SIMULATION OF PATH OPTIMISATION ALGORITHMS

SIM CHOON YEE

UNIVERSITI SAINS MALAYSIA

2017

SIMULATION OF PATH OPTIMISATION ALGORITHMS

by

SIM CHOON YEE

Thesis submitted in fulfilment of the requirements
for the degree of
Bachelor of Electronic Engineering

JUNE 2017

ACKNOWLEDGEMENT

Firstly, I will like to express my utmost gratitude to Dr. Nur Syazreen Ahmad, my thesis advisor, and project supervisor, for providing me guidance and advice time to time until the stage of the completion of this thesis. Her guided direction and constant discussion have allowed this thesis to be completed successfully.

Secondly, I will like to thank to the School of Electrical and Electronic Engineering for giving me this opportunity to apply the engineering knowledge in this thesis and future work.

Lastly, my gratitude is extended to my family and friends, who have supported all the way through this academic life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ALGORITMAS SIMULASI OPTIMASI RUTE PERJALANAN

ABSTRAK

Matlamat kertas kerja ini adalah untuk mencari dan mengira jarak yang singkat dari satu nod yang lain dengan cara yang paling berkesan. 3 algoritma (algoritma A *, algoritma Dijkstra dan algoritma Greedy) dan 2 konsep (Pencarian Dua Arah dan perancangan semula) akan diterokai dan simulasi. Penanda aras untuk perbandingan keberkesanan setiap algoritma dengan teliti ditakrifkan dengan niat mencari algoritma yang terbaik dalam persekitaran yang direka. Semua simulasi Matlab dilakukan dalam 2D, 300x300 piksel imej dengan hanya satu permulaan yang tetap dan nod akhir. Bagi kaedah Dua Arah Mencari, ia ditetapkan untuk berhenti apabila kawasan pencarian dari titik permulaan dan kawasan mencari titik akhir bertemu. Selain itu, bagi maksud simulasi menggunakan konsep perancangan semula, peta asal akan dimasukkan dengan 4 dinding dan 39 Ruang putih. Sebanyak 4 eksperimen akan dilakukan untuk membandingkan sama ada 2 konsep digabungkan sebenarnya meningkatkan prestasi pencarian 3 algoritma itu, iaitu Pencarian Satu Arah tanpa perancangan semula, Pencarian Satu Arah dengan perancangan semula, Pencarian Dua Arah tanpa perancangan semula dan Pencarian Dua Arah Mencari dengan perancangan semula. Setiap keputusan akan dimasukkan dengan perbincangan tentang persamaan dan perbezaan keputusan setiap algoritma. Secara ringkasnya, jumlah Nod Dicari untuk Pencarian Dua Arah dengan perancangan semula mempunyai pengurangan keseluruhan (28%, 37% dan 51%) berbanding jumlah Nod Dicari untuk Pencarian Dua Arah tanpa perancangan semula. Di samping itu, jumlah masa algoritma ditunjukkan mempunyai pengurangan sebanyak 73,69%, 64% dan 4.62% berdasarkan simulasi peta kedua. Begitu juga jika dibandingkan antara Pencarian Dua Arah dengan perancangan semula dan Pencarian Satu Arah dengan perancangan semula, penurunan sebanyak 41.08%, 25.57% dan 14.41% ke atas jumlah masa algoritma dapat dilihat. Oleh itu, dalam projek tahun akhir ini, algoritma optimum untuk digunakan adalah A * algoritma dengan kaedah Cari dwiarah dan kaedah perancangan semula termasuk semasa proses pencarian. Kod lengkap termasuk dalam Lampiran untuk tujuan kebolehulangan.

SIMULATION OF PATH OPTIMISATION ALGORITHMS

ABSTRACT

The main concern is to find and calculate the shortest distance from one node to another in the most efficient way. 3 algorithms (A* algorithm, Dijkstra algorithm and Greedy algorithm) and 2 concepts (Bidirectional Searching and Replanning) are explored and simulated. The benchmark for comparing the effectiveness of each algorithm are thoroughly defined with the intention of finding the best algorithm in the designed environment. All Matlab simulations are done in a 2D, 300x300 image pixel with only a fixed start and an end node. For the Bidirectional Searching method, it is set to stop when the searching area from starting point and the searching area of end point are met. Next, for the purposes of simulating using Replanning concept, the original map is introduced with 4 walls and 39 whitespaces. A total of 4 experiments are done to compare whether the 2 concepts combined do actually improve the searching performance of the 3 algorithms, namely, Unidirectional Searching without Replanning, Unidirectional Searching with Replanning, Bidirectional Searching without Replanning and Bidirectional Searching with Replanning. Each result of the simulations is included with a discussion of the similarities and difference of the results of each algorithm. In summary, the total Nodes Searched of Bidirectional Searching with Replanning has an overall reduction (28%, 37% and 51%) when compared to total Nodes Searched of Bidirectional Searching without Replanning. In addition, the algorithm total time is shown to have decrement of 73.69%, 64% and 4.62% based on the second map simulations. Likewise, when comparing between Bidirectional Searching with Replanning and Unidirectional Searching with Replanning, a decrease of 41.08%, 25.57% and 14.41% on algorithm total time can be seen. Thus, in this final year project, the optimum algorithm to use is A* algorithm with Bidirectional Searching method and Replanning method included during the searching process. The complete code is included in Appendix for reproducibility purposes.

1 INTRODUCTION

1.1 Overview

The shortest problem is essentially how to calculate the shortest distance from one place to another. The problem itself normally is represented in the form of a graph as an input[1].

Generally, before the searching for shortest distance, a representation of graphical input need to be called upon. Most of the graph can be represented by a collection of 'nodes' and 'edges'. The 'nodes' are represented as physical locations. It can be squared, grid, circles. The lines that connect those nodes are called 'edges', which has a number, and is called edges' weight. The number is a determining factor of how expensive it is to walk or cross that edge. In the same analogy of the actual world, it can be roads connecting to different locations and the numbers are how long it took to drive on that particular road from one place to another.

