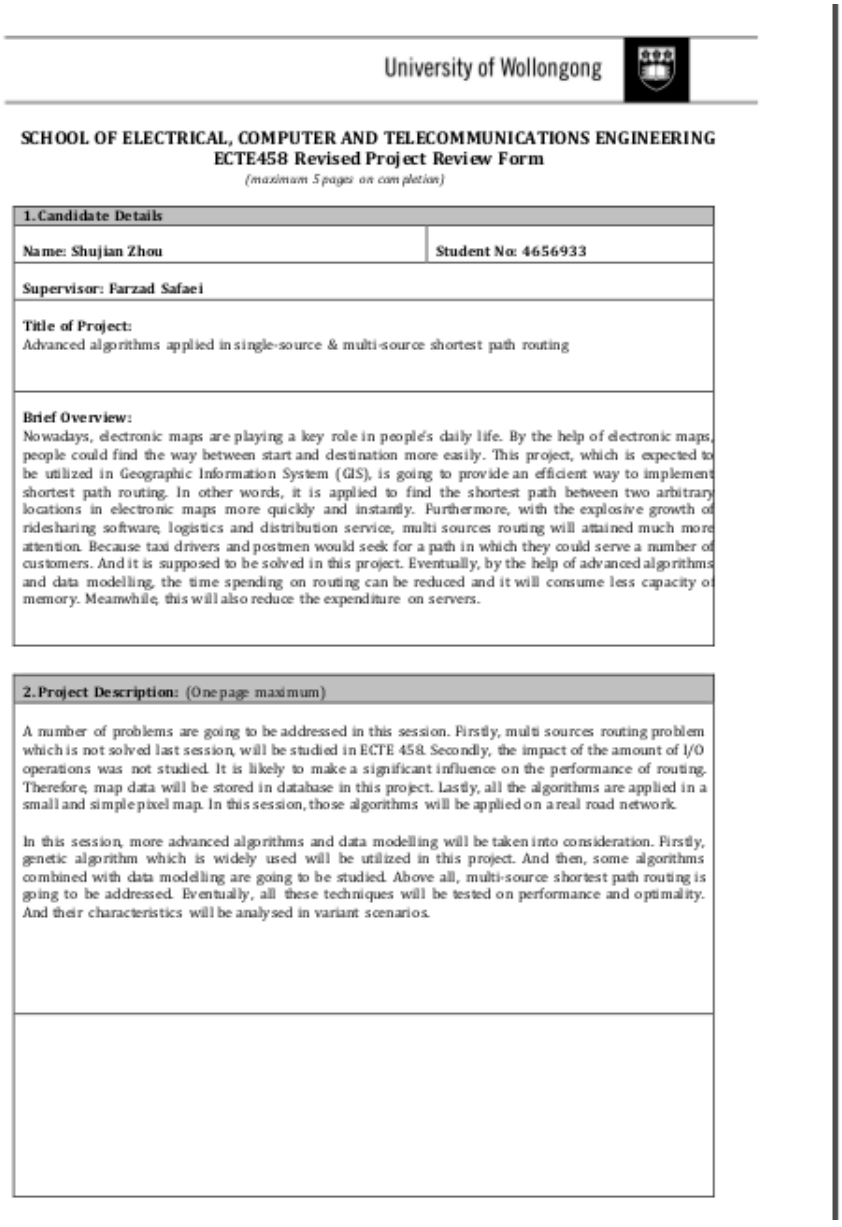
[19] A. V. Goldberg, H. Kaplan, and R. F. Werneck, "Reach for a*: Efficient point-to-point shortest path algorithms.," in *ALENEX*, vol. 6, pp. 129–143, SIAM, 2006.

[20] F. Fuchs, "On preprocessing the alt-algorithm," *Student thesis, Faculty of Computer Science, Institut for Theoretical Informatics (ITI), Karlsruhe Institute of Technology (KIT)*, 2010.

