Pathfinding & Collision Avoidance

William John Lautama

Table of Contents

Problem

There are 2 main problems that I am trying to solve which are pathfinding and collision-avoidance. The layout of the problem is presented with a map with green symbolizing land, dark blue symbolizing deep water, and light blue representing shallow water. Each tile also has their own cost with land having 100 / impassible, deep water having 3, and shallow water having 1. In the map, there are also multiple agents symbolized by “Ships.” These ships have their own goal and they need to pathfind and also avoid collision with other ships. The ships also need to adhere to the cost of each tile to get the most efficient path cost wise. The explanations of pathfinding and collision avoidance will be described in more detail below.

Pathfinding

The first problem that I need to solve is pathfinding. Based on wikipedia (https://en.wikipedia.org/wiki/Pathfinding), pathfinding or pathing is defined as the plotting, by a computer application, of the shortest route between two points. That means, pathfinding isn’t just trying to find a way from the start to the goal, but rather finding the shortest way to achieve it.

There are a couple of algorithms to achieve this but the efficiency and results of each algorithm varies. These algorithms are :

- DFS
- BFS
- Djikstra
- A*
- Jump-Point Search

DFS and BFS are the simplest algorithms with DFS having a LIFO (Last In First Out) list and BFS having a FIFO (First In First Out) list. In theory, these algorithms will achieve their goal but there is no guarantee these algorithms will have the optimal path cost wise.

Djikstra on the other hand are more efficient than BFS and DFS since it utilizes the g-value of each tile and can get the most optimal path to the goal. However, the disadvantage of Djikstra is that it may expand unnecessary nodes since it is a blind algorithm.

A* is a better version of Djikstra where it applies heuristics into the algorithm. The heuristics that I will be using is the Manhattan distance heuristic. The algorithm calculates the g-value but it also applies the h-value turning it into an f-value. It detects where the goal is and expands nodes towards the goal. This can create a more efficient algorithm with also an optimal pathing solution.

Jump-Point search is the most efficient algorithm but since I haven't delved too much into it, it will not be implemented. However, it is worth noting that Jump-Point search can be very useful in environments in games with large spaces such as MMORPG, RTS, etc.
In this assignment, I will be conducting an experiment with each algorithm mentioned above except Jump-Point Search to determine which one is the best. Even though theory wise, A* is clearly the best, it is still worth experimenting with other algorithms to have a more concrete result rather than just theory. It is also worth mentioning that the threshold for going through 100 ships is 60 seconds so if it surpasses 60 seconds, it will be considered a failure. Another aspect that needs to be seen is also the amount of nodes expanded and the planned path cost. Each scenario will have a dedicated threshold of nodes expanded and planned path cost that will be discussed further in detail and compared to in the **Results** section.

Collision Avoidance

When it comes to multiple agents in AI pathfinding, collision avoidance is very important. There are a couple of examples when it comes to collision avoidance such as Mario Kart (Enemy AIs), World of Warcraft (Heroes / Small Troops Movement), etc. Collision avoidance is also used in the field of robotics, automotive industries, etc.

In this problem, based on what was mentioned before in the previous section, there will be multiple ships in the map. The previous algorithms that were mentioned assume that the environment is static but it is new to be changed. There are a couple of collision avoidance methods that I have knowledge of which are :

- Reactive Planning (Push, Priority Planning, etc)
- Preactive Planning (Time Expanded A*)

Reactive planning reacts to the collision in real time instead of planning to avoid the collision. For example, when a ship is about to crash with another ship, the ship decides who gets priority and the one who didn't get priority has to replan. While this can be quite efficient in some areas, there are a couple of weaknesses depending on the environment and circumstances. Sometimes it can be incomplete and it can also be unoptimized but in a big environment, it can be utilized well.

Another type of collision avoidance is preactive planning. This involves planning beforehand before all of the ships even move to avoid collision full stop. This can create the same ratio between the planned and the actual cost of the ship but this can take a toll in the memory space since it involves storing a lot of data in each ship.

To avoid the ships crashing, I decided to implement reactive planning, specifically priority planning. In theory, since the map of the level is considered to be big, the algorithm should work well. Although Preactive Planning may seem like the perfect solution, the implementation is way

more trivial than reactive planning. This is because instead of a 2D grid, preactive planning sees the world in a 3D grid with time as an extra variable. This time variable can also take a toll in the memory space, making the simulation run slower than intended.