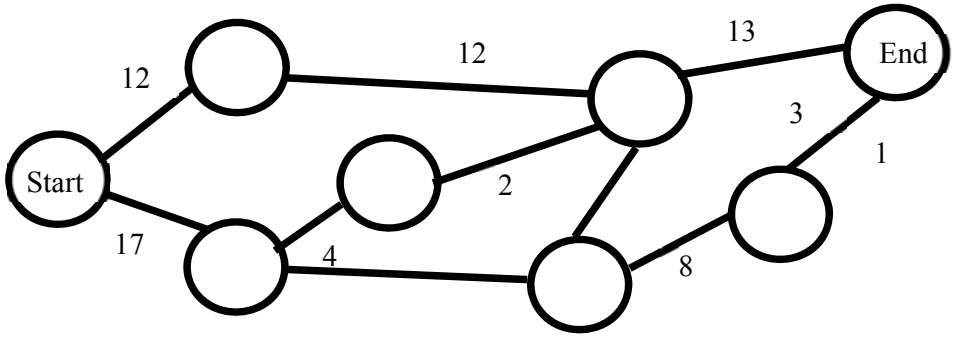

Figure 1-1 Example of nodes and lines (self-drawn)

Pathfinding begins by searching the graph from one place (starting point known as vertex) to another neighbouring node. Given the coordinates or locations of the starting vertex and final vertex, different strategies can be implemented to link the destination. Each of the edges can be either bidirectional or unidirectional.

1.2 Problem Statement

The primary interest in this project is focused on finding the shortest path in the smallest amount of time. However, in reality, in order to find the ideal path, multiple algorithms are needed and applied in finding the shortest path. The real challenge of this

project lies in deciding the optimum pathfinding method and improve A*, Dijkstra and Greedy algorithms.

The optimum way of finding the shortest problem needs to be analysed from different degrees of complexity, ranging from single node source to single node destination, single node source to multiple nodes destination and vice versa and multiple nodes sources to multiple nodes destination.

1.3 Objective

Based on a few main concern of the problem, the aim of this final year project is to simulate different search technique by using edges' information to find the fastest route towards the destination in an efficient way. In pursuance of optimising the most efficient path, comparison and findings for the best-optimised method are taken into account in this project.

1.4 Scope of Project

The project is aimed to test out 3 common pathfinding algorithms in a 20x20 grid which are as the following:

- Greedy Best First Search
- A* pathfinding algorithm
- Dijkstra's Algorithm

Next, the 3 algorithms are incorporated with concepts of Bidirectional Searching and Replanning to show if there is any improvement of the algorithm performance. Comparison of the 3 algorithms performance is needed to be done to show the best pathfinding algorithm in different situations. [2]

Mainly, the judging criteria are as follows, taking account length as the cost to travel:

- The total number of times each node was visited during the sequence of the path calculation including repeated visit.
- Successful of linking begin to end.
- Total number of nodes travelled from the starting node to ending node.
- Total time needed to simulate the algorithm
- Number of nodes that are needed to store for the next search (Replanning purposes)
- Number of iterations run during the execution of the code

Results are expected to be gathered and further improvement can be possibly made by tweaking the original algorithms.

1.5 Thesis Organisation

The thesis of this project is organised as such. Chapter 2 is presented with related previous work done by past researchers. Literature review and history on the following above strategies are also included. Chapter 3 is encompassed with methodologies on showing possible path using pathfinding algorithms. Chapter 4 is presented with the results and discussion for the simulations. The conclusion and future work is included in Chapter 5.

2 LITERATURE REVIEW

2.1 Introduction

In this literature review, previous work of researchers and improvement are stated in here. Some explanation of how the algorithm works, its advantage and disadvantage are also included.

2.2 Background Study On Breadth First Algorithm (Queue)

To begin with pathfinding algorithm, Breadth First search is one of the Search algorithm used to find all nodes in the graph. It starts at the root or first vertex and explore second level nodes before moving into further into third-level nodes.[3]1

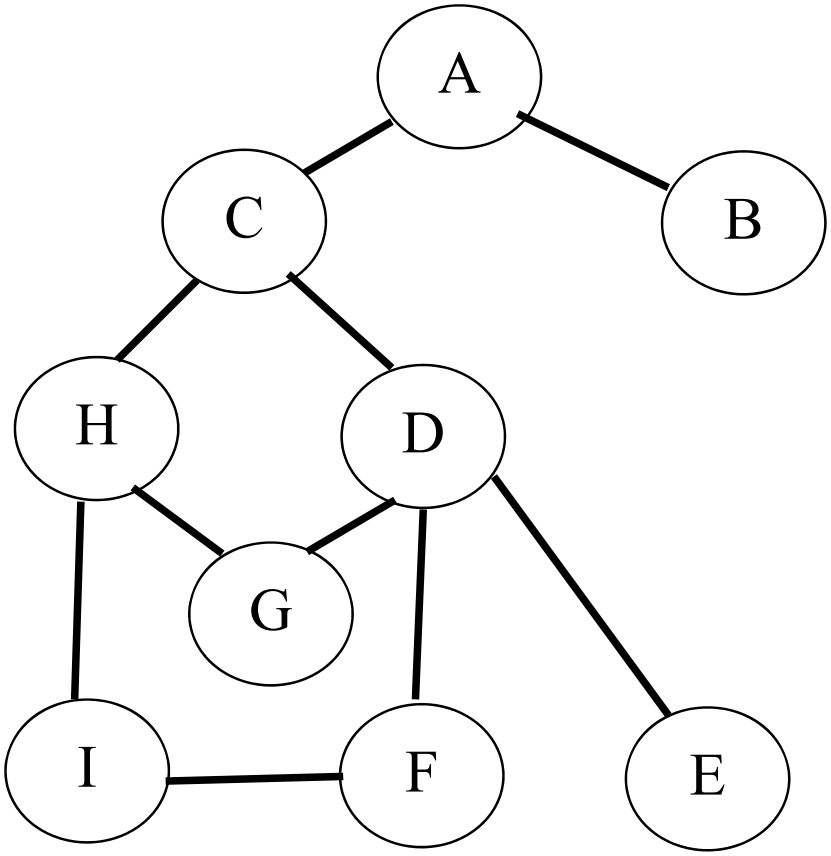

Figure 2-1Breadth First Search Example

In this example, the Breadth First Search has a Q (queue) containing nodes to be searched. The starting node, in this case will be node A and it is in the queue. For Breadth First Search, the neighbouring nodes are found which are node B and node C. Since node

A has no more expanding nodes besides node B and C, it is taken out from the queue. The output result is A->B->C.