[21] R. W. Sinnott, "Virtues of the haversine," *Sky and telescope*, vol. 68, p. 158, 1984.

Appendix A

ECTE458 Revised Project Review Form



University of Wollongong

SCHOOL OF ELECTRICAL, COMPUTER AND TELECOMMUNICATIONS ENGINEERING
ECTE458 Revised Project Review Form
(maximum 5 pages on completion)

1. Candidate Details

Name: Shujian Zhou | **Student No: 4656933**

Supervisor: Farzad Safaei

Title of Project:
Advanced algorithms applied in single-source & multi-source shortest path routing

Brief Overview:
Nowadays, electronic maps are playing a key role in people's daily life. By the help of electronic maps, people could find the way between start and destination more easily. This project, which is expected to be utilized in Geographic Information System (GIS), is going to provide an efficient way to implement shortest path routing. In other words, it is applied to find the shortest path between two arbitrary locations in electronic maps more quickly and instantly. Furthermore, with the explosive growth of ridesharing software, logistics and distribution service, multi sources routing will attained much more attention. Because taxi drivers and postmen would seek for a path in which they could serve a number of customers. And it is supposed to be solved in this project. Eventually, by the help of advanced algorithms and data modelling, the time spending on routing can be reduced and it will consume less capacity of memory. Meanwhile, this will also reduce the expenditure on servers.

2. Project Description: (One page maximum)

A number of problems are going to be addressed in this session. Firstly, multi sources routing problem which is not solved last session, will be studied in ECTE 458. Secondly, the impact of the amount of I/O operations was not studied. It is likely to make a significant influence on the performance of routing. Therefore, map data will be stored in database in this project. Lastly, all the algorithms are applied in a small and simple pixel map. In this session, those algorithms will be applied on a real road network.

In this session, more advanced algorithms and data modelling will be taken into consideration. Firstly, genetic algorithm which is widely used will be utilized in this project. And then, some algorithms combined with data modelling are going to be studied. Above all, multi-source shortest path routing is going to be addressed. Eventually, all these techniques will be tested on performance and optimality. And their characteristics will be analysed in variant scenarios.

3. Project Plan: (Two pages maximum)

Detail your reviewed project plan. This should be a resilient engineering plan accommodating realistic alternatives and contingency measures to meet the objectives in this remaining session. Again, budget constraints should also be considered. Questions that you should answer are:

(a) What do you intend doing in this last session? Briefly review the methods that you will use to achieve the objectives stated above as well as the software and/or hardware that will be developed.

(b) Have any of the initial strategies proposed in the previous session changed? If so indicate with reference to the previous outcomes and any literature you have read so far.

(c) How do you intend to validate your solution/experimental results/simulations/procedures?

(d) Indicate all milestones and deliverables, to be achieved by the end of ECTE458.

(a) During the last session, those issues that were not addressed in ECTE451 are expected to be solved. Initially, the multi-source shortest path routing will be discussed and applied. After this, more advanced algorithms and data modelling will be tested and then improved probably. In this way, we could know the characteristics of these different methods and make a choice among them. Lastly, in terms of the visualization which intends to show the result intuitively, "pgrouting" and "Graphserver" would be used to achieve this purpose.

(c) As to validate the results, all the approaches would be applied on a real road map. Furthermore, this project is likely to be implemented on a large map in which the differences among those algorithms can be shown obviously.

(d)Planned milestones and outcomes are presented below:

Week / Task	1	2	3	4	5	6	7	8	9	10	11	12	13
Lecture review													
Research on algorithms													
Programming													
Testing & Debugging													
Final thesis report													
Demonstration and presentation													

4. Adaption of Supervisor and Examiners feedback in the ECTE451 report: (Half a page maximum)
First of all, my thesis did not show the depth of my knowledge. Because I just read some text books and journal articles and then paraphrasing them. In ECTE458, I will try to understand and explain concepts and theories in my way and use particular examples so that it can be easier for common people to. Secondly, as the first time to write a thesis, the format and layout of my report shows the lack of my experience on writing a thesis. Having learned LaTex and got comments on ECTE451, I am able to give a more professional report in this session.