Ideas

Pathfinding

The main idea of the pathfinding is to find the most optimal path with the least amount of cells expanded. Jump-Point Search might be the best algorithm to implement currently but since I haven't gotten a full grasp on the full concept of the algorithm, I will be implementing A* since it is the 2nd best. Before I started implementation of A* I decided to experiment with other algorithms to get a good grasp of the concept of pathfinding algorithms.

DFS Algorithm

The first algorithm that I am experimenting with is DFS. DFS utilizes the LIFO (Last In First Out) list and it is very simple.

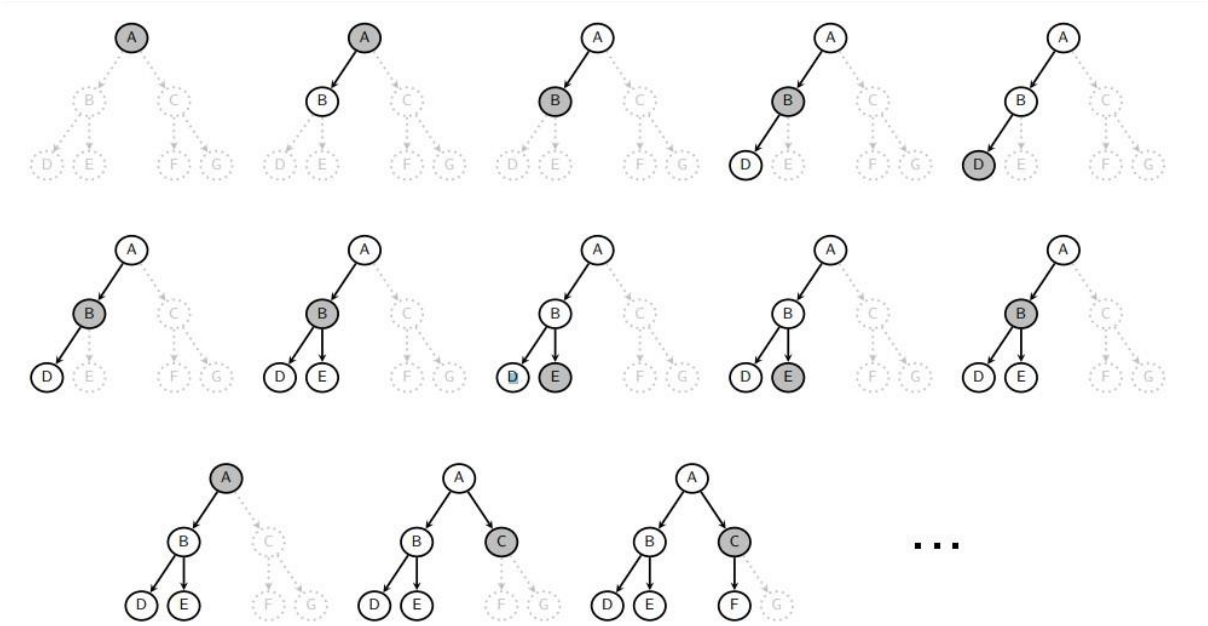

*Image taken from FIT3094 Week 2 Lecture

This image shows that it searches through the depth of each tree then moves to the next part of the branch. DFS is not optimal and it is also not complete. This means that DFS won't guarantee an optimal path and a complete path. The time complexity of it is also $O(b^m)$ where m is the length of the longest branch. The space complexity is O(m) assuming each node remembers the sibling to explore next.

Here is the Pseudocode that was given in Lab the week 2 lab :

```
Define starting node
Define end node
Add starting node to the OPEN list (LIFO data structure)
While stack contains nodes AND current node is NOT end node
        Current Node = pop next node from the beginning of the OPEN list
        If Current Node is the goal node
            return the current node as a solution
        Check current node neighbours
        For each neighbour (Start with left, up, right, down)
            If node is not traversable, continue
            If node is not on OPEN and node is not on CLOSED
                Add node to the beginning of the OPEN list
                Set parent of this node to be current node
```

This algorithm can be implemented to the problem but it is arguably one of the worst solutions out of all of the algorithms I mentioned. The result of this implementation will be discussed further in the **Results** section.

BFS Algorithm

The second algorithm that I experimented with is BFS. BFS is also very simple but instead of LIFO (Last In First Out) list, it utilizes FIFO (First In First Out).