Following the alphabetical order, neighbouring nodes of B node is explored. However, node B does not contain any neighbouring nodes. Then, it is discarded from the queue. Node C has neighbouring node D and node H. Node C is also discarded while node D and node H is included in the queue. The current output result is A->B->C->D->H.

After that, the node D is analysed in the queue. Neighbouring nodes of node D which are node E,F and G is included in the queue while node D is discarded. The current queue is sorted as following: H->E->F->G. The output result is A->B->C->D->H->E->F->G.

Next, the node H is analysed in the queue. Neighbouring nodes of node H which is only node I is included in the queue while node H is discarded. The current queue is sorted as following: E->F->G->I. The output result is A->B->C->D->H->E->F->G->I.

Finally, node E,F,G and I do not has any unexplored nodes. Therefore, the searching has stop and each node from the queue is discarded. All nodes in the graph is found.

2.3 Background Study of Depth First Search Algorithm (Stack)

For the method of Depth First Search Algorithm, the way of traversing graphs is to go as deep as possible before backtracking. A root vertex is needed as the place to begin the searching. However, it is different from Breadth First Search, it will move along the current path until all nodes are searched. If none of the nodes is available, backtrack from the same path until all there are available nodes.[4]

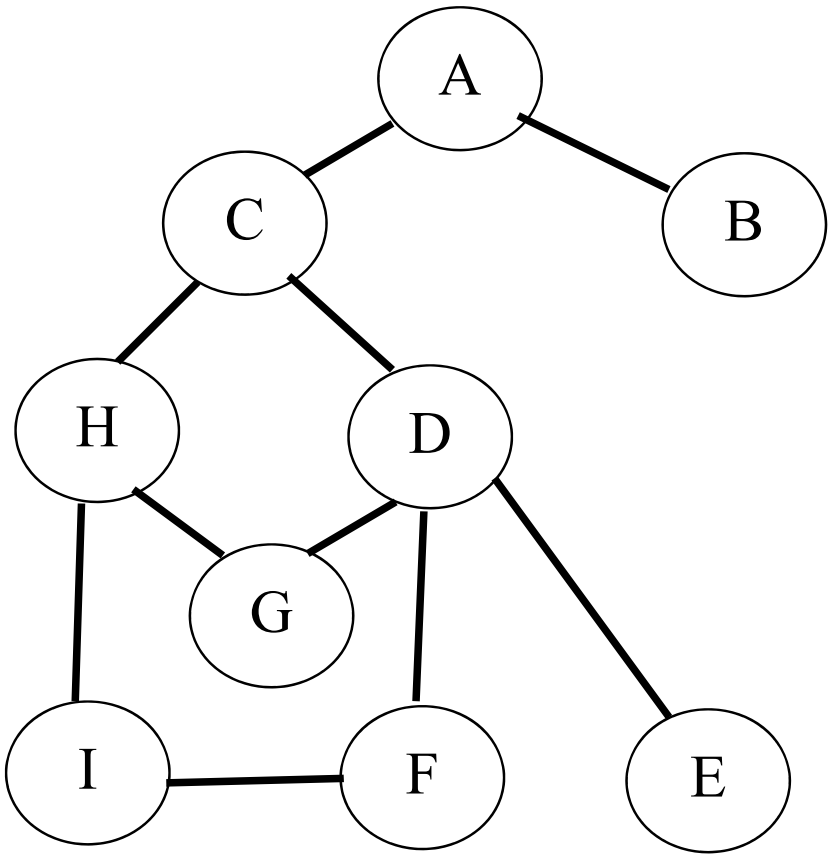

Figure 2-2 Depth First Search Example

The basic idea of tracking the nodes in Depth First Search is using Stack. In this example, node A is picked as the starting node and node A is pushed into Stack. Then, node B is pushed into the Stack. However, node B does not have any neighbouring nodes. Pop B from Stack. The output result is A->B.

Next, backtrack to node A. Now, node C is pushed into Stack. Following the alphabetical order, node D is pushed into stack followed by node E. The output result is

A->B->C->D->E. Node E does not have any neighbouring nodes so it is pop from the Stack. Backtracked to node D, node F is pushed into Stack. Next, it will be the following order: node I, node H and node G. The output result is A->B->C->D->E ->F->I->H->G. Since after node G, there are no neighbouring nodes, it is pop from the stack, continue with node H, node I, node F, node D, node C and node A.

Finally, all nodes are visited and the Depth First Search may be considered completed.

2.4 Background Study On Dijkstra's Algorithm

Dijkstra's algorithm works by prioritising the lowest cost to find the shortest path. It is done by choosing the vertex nearest to the current vertex. Formulated by Dijkstra[5], it is started by adding all vertices into a list. If the list is not empty, continue searching the minimum cost of the vertex and add it to the closed list. After that, the neighbouring vertex surrounding the newest vertex need to be updated of its cost. Continue the steps until the final goal is found.[6]

```
1:  function Dijkstra(Graph, source):
2:      for each vertex v in Graph:                 // Initialization
3:          dist[v] := infinity                     // initial distance from source to vertex v is set to infinite
4:          previous[v] := undefined                // Previous node in optimal path from source
5:      dist[source] := 0                           // Distance from source to source
6:      Q := the set of all nodes in Graph          // all nodes in the graph are unoptimized - thus are in Q
7:      while Q is not empty:                       // main loop
8:          u := node in Q with smallest dist[ ]
9:          remove u from Q
10:         for each neighbor v of u:               // where v has not yet been removed from Q.
11:             alt := dist[u] + dist_between(u, v)
12:             if alt < dist[v]                    // Relax (u,v)
13:                 dist[v] := alt
14:                 previous[v] := u
15:     return previous[ ]
```

Figure 2-3 Showing the pseudocode of Dijkstra algorithm*[6]*

To reduce the implementation time of Dijkstra's algorithm, it is suggested that to use Decrease-key method in the algorithm. [7] Instead of containing all vertices in the grid itself, the Decrease-key method only includes the new vertices when the algorithm explore through the grid. If the minimum cost of the vertex is in the included list, then it can be discarded from the list and treated as one of the closed-vertex.