5. Overall Project Proposal *(The overall project proposal should be assessed in terms of the feasibility and, aims. The scope of the project should be appropriate for the subject.)*	
Mark for Progress to Date.	8 /10
Mark for Overall project review (adaptation of project description and plan).	7 /10
Comments: The focus of the project for this session is not fully clear.	

Supervisor Signature		
	Signature	**Date**
Supervisor Name: F. Safaei		25/8/14

Student Signature ***Declaration by the student: I have understood the feedback provided to me by the supervisor.***		
	Signature	**Date**
Student Name: Shujian ZHOU		

Appendix B

Logbook Summary Signature Sheet

A Logbook Summary Signature Sheet

SCHOOL OF ELECTRICAL, COMPUTER AND TELECOMMUNICATIONS ENGINEERING

ECTE458 Thesis: Logbook Summary Signature Sheet

Week No.	Date	Comments, if applicable	Student's Signature	Supervisor's Signature
1	8/8/2014		Shujian Zhou	
2	15/8/2014		Shujian Zhou	
3	22/8/2014		Shujian Zhou	
4	22/8/2014		Shujian Zhou	
5	5/9/2014		Shujian Zhou	
7	12/9/2014		Shujian Zhou	
9	26/9/14		Shujian Zhou	
10	10/10/14		Shujian Zhou	