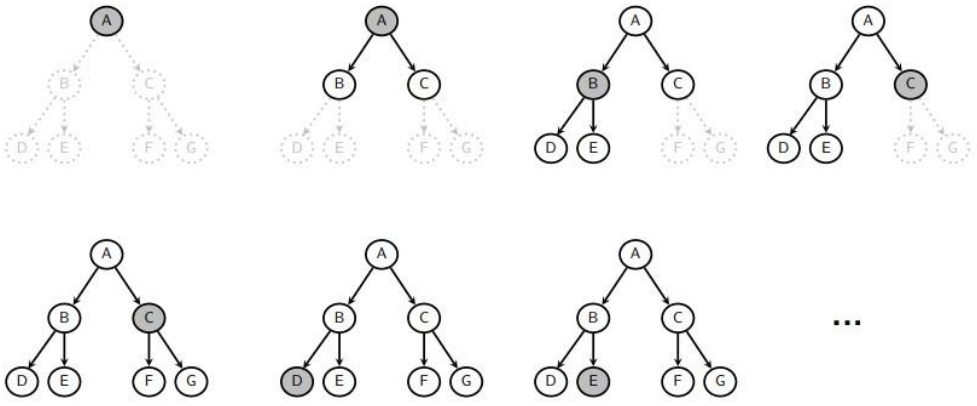

*Image taken from FIT3094 Week 2 Lecture

In contrast to DFS, BFS goes through the surface of each branch first in a layer and then goes out to the next one after the layer is done. This can be more efficient than DFS in some way

depending on where the goal is placed in the level. BFS is considered to be optimal provided the cost and depth are correlated and it is also complete. The time complexity of it is $O(b^d)$ and the space complexity is $O(b^d)$. Although the time and space complexity are identical, one is much more prohibitive than the other.

Here is the pseudocode for BFS :

```
Define starting node
Define end node
Add starting node to the OPEN list (FIFO data structure)
While stack contains nodes AND current node is NOT end node
        Current Node = get index 0 of Open List
        OpenList remove at index 0
        If Current Node is the goal node return the
                current node as a solution
        Check current node neighbours
        For each neighbour (Start with left, up, right, down)
                If node is not traversable, continue
                If node is not on OPEN and node is not on CLOSED
                        Add node to the beginning of the OPEN list
                        Set parent of this node to be current node.
```

The only difference between the code for BFS and DFS is how the data structure works. BFS uses FIFO so instead of popping the next node from the beginning of the open list, it gets the variable from index 0 of the open list and then removes it.

This algorithm can be implemented to the problem but it might not be the most optimal and efficient solution. The result of this implementation will be discussed further in the **Results** section.

Dijkstra Algorithm

The way dijkstra is implemented is more complex than BFS and DFS. It searches through the node but also takes account of the cost of the nodes so that it can find the most efficient path cost wise. This cost is stored in a variable called g and then the decision of searching through a tile is based on how much g value the tile has. This can make the search more efficient than BFS and DFS and also make the results more valuable.

Here is the Pseudocode that was given in Lab the week 3 lab :

```
Clear OpenList
Clear ClosedList
startNode.g = 0
Add start Node to OpenList
While (OpenList is not empty)
        currentNode = RemoveNodeWithSmallest_g from OpenList
        Add currentNode to ClosedList
        If (currentNode is goalNode)
                Return path from startNode to goalNode --- success!
        For Each (nextNode accessible from currentNode)
                If (nextNode is in ClosedList)
                        Skip nextNode
                Else
                        possible_g = currentNode.g + CostBetween(currentNode, nextNode)
                        possible_g_isBetter = false
                        If (nextNode is not in OpenList)
                                Add nextNode to OpenList
                                possible_g_isBetter = true
                        Else If (possible_g < nextNode.g)
                                possible_g_isBetter = true


                        If (possible_g_isBetter is true)
                                nextNode.cameFrom = currentNode
                                nextNode.g = possible_g
Return null --- failed to find a path!
```

Djikstra might be considered to be very optimal since it can find the shortest and the most efficient path cost wise from the start to the goal. The only drawback of djikstra is the amount of cells it might expand since it is a blind algorithm. The result of this implementation will be discussed further in the **Results** section.

A* Algorithm (My Solution)

A* algorithm fundamentally is a better version of the Dijkstra algorithm. It utilizes heuristics into the code. There are a couple of heuristics that can be used but I will be using the Manhattan Distance heuristic since it is the most suitable for the problem.