There is another variant of Dijkstra's algorithm which is UCS (Uniform Cost Searching). Instead of loading all vertices and search through each node, UCS is one of the best-first search schemes that is similar to Dijkstra's algorithm. The difference in UCS is that it stops searching once the final goal is reached during the exploration of nodes. The route found for UCS might not be the shortest path. However, it has significant advantages over Dijkstra's algorithm which are in the pseudo code, in its time and memory needs and behaviour in practice according to [8].

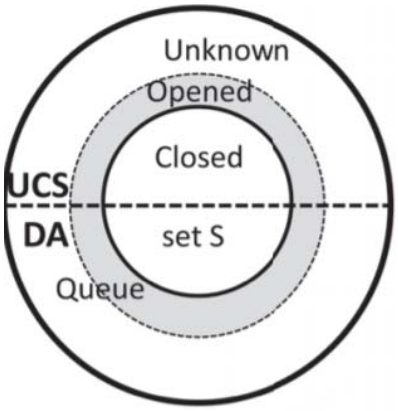

Figure 2-4 Showing UCS is the subset of Dijkstra

From the figure above, all the unknown nodes in Q (Queue in Dijkstra) are $=\infty$. The opened nodes (which are neighbouring nodes of closed nodes) is $\neq\infty$. The closed nodes in UCS is S which is the same as in Dijkstra.

In terms of improving the algorithm through hardware, a parallel analysis has done before in terms of serial and parallel execution.[9] The method of UCS is shown as below:

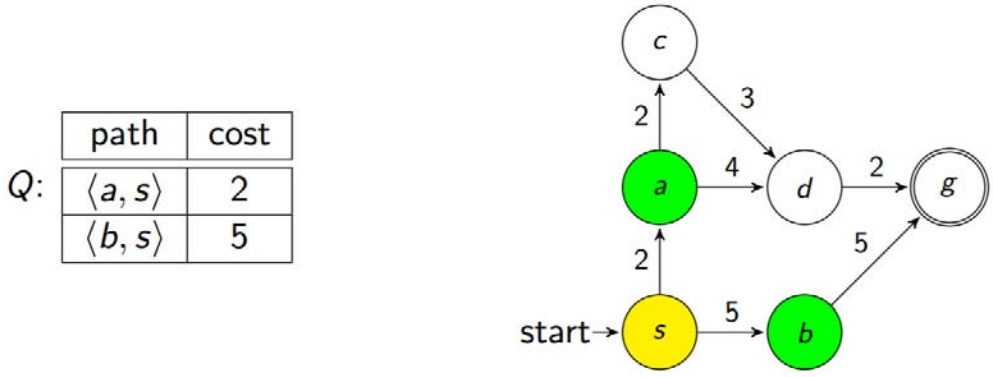

Figure 2-5 UCS algorithm: Concerning about the edges cost

2.5 Background Study on Greedy Algorithm

In the case of the Greedy algorithm, it has the similar concept as Dijkstra's algorithm. However, instead of choosing the shortest path based on the lowest cost incurred for travelling through edges, its decision is based on an estimate called heuristic to determine whether it should include the node. As long as the cost of any vertex found is nearer to the final goal, the vertex will be chosen as the next vertex. This means that Greedy algorithm does not always find the shortest path.

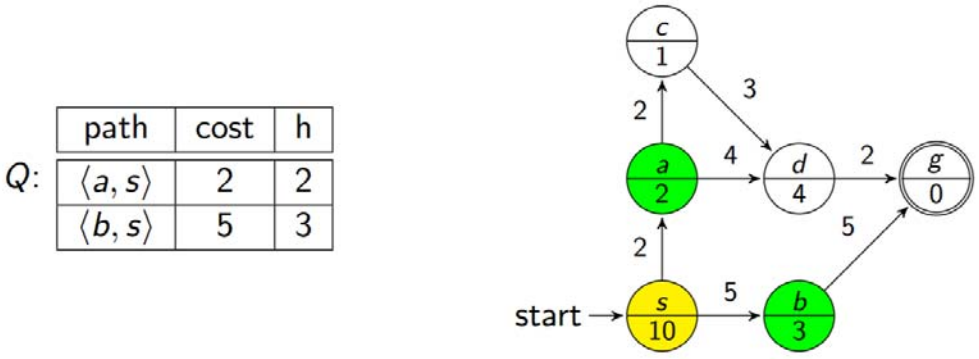

Figure 2-6 Greedy Algorithm: Decision making based on Heuristic score

For this algorithm, the edges cost is ignored. Instead, the algorithm will choose the shortest path based on the h value. As long as the goal target (node g in this case) is in the open list, then the algorithm is considered completed.

2.6 Background Study On A* Algorithm

Generally, for A* algorithm, there are 2 costs are involved in calculating the cost of each node in the grid: G cost, the cost incurred for current tiles or nodes from the starting point and H cost, that can be represented as cost calculated from current nodes to goal. [10]

$$\text{F (total cost)} = G + H$$

Equation 2-1 A* algorithm node cost

To calculate the cost for each tile, mostly the tiles for left, right, up and down are treated as 1 unit while diagonal tiles are treated as 1.4. Diagonal tiles' cost actual value is$\sqrt{2}$, but for simplicity, the value is treated as 1.4. The cost can be later multiplied into the unit to get the desired cost of each tile.

1.4	1.0	1.4
1.0	S	1.0
1.4	1.0	1.4

Figure 2-7 Cost of each tile (self-drawn)

From previous research, the standard way of G cost is calculated is summing up the cost of each tile using the lowest cost route. For H cost, multiple strategies can be used to calculate it.