Appendix C

Load Map Data

```
import sys
import os
import tiledata
import tilenames
import re
import weights

class LoadOsm:
  def __init__(self, transport):
    self.routing = {}
    self.rnodes = {}
    self.transport = transport
    self.tiles = {}
    self.weights = weights.RoutingWeights()

  def getArea(self, lat, lon):
    z = tiledata.DownloadLevel()
    (x,y) = tilenames.tileXY(lat, lon, z)
    tileID = '%d,%d'%(x,y)
    if(self.tiles.get(tileID,False)):
      #print "Already got %s" % tileID
      return
    self.tiles[tileID] = True
    filename = tiledata.GetOsmTileData(z,x,y)
    #print "Loading %d,%d at z%d from %s" % (x,y,z,filename)
    return(self.loadOsm(filename))

  def loadOsm(self, filename):
    if(not os.path.exists(filename)):
      print "No such data file %s" % filename
      return(False)
    fp = open(filename, "r")
    re_way = re.compile("<way id='(\d+)'>\s*$")
    re_nd = re.compile("\s+<nd id='(\d+)' x='(\d+)' y='(\d+)' />\s*$")
    re_tag = re.compile("\s+<tag k='(.*)' v='(.*)' />\s*$")
    re_endway = re.compile("</way>$")
    in_way = 0

    way_tags = {}
    way_nodes = []

    for line in fp:
      result_way = re_way.match(line)
      result_endway = re_endway.match(line)
      if(result_way):
        in_way = True
```

```
47          way_tags = {}
48          way_nodes = []
49          way_id = int(result_way.group(1))
50        elif(result_endway):
51          in_way = False
52          self.storeWay(way_id, way_tags, way_nodes)
53        elif(in_way):
54          result_nd = re_nd.match(line)
55          if(result_nd):
56            node_id = int(result_nd.group(1))
57            x = float(result_nd.group(2))
58            y = float(result_nd.group(3))
59
60            (lat,lon) = tilenames.xy2latlon(x,y,31)
61
62            way_nodes.append([node_id,lat,lon])
63          else:
64            result_tag = re_tag.match(line)
65            if(result_tag):
66              way_tags[result_tag.group(1)] = result_tag.group(2)
67      return(True)
68
69    def storeWay(self, wayID, tags, nodes):
70      highway = self.equivalent(tags.get('highway', ''))
71      railway = self.equivalent(tags.get('railway', ''))
72      oneway = tags.get('oneway', '')
73      reversible = not oneway in('yes','true','1')
74      access = {}
75      access['cycle'] = highway in ('primary','secondary','tertiary','uncl
76      access['car'] = highway in ('motorway','trunk','primary','secondary'
77      access['train'] = railway in('rail','light_rail','subway')
78      access['foot'] = access['cycle'] or highway in('footway','steps')
79      access['horse'] = highway in ('track','unclassified','bridleway')
80
81      last = [None,None,None]
82
83      if(wayID == 41 and 0):
84        print nodes
85        sys.exit()
86      for node in nodes:
87        (node_id,x,y) = node
88        if last[0]:
89          if(access[self.transport]):
90            weight = self.weights.get(self.transport, highway)
91            self.addLink(last[0], node_id, weight)
92            self.makeNodeRouteable(last)
93            if reversible or self.transport == 'foot':
94              self.addLink(node_id, last[0], weight)
95              self.makeNodeRouteable(node)
96        last = node
97
```

```
98    def makeNodeRouteable(self,node):
99      self.rnodes[node[0]] = [node[1],node[2]]
100
101   def addLink(self,fr,to, weight=1):
102     """Add a routeable edge to the scenario"""
103     try:
104       if to in self.routing[fr].keys():
105         return
106       self.routing[fr][to] = weight
107     except KeyError:
108       self.routing[fr] = {to: weight}
109
110   def equivalent(self,tag):
111     """Simplifies a bunch of tags to nearly-equivalent ones"""
112     equivalent = { \
113       "primary_link":"primary",
114       "trunk":"primary",
115       "trunk_link":"primary",
116       "secondary_link":"secondary",
117       "tertiary":"secondary",
118       "tertiary_link":"secondary",
119       "residential":"unclassified",
120       "minor":"unclassified",
121       "steps":"footway",
122       "driveway":"service",
123       "pedestrian":"footway",
124       "bridleway":"cycleway",
125       "track":"cycleway",
126       "arcade":"footway",
127       "canal":"river",
128       "riverbank":"river",
129       "lake":"river",
130       "light_rail":"railway"
131       }
132     try:
133       return(equivalent[tag])
134     except KeyError:
135       return(tag)
136
137   def findNode(self,lat,lon):
138     """Find the nearest node that can be the start of a route"""
139     self.getArea(lat,lon)
140     maxDist = 1E+20
141     nodeFound = None
142     posFound = None
143     for (node_id,pos) in self.rnodes.items():
144       dy = pos[0] - lat
145       dx = pos[1] - lon
146       dist = dx * dx + dy * dy
147       if(dist < maxDist):
148         maxDist = dist
```

```
149             nodeFound = node_id
150             posFound = pos
151         #print "found at %s"%str(posFound)
152         return(nodeFound)
153 
154     def report(self):
155         """Display some info about the loaded data"""
156         print "Loaded %d nodes" % len(self.rnodes.keys())
157         print "Loaded %d %s routes" % (len(self.routing.keys()), self.transp
158 
159 # Parse the supplied OSM file
160 if __name__ == "__main__":
161     data = LoadOsm("cycle")
162     if(not data.getArea(-34.407900,150.885200)):
163         print "Failed to get data"
164     data.getArea(-34.407900,150.885200)
165     data.report()
166 
167     print "Searching for node: found " + str(data.findNode(-34.407900,150.
```