Here is the Pseudocode for A* (Manhattan Distance) :

Clear OpenList Clear
ClosedLIst startNode.g = 0
Add start Node to OpenList
While (OpenList is not empty)

```
    currentNode = RemoveNodeWithSmallest_g and smallest_f from OpenList
Add currentNode to ClosedList
If (currentNode is goal Node)
        Return path from startNode to goalNode --- success!
For Each (nextNode accessible from currentNode)
        If(nextNode is in ClosedLIst)
            Skip nextNode
        Else possible_g = currentNode.g + CostBetween(currentNode, nextNode)
            possible_f = possible_g + absolute of X of start – X of goal + absolute
            of Y of start – Y of goal possible_cost_isBetter = false if(nextNode is
            not in OpenList)
              Add nextNode to OpenList
                 Possible_cost_isBetter = rue
            Else If (possible_g < nextNode.g)
                 Possible_cost_isBetter = true

           If(possible_g_isBetter is true)
nextNode.cameFrom = currentNode nextNode.g =
possible_g nextNode.f = possible_f Return null --- failed to
find a path!
```

As it can be seen from the pseudocode, it is just an adjustment of the dijkstra algorithm. The extra adjustments are the h value adding into the f value. The h value came from the formula : $|x_1 - x_2| + |y_1 - y_2|$

Then, the h value is added into the g value then it becomes an f value. Then, the f value is compared with the other nodes for the algorithm to figure out which node to expand next. This helps with the efficiency of the algorithm while maintaining that optimal solution. The result of this implementation will be discussed further in the **Results** section.

Collision Avoidance

As mentioned before, I decided to implement priority planning for the collision avoidance. I couldn't experiment with other collision avoidance algorithms due to time constraints but there are a couple of things that can be noted just from priority planning alone. Here are some of the things that I experimented with :

Algorithm

Priority Planning without Priority

This method is quite interesting since it didn't use the priority aspect of the algorithm. This could cause problems and also inefficiency. Here is the pseudo code of priority planning without priority :

```
if collision is deactivated for ships number ship at
        next node and going to goal
        return

GridNode array LastNodes
GrideNode array CurrentNodes

Gridnode array NextNodes

for i in ships number ship at next node and going to
        goal add ship's last node to LastNodes list
        add Ships's current node to CurrentNodes list

        if ship at goal put that ship's goal node to
                NextNodes list
        else put the ship's next path to NextNodes list

        for i in number of NextNodes for j = i + 1 in number of NextNodes
                if NextNodes i index is equals to NextNodes j index
                                Replan ship by re running A* but block the path in NextNodes
                for j = i + 1 in number of LastNodes if NextNodes i index is
                        equals to NextNodes j index
                                Replan ship by re running A* but block the path in NextNodes
```

This code looks very simple but the idea of it is that it checks through the arrays of the ships and checks if a ship is about to crash 1 step ahead. If a ship is about to crash, the ship that triggered the code will have to replan the ship by re running A* but put the node that the ship is about to crash in as a wall.

The general idea of this can possibly work since it will avoid the ship but it can be highly inefficient since there are no priorities and both ships can trigger the replanning. For example, if there are 2 ships that are about to crash at the same time, both of them could trigger the replan

and avoid the original plan even though the original plan can be used for at least one of them. The result of this implementation will be discussed further in the **Results** section.

Priority Planning (My Solution)

This solution is the one that I implemented due to its simplicity and effectiveness. Logically, by implementing this, the ship won't throw away the original plan but instead will choose who will replan. This is not the most effective method but it will work effectively in a map in the problem since it is a very big map. Here is the pseudocode for Priority Planning :

```
if collision is deactivated for ships number ship at
        next node and going to goal
        return

GridNode array LastNodes
GrideNode array CurrentNodes

Gridnode array NextNodes

for i in ships number ship at next node and going to
        goal add ship's last node to LastNodes list
        add Ships's current node to CurrentNodes list

        if ship at goal put that ship's goal node to
                NextNodes list
        else put the ship's next path to NextNodes list

        for i in number of NextNodes for j = i + 1 in number of NextNodes if NextNodes
                i index is equals to NextNodes j index if NextNodes i index is equals to
                TempGridNode variable continue
                                else
                                        TempGridNode = NextNodes i index
                                        Replan ship by re running A* but block the path in
                                        NextNodes
                for j = i + 1 in number of LastNodes if NextNodes i index is equals to
                        NextNodes j index if NextNodes i index is equals to
                        TempGridNode variable continue
                                else
                                        TempGridNode = NextNodes i index
                                        Replan ship by re running A* but block the path in
                                        NextNodes
```

The only difference between this code and the code previously for priority planning without planning is the planning type. The priority is taken by the ship who is detected to crash first as it can be seen from the TempGridNode variable. This leads to a more efficient ship collision avoidance and can help with solving the problem in a well thought out manner. The result of this implementation will be discussed further in the **Results** section.