2.7 Varying Method Of Calculating H (Heuristic) Costs In A* Algorithm

The standard way of calculating heuristic costs is Manhattan method which allows computing of heuristic costs without any diagonal.

$$h = 10(any\ cost\ desired) * (abs(currentX - finalX) + abs(currentY - finalY))$$

Equation 2-2 Manhattan method for H cost

Another featured method of calculating is the Diagonal shortcut method. With a little slower than Manhattan method, it is more balanced in the sense that it considers the diagonal cost. The equation is:

$$\text{xDistance} = \text{abs}(\text{currentX} - \text{targetX})$$
$$\text{yDistance} = \text{abs}(\text{currentY} - \text{targetY})$$

```
if (xDistance>yDistance)
{
h = 14*yDistance + 10*(xDistance-yDistance)
}
else h = 14*xDistance + 10*(yDistance-xDistance)
end if
```

Equation 2-3 Diagonal shortcut method

The third method is using the Chebyshev distance as calculating heuristic cost. It takes account that the diagonal distance is 1 unit instead of 1.4. It is used in chess AI where the piece 'King' moves in such way where all cost surrounding it is the same.[11]

1.0	1.0	1.0
1.0	S	1.0
1.0	1.0	1.0

Equation 2-4 Chebyshev

Another method is using Euclidean Distance which is similar to Pythagorean Theorem.

$$\text{xDistance} = \text{abs}(\text{currentX} - \text{targetX})$$
$$\text{yDistance} = \text{abs}(\text{currentY} - \text{targetY})$$

$$h = \sqrt{xDistance^2 + yDistance^2}$$

Figure 2-8 Euclidean Distance

However, A* method does not always provide the shortest path in a realistic graph. If the movements of the nodes are not entirely constrained on a grid, which means diagonal path with different angles are possible, then Basic Theta * can be used for this purposes. A comparison between these two algorithms have been made from previous research and it is found out that Basic Theta* algorithm is slightly more advantageous for shortest route.[12].

In order to improve A* algorithm, the whitespace between grid is needed to be passable from vertex to another vertex. [13]

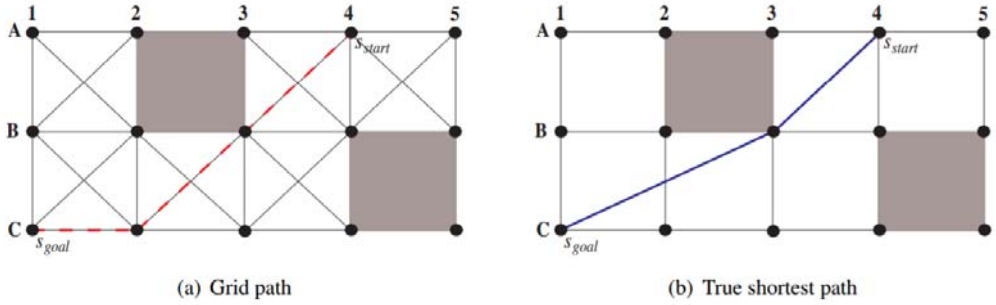

Figure 2-9 shows that (left) is the shortest path for A* while (right) is the shortest path without limiting the angle of reaching each vertex, namely A*PS method

First, A* algorithm can be further improved with post smoothing path. A* with Post-smoothed Paths (A*PS) [13]. It can be described as collecting all vertices found during the A* algorithm and trying to connect those vertices that are in line of sight of the current vertices. Therefore, in this method, the lines can be connected passing through the whitespace between the vertices.

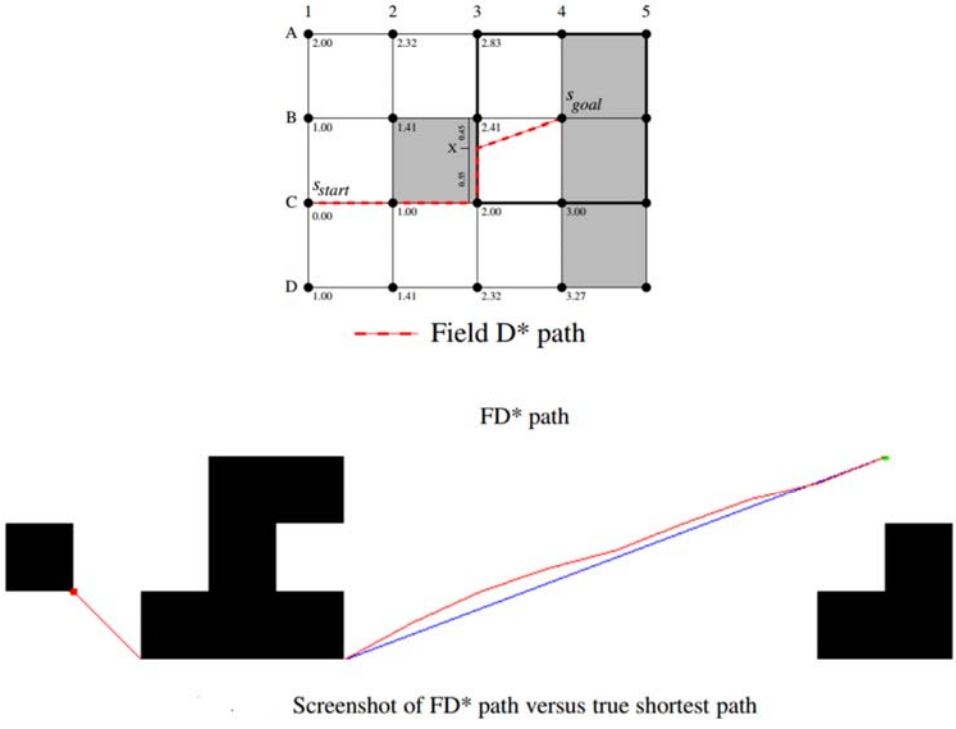

Figure 2-10 Field D* method showing that the bloackage of FD* method. (top) line is FD*path while blue line is the shortest path

Secondly, the A* can be also possibly improved by FD*(Field D*) pathfinding method. Basically, the method itself consider the value between edges to connect vertex. However, there might be some mistake in this method as blockage cell might make them longer resulting non-true shortest path.

Another way to improve the A* algorithm bidirectional searching is to implement parallel bidirectional searching by using multicores in the computer itself, namely using 2 cores.[14] This method is based on during the bidirectional pathfinding process, the start node and end node has their own cores running simultaneously instead using one core which is using shared memory to calculate the shortest distance. Two threading is created for this parallel approach and is stopped when both nodes met.