Routing Algorithm

```
1  try:
2      import simplejson as json
3  except ImportError:
4      import json
5  from heapq import heappush, heappop
6  from Quadtree import point_dict_to_quadtree
7  from collections import defaultdict
8  try:
9      # my C module for basic GIS stuff
10     from gisutil import haversine, bearing
11 except ImportError:
12     from util import haversine, bearing
13 
14 def find_closest_node(target, quadtree, rng=.01):
15     x, y = target
16     close_nodes = quadtree.query_range(x - rng, x + rng, y - rng, y +
   rng)
17     best_node = None
18     best_dist = float("inf")
19     for point, nodes in close_nodes.iteritems():
20         dist = haversine(point[0], point[1], target[0], target[1])
21         if dist < best_dist:
22             best_dist = dist
23             best_node = nodes
24     if best_node:
25         return best_node[0]
26     else:
27         return None
28 
```

```
def exact_dist(source, dest, graph, coords):
    dest_x, dest_y = coords[dest]
    pred_list = {source : 0}
    closed_set = set()
    unseen = [(0, source)]    # keeps a set and heap structure
    while unseen:
        _, vert = heappop(unseen)
        if vert in closed_set:
            # needed because we dont have a heap with decrease-key
            continue
        elif vert == dest:
            return pred_list[vert]
        closed_set.add(vert)
        for arc, arc_len in graph[vert]:
            if arc not in closed_set:
                new_dist = (pred_list[vert] + arc_len)
                if arc not in pred_list or new_dist < pred_list[arc]:
                    pred_list[arc] = new_dist
                    x, y = coords[arc]
                    heappush(unseen, (new_dist + haversine(x, y, dest_x,
    return None    # no valid path found

def lm_exact_dist(source, dest, graph, coords, lm_dists):
    """ Speed up by using already calculated euclidean distance. """
    lm_dists = lm_dists[dest]
    pred_list = {source : 0}
    closed_set = set()
    unseen = [(0, source)]    # keeps a set and heap structure
    while unseen:
        _, vert = heappop(unseen)
        if vert in closed_set:
            # needed because we dont have a heap with decrease-key
            continue
        elif vert == dest:
            return pred_list[vert]
        closed_set.add(vert)
        for arc, arc_len in graph[vert]:
            if arc not in closed_set:
                new_dist = (pred_list[vert] + arc_len)
                if arc not in pred_list or new_dist < pred_list[arc]:
                    pred_list[arc] = new_dist
                    heappush(unseen, (new_dist + lm_dists[arc], arc))
    return None    # no valid path found

def landmark_distances(landmarks, graph, graph_coords):
    lm_est = {}
    for lm_id, lm_coord in landmarks:
        lm_est[lm_id] = {}
        x,y = lm_coord
        for pid, coord in graph_coords.iteritems():
            lm_est[lm_id][pid] = haversine(x, y, coord[0], coord[1])
```

```
80      qtree = point_dict_to_quadtree(graph_coords, multiquadtree=True)
81      lm_dists = defaultdict(list)
82      l = len(graph_coords)
83      for i, pid in enumerate(graph_coords):
84          print i, '/', l, ':', pid
85          for landmark, _ in landmarks:
86              try:
87                  d = lm_exact_dist(pid, landmark, graph, graph_coords, lm
88              except KeyError:
89                  d = None
90              lm_dists[pid].append(d)
91      return lm_dists
92
93  def planar_landmark_selection(k, origin, coords, graph, qtree):
94      origin_id = find_closest_node(origin, qtree)
95      origin = coords[origin_id]
96      sectors = section_plane(k, origin, coords)
97      landmarks = []
98      for sector in sectors:
99          max_dist = float("-inf")
100         best_candidate = None
101         for pid, coord in sector:
102             cur_dist = haversine(origin[0], origin[1], coord[0], coord[1
103             if cur_dist > max_dist and exact_dist(pid, origin_id, graph,
104                 max_dist = cur_dist
105                 best_candidate = coord
106         landmarks.append(best_candidate)
107     return landmarks
108
109 def section_plane(k, origin, coords):
110     sectors = [[] for _ in xrange(k)]