Placement of Function

While I was solving this problem, I encountered an interesting encounter. Putting the function after each boat is in the next node or every tick greatly affects the results. Here are a brief explanation of each placements :

Putting Priority Planning After Each Boat is in Next Node

Putting priority planning at the end of tick or when the ship is about to crash is regarded as efficient. This is because it doesn't run every frame and can save up in processing power. In theory this should be the best solution and I should've used this placement but I opted for the other placement. This is because in theory, the ship wouldn't be able to react fast enough to recalculate and re-plan the positions. This can cause problems and make it too late for the ship to replan to avoid collision. The result of this implementation will be discussed further in the **Results** section.

Putting Priority Planning Every Tick (My Solution)

Putting priority planning at every tick is very inefficient but I still opted for this option. This is because by putting it in every tick, this causes the ship to be highly reactive to the surroundings and makes the ship less prone to error and crash before it's too late to calculate. The result of this implementation will be discussed further in the **Results** section.

Results

In this section, I will discuss the results of each method of pathfinding and collision avoidance that I mentioned in the previous section.

Pathfinding

Each pathfinding result will be accompanied by a table of the results of each data. The data that I am going to include is the Amount of Cells Expanded, Planned Path Cost, and Planning Time for each scenario. If the test runs above 60 seconds I will not continue the experiment and I will consider it as a failure. The optimality of the pathing will also be considered as a failure if it is too much.

DFS Algorithm

As it can be seen from the red dots, the algorithm is not efficient at all pathing wise since it is going through every crevice of the map. The statistics for this algorithm will not be researched anymore since this algorithm is a clear bad option from scenario 1.

BFS Algorithm

Scenarios	Cells Expanded	Planned Path Cost	Planning Time (Secs)

Scenario 1	27874 < 27978	470 > 299	0
Scenario 2	55748 > 19348	391 > 302	1
Scenario 3	139370 > 59722	916 > 695	2
Scenario 4	278740 > 113829	2348 > 1577	5
Scenario 5	696850 > 386396	6685 > 4784	12
Scenario 6	1393700 > 722914	13485 > 8748	23
Scenario 7	2787400 > 1414194	26767 > 17866	47

Based on these results, the only variable and scenario that is good enough according to the threshold is Scenario 1's cells searched. Even though all of the scenarios are below 60 seconds, the cells searched and the path cost are highly inefficient. Through this experiment, it can be concluded that BFS is not the preferred algorithm for this problem.

Dijkstra Algorithm

Scenarios	Cells Expanded	Planned Path Cost	Planning Time (Secs)
Scenario 1	27874 < 27978	298 < 299	0
Scenario 2	55748 > 19348	301 < 302	1
Scenario 3	139370 >59722	694 < 695	2
Scenario 4	278740 > 113829	1576 < 1577	4
Scenario 5	696850 > 386396	4783 < 4784	10
Scenario 6	1393700 > 722914	8747 < 8748	21
Scenario 7	2787400 > 1414194	17867 > 17866	41

The results from this algorithm are very interesting. Supposedly, it's supposed to give the most optimal path. It did but specifically for scenario 7 it didn't which could be a bug. It also runs

under 60 seconds but all but scenario 1's cells expanded are not optimized. The Dijkstra algorithm could be used to solve this problem but it needs to be optimized for cells searching wise and also fixed in terms of the bug in scenario 7.