In pathfinding, there are plenty of examples of using A* algorithm. In games, there have been studies showing how to control the AI players to move in the fastest way possible.[15] A-star algorithm is one of the common algorithm in finding the shortest path in maps. Previously there have been works done in Unity 3D to showcase the functionality of A-star in games. [10]

2.8 Summary of Algorithm

The whole literature review is based on Graph Searching Algorithm an A* path searching variation. Figure 2-11 and Figure 2-12 shows the overall algorithms that are discussed in the literature review.

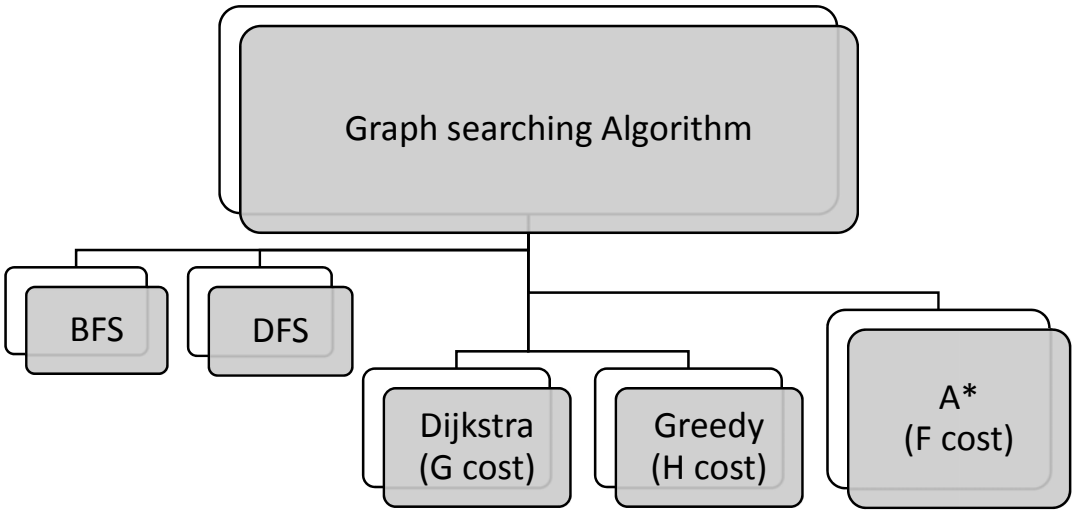

Figure 2-11 Overall concept of path searching

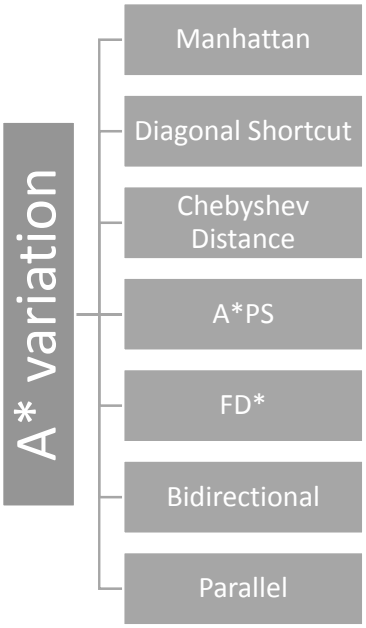

Figure 2-12 Different type of A* variation

2.9 Summary of Literature Review

After understanding of past research done, the main purpose of this literature review is to act as a guideline direction for this thesis.

3 METHODOLOGY

3.1 Introduction

The whole project is divided into 3 parts. At the beginning, the first step is the need to collect data, specify the range of data and determine the field of data the author will like to work on. After we have initialised the input, there is a need to specify the number of methods to be used, memory space for each method, variation of each pathfinding method. Then, a comparison of different methods is needed to be done which has certain detail criteria. Furthermore, limitation of each method is needed to be shown and the solution suggested to overcome it.

3.2 Project Implementation Flow

Figure 3-1 shows the overall flow of the whole process of simulating the pathfinding algorithm.