111     sector_size = 360.0 / k
112     for pid, coord in coords.iteritems():
113         b = bearing(origin[0], origin[1], coord[0], coord[1], True)
114         s = int(b / sector_size)
115         sectors[s].append((pid, coord))
116     return sectors
117
118 def farthest_landmark_selection(k, origin, coords):
119     landmarks = [origin]
120     for _ in xrange(k):
121         max_dist = float("-inf")
122         best_candidate = None
123         for pid, coord in coords.iteritems():
124             cur_dist = 0
125             for lm in landmarks:
126                 cur_dist += haversine(lm[0], lm[1], coord[0], coord[1])
127             if cur_dist > max_dist and not coord in landmarks:
128                 max_dist = cur_dist
129                 best_candidate = coord
130         landmarks.append(best_candidate)
```

```
131             if len(landmarks) > k:
132                 landmarks.pop(0)
133         return landmarks
134 
135     def load_graph():
136         with open('sf.j', 'r') as fp:
137             graph = json.loads(fp.read())
138         with open('sf_coords.j', 'r') as fp:
139             graph_coords = json.loads(fp.read())
140         return graph, graph_coords
141 
142 
143     from multiprocessing import Queue, Process, Value
144     import math
145 
146     def mp_landmark_distances(landmarks, graph, graph_coords, nprocs):
147         def worker(landmarks, graph, graph_coords, pids, out_q, counter):
148             """ The worker function, invoked in a process. 'nums' is a
149                 list of numbers to factor. The results are placed in
150                 a dictionary that's pushed to a queue.
151             """
152             lm_est = {}
153             for lm_id, lm_coord in landmarks:
154                 lm_est[lm_id] = {}
155                 x,y = lm_coord
156                 for pid, coord in graph_coords.iteritems():
157                     lm_est[lm_id][pid] = haversine(x, y, coord[0], coord[1])
158             qtree = point_dict_to_quadtree(graph_coords, multiquadtree=True)
159             lm_dists = defaultdict(list)
160             l = len(graph_coords)
161             for pid in pids:
162                 counter.value += 1
163                 print counter.value, '/', l, ':', pid
164                 for landmark, _ in landmarks:
165                     try:
166                         d = lm_exact_dist(pid, landmark, graph, graph_coords
167                     except KeyError:
168                         d = None
169                     lm_dists[pid].append(d)
170             out_q.put(lm_dists)
171 
172         # Each process will get 'chunksize' nums and a queue to put his out
173         # dict into
174         counter = Value("i")
175         all_pids = graph_coords.keys()
176         out_q = Queue()
177         chunksize = int(math.ceil(len(all_pids) / float(nprocs)))
178         procs = []
179 
180         for i in range(nprocs):
181             chunk = all_pids[chunksize * i:chunksize * (i + 1)]
```

```
182         p = Process(
183                 target=worker,
184                 args=(landmarks, graph, graph_coords, chunk, out_q, coun
185             )
186         procs.append(p)
187         p.start()
188 
189     # Collect all results into a single result dict. We know how many di
190     # with results to expect.
191     resultdict = {}
192     for i in range(nprocs):
193         resultdict.update(out_q.get())
194 
195     # Wait for all worker processes to finish
196     for p in procs:
197         p.join()
198     return resultdict
199 
200 def main():
201     graph, graph_coords = load_graph()
202     origin = 37.772614,-122.423798
203     k = 16
204     qtree = point_dict_to_quadtree(graph_coords, multiquadtree=True)
205     landmarks = planar_landmark_selection(k, origin, graph_coords, graph
206     # create a quadtree that can hold multiple items per point
207 
208     landmarks = zip([find_closest_node(l, qtree) for l in landmarks], la
209     lm_dists = mp_landmark_distances(landmarks, graph, graph_coords, 8)
210     with open('lm_dists_2.j', 'w') as fp:
211         fp.write(json.dumps(lm_dists))
212 
213 if __name__ == '__main__':
214     main()
215     # import cProfile
216     # cProfile.run('main()', sort=1)
```