A* Algorithm (My Solution)

Scenarios	Cells Expanded	Planned Path Cost	Planning Time (Secs)
Scenario 1	12982 < 27978	298 < 299	0
Scenario 2	8114 < 19348	301 < 302	0
Scenario 3	15625 < 59722	694 < 695	0
Scenario 4	41325 < 113829	1576 < 1577	0
Scenario 5	158197 < 386396	4783 < 4784	2
Scenario 6	283578 < 722914	8747 < 8748	4
Scenario 7	593541 < 1414194	17865 < 17866	7

This scenario is the most optimal scenario. As it can be seen. Everything is under the threshold which is the most ideal. The planning time is also way faster than Dijkstra because it's not blind and knows where the goal is. Based on these experiments, I decided to go with A* algorithm for my pathfinding problem.

Collision Avoidance

Priority Planning Without Priority

*Exact results may vary

Scenarios	Ratio of Actual Cost to Planned Cost
Scenario 1	1x
Scenario 2	1x

Scenario 3	1x
Scenario 4	1.007614x
Scenario 5	1.003136x
Scenario 6	1.006860x
Scenario 7	Undetermined

The results of priority planning without priority is interesting. It is put in tick and when tested, everything works accordingly except for scenario 7. In scenario 7, it can be seen that a lot of the ships are in a stalemate where they cannot move and they are only going back and forth. They will eventually move, but the time it will take for them to move is a lot. This means that priority planning cannot be executed properly without deciding who is the priority.

Putting Priority Planning After Each Boat is in Next Node

*Exact results may vary

Scenarios	**Ratio of Actual Cost to Planned Cost**
Scenario 1	1x
Scenario 2	1x
Scenario 3	1x
Scenario 4	1.007614x
Scenario 5	1.378424x
Scenario 6	1.323539x
Scenario 7	1.808396x

The ratio of the priority planning after each boat is in the next node is very significant. This is considered to be bad since the ratio is still very high and there are still a lot of ships crashing especially in scenario 5 and above. This shows that the placement of this priority planning still needs fixing. The reasoning for this is because when the ship is about to crash, it is too late to replan so they will crash and then replan.

Putting Priority Planning Every Tick (My Solution)

*Exact results may vary

Scenarios	Ratio of Actual Cost to Planned Cost
Scenario 1	1x
Scenario 2	1x
Scenario 3	1x
Scenario 4	1.007614x
Scenario 5	1.004182x
Scenario 6	1.007203x
Scenario 7	1.015001x

From these results, it can be seen that by placing the function call in tick() (every frame) can decrease the ratio and prevent all crashes. This shows that priority planning does work but it has to be put in a specific call. There may be a better solution than putting it in tick() but since optimisation is not a main problem in the problem, this is already one of the best solutions other than time expanded A*. Based on these experiment results, I decided to go with this method. Though, there is one major bug with this implementation which is the ship sometimes moves diagonally because of its inaccuracies of getting the current location. This is not considered to be invalid since technically the ship is still programmed to move up, left, right, and down.

Discussion & Future Work

My proposed solution and its implementation is not the best solution that is available but it is good enough to solve the problem accordingly. It's the best when it comes to getting the most optimal path but it is not the best when it comes to cell searching and collision avoidance. For cell searching, Jump-point search could be a better candidate but unfortunately I don't have enough knowledge of it to implement it. For collision avoidance, time expanded A* is the best option but again, unfortunately my knowledge and time is scarce. But overall, my design of A* and Priority Planning is not a bad implementation and it works perfectly fine according to the threshold of the problem. The only main issue is the bug surrounding the potential diagonal movements and also the inefficiency of the code by putting it in every frame that could be fixed if I had more time.

My algorithm worked perfectly as expected but some of the experimented algorithms worked oddly. For example, the Dijkstra algorithm should’ve gotten the optimal path but specifically for scenario 7 it didn't, which is most likely caused by a bug. But my main A* algorithm and Priority Planning is working as expected except for the main issue with the priority planning where it could go diagonal because of its inaccuracy.

The main challenge in implementing the algorithm is the knowledge going into it. Without the proper knowledge, I didn't have a clue on how to implement any of the algorithms. With the help of people with more experience and self research, I was able to grasp my mind on the topic of pathfinding algorithms. It just takes time to understand it more but overall, it is highly expected of this problem.

Moving forward, if I could do it differently next time, I would lessen my experiments with the simple algorithms (BFS, DFS, etc) and go straight with the complex algorithms. This would give me more time to improve it and get better solutions to solve the problem.

Overall this problem was a very interesting problem that could be applied to the real world as well as the games industry. Whether its enemy pathfinding, robot pathfinding, or online maps, pathfinding algorithms are one of the most useful tools in the technology industry.