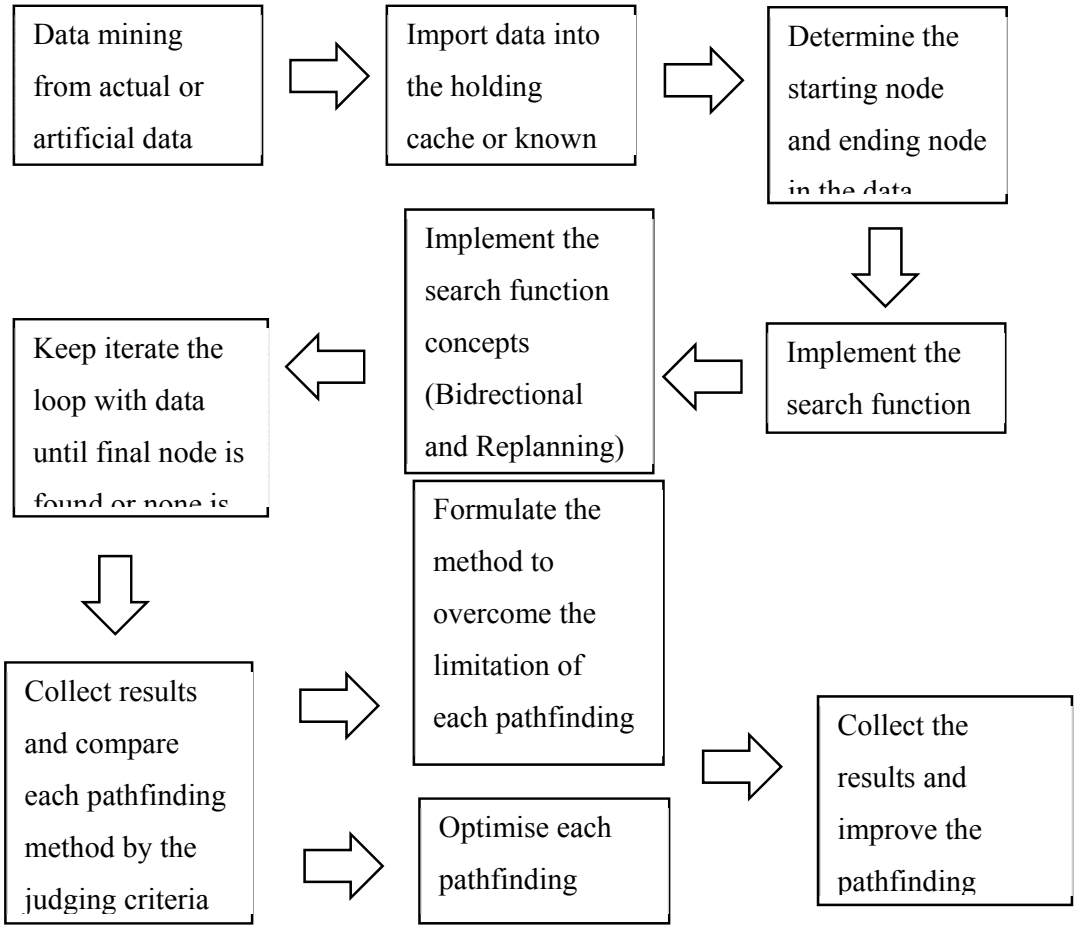

Figure 3-1 Implementation flowchart

3.3 Project Requirement

Software – MATLAB

3.3.1 Data Collection And Measurement Methods

Before implementing the pathfinding method, the field of work is needed to be set. The area of analysing is needed to determine whether is in a 2D/3D environment. The working environment is needed to be represented in proper form, grid or non-grid representation for the navigation of the pathfinding method (tile movement, edge movement, and vertex movement or hybrid movement) later on. Each vertex node can be even represented by hexagonal, squares or circles for path travelling. The working environment can be included with 'fog of sight' which prevent the pathfinding method to lookup before searching or it could be completely exposed to the line of sight of the pathfinding method. There is also the concern of each edge or traversable path properties whether it is unweighted edges, weighted edges or negative edges.

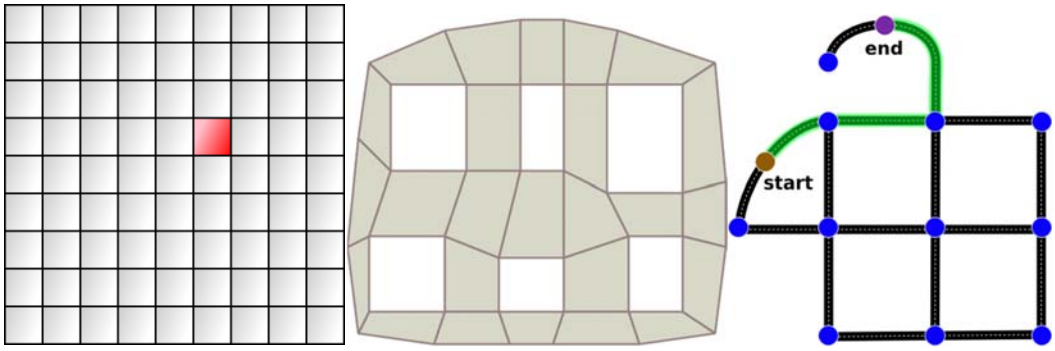

Figure 3-2 Different types of grid or non-grid diagram. Retrieved from *[20]*

3.3.2 Methodology: Pathfinding Algorithm Implementation (A* Algorithm)

For the sake of simplicity, 2D grid area is used as the area representation of the working area. Before executing the algorithm, the grid is first defined and the cost of the blocked cell is set to infinity (Inf). The starting point and ending point is loaded into our list. Therefore, the ending coordinates are known to the algorithm.

Starting off, the current node is set to the starting node coordinate (1, 4). The current node is kept updated as soon as a new minimum F cost node is found. The neighbouring nodes are added into an "open list" which has cost included. Blocked nodes such as inaccessible terrain is ignored as the cost is infinity.

For the 8 neighbouring nodes (since it is a grid), the lowest F cost is chosen as the next node (F = G + H from Equation 2-1) as mentioned above.

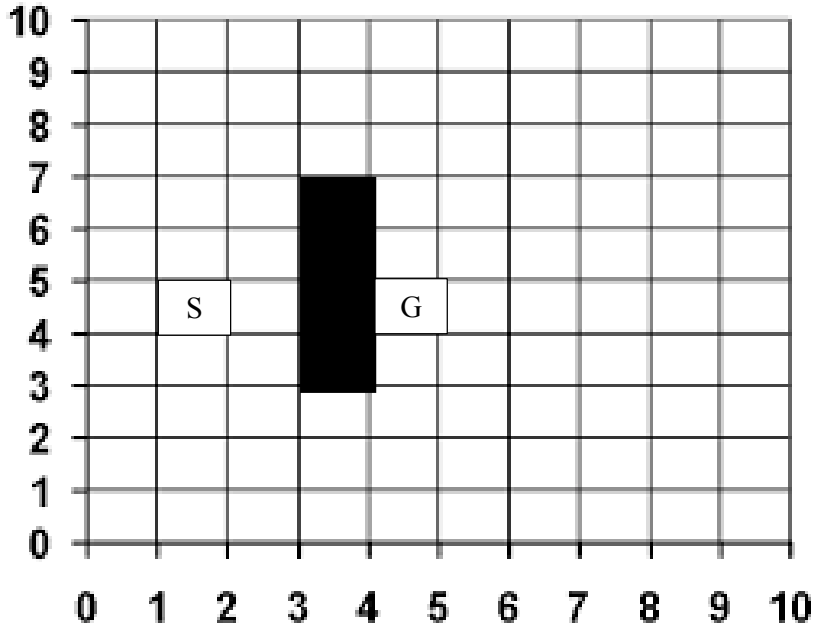

Figure 3-3 Coordinates of all nodes in a grid

From here, if neighbouring nodes of starting node is analysed, a few facts can be observed:

- The node (square) on the right has the lowest F cost = 10 + 20
- The starting node is added into 'Closed' list
- The 8 neighbouring nodes are added into 'Open' list
- The node (2, 4) is assigned as the current node.

For the detailed costs of all neighbouring nodes:

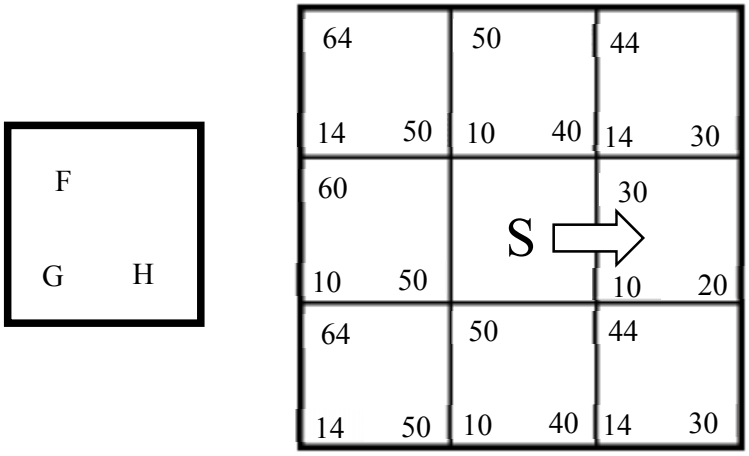

Figure 3-4 Detailed cost of each nodes.

For the part of H cost calculation (as an example), the Manhattan method is used for only calculating Heuristic cost, which means the diagonal path is not allowed. As an example. H cost of the node (2, 3) is calculated by moving 2 nodes right and one node up. In total, 30 = 20 + 10 is the cost

As for the part of G cost, for node (2, 3), it is 14 since it is directly diagonal from the starting node. The number 14 is chosen as approximation for Pythagorean distance ($\sqrt{10^2 + 10^2}$).

The whole process is repeated until the final G (goal) is reached. If during the loop, there is an adjacent open list square that has discovered before, the algorithm is needed to check whether there is a better path for it and assign the cost accordingly.[20]

3.4 Data Analysis

The idea of data analysis is to represent the data in a tabular format for easier identification, comparison and optimisation. Performance evaluation is crucial for the use of path optimisation. The data obtained from the pathfinding results should be comparable so further improvement can be done. If there is no obvious gain for the optimisation, the best results during the data analysis are expected. The trade-off between each algorithm needs to be stated down for the usage of data analysis. In this project, a few of the performance index are recorded for the use of judging criteria[21][22]:

- The total area searched either during searching process (Nodes Searched)
- The nodes stored after simulating the algorithm for next replanning purposes. (Closed Nodes)
- The nodes that are still in priority list after the algorithm simulation (Open Nodes)
- The successful linking of nodes from start to end. (Completion of the algorithm)
- The time taken for the completion of the simulating algorithm (Algorithm total time)
- Average time taken for running each iterations (Average time per loop)
- Number of node visitations. (Iterations)
- Total nodes affected due to the external walls and whitespaces introduced in the first map. (Total affected nodes)
- The nodes that are required to store in memory for the next replanning searching process. (Old Closed Nodes)
- The total nodes searched during the forward Bidirectional Searching and backward Bidirectional Searching (Nodes Searched FW and Nodes Searched BW)

3.5 Summary of Methodology

Path optimisation and verification of algorithm are expected to be achieved in this thesis.

4 RESULTS AND DISCUSSION

4.1 Introduction

This chapter presents the results of 5 concepts, namely, Dijkstra, Greedy, A*, Bidirectional, Dynamic planning and combination of A*, Bidirectional with Dynamic planning.

4.2 Initialization of Environment

For these simulations of path algorithm, the environment is needed to be defined so that each algorithm produce results with controlled variables. The map for path simulation is obtained from [23]. The maze is cropped and reduced to a 200x200 pixels image for faster simulation process. The path algorithms are used to find a single source starting point to a single goal. The field of this environment will be 10, this means the environment has same weighted edge cost.

4.3 Explanation example of simulations

An example of A* Bidirectional Re-planning is shown below

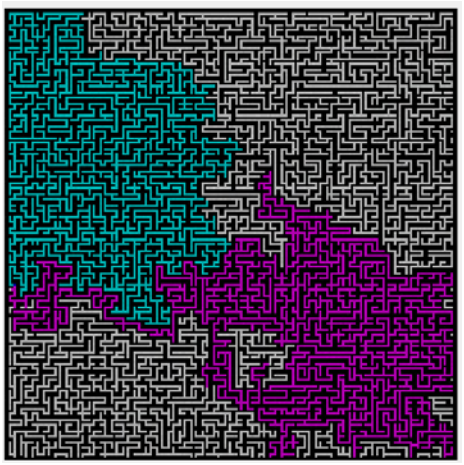

Figure 4-1 A* Bidirectional Re-planning

In this figure:

1. Black colour nodes are represented as walls.
2. White colour nodes are represented as passable path.
3. Blue colour nodes are represented as the simulation of algorithm(in this case: A* Bidirectional Re-planning) from start position towards goal position
4. Purple colour nodes are represented as the simultaneous simulation of the algorithm from goal position towards start position (Bidirectional purposes)

4.4 Algorithm Implementation

In these simulation, algorithms of different variant are simulated as follows:

1. A* algorithm
2. Dijkstra algorithm
3. Greedy algorithm
4. A* algorithm with replanning
5. Dijkstra algorithm with replanning
6. Greedy algorithm with replanning
7. A* algorithm with bidirectional searching
8. Dijkstra algorithm with bidirectional searching
9. Greedy algorithm with bidirectional searching
10. A* algorithm with bidirectional searching and replanning
11. Dijkstra algorithm with bidirectional searching and replanning
12. Greedy algorithm with bidirectional searching and replanning

Each movement from one node to another node is set to 10. 'Mahattan distance' is the heuristic method used to calculate the cost from start to goal.

4.5 Data benchmark

In these simulation, a few benchmark is set to analyse the competency of the algorithm.

1. Total number of nodes searched during the algorithm.
   (matlab code: numFinalTotalNodes)
2. Total number of nodes from starting position to end
   (matlab code: numNodesVisited)
3. Completion of the path
4. Total time required to run the algorithm
   (matlab code: AlgoEnd).
5. Average time of loop execution
   (matlab code: AvgTotaltElapsed)
6. Memory space used by the algorithm