SWINBURNE UNIVERSITY OF TECHNOLOGY

SCHOOL OF SCIENCE, COMPUTING AND ENGINEERING TECHNOLOGIES

***

COS30019 – INTRODUCTION TO AI

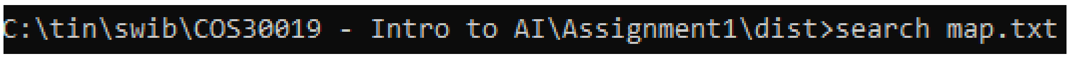
ASSIGNMENT 1 ROBOT NAVIGATION PROBLEM

Name: Gia Hung Tran
Student ID: 103509199

Table of Contents

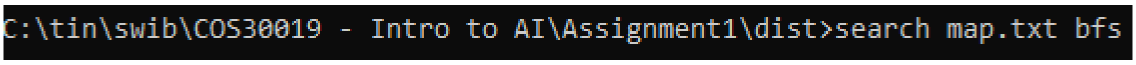
Instruction

The program can be executed either via a command prompt or through a user-friendly graphical user interface (GUI) that offers navigation across various search algorithms. To print the order of moves that brings you from start-configuration to end-configuration, the command line argument is required to input the type of search algorithm. The syntax for the console is "*search <filename> < method>* ". The selected method can be entered in lowercase or uppercase, provided it is in the following list: dfs, dfsr, bfs, gbfs, as, cus1, cus2. Besides, The GUI version starts with the search.exe file but without the command line argument for a chosen method. Figure 3 illustrates that the program has 6 different search algorithms including BFS, DFS, A*, GBFS, and two custom searches (bidirectional search, and bidirectional A*). The red cell signifies the starting point while the green cells denote the goals. During the search algorithm, the program displays the expanded nodes in a darker green colour, while the nodes in the frontier but not yet visited are shown in orange. When the program finds a path, it highlights it in a bisque colour.



```
C:\tin\swib\COS30019 - Intro to AI\Assignment1\dist>search map.txt
```

Figure 1. Syntax to run GUI.



```
C:\tin\swib\COS30019 - Intro to AI\Assignment1\dist>search map.txt bfs
```

Figure 2. Syntax to print the result of a specific algorithm in console.

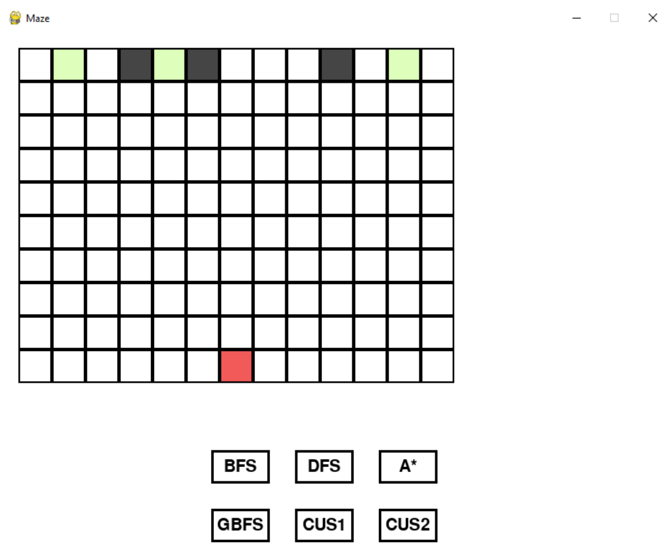


Figure 3. GUI program

Introduction

The problem of robot navigation in mazes has garnered significant attention in the field of AI, particularly in the context of pathfinding. A wide range of solutions has been proposed to address this problem, which can be broadly categorized into two groups: informed and uninformed search algorithms. This research aims to compare and contrast these two algorithmic approaches, with a particular focus on tree-based search rather than graph-based search. While these approaches share some similarities, they differ fundamentally in how they manage the explored set

In this study, we consider mazes as environments that consist of an NxM grid, with the agent's initial state located in an empty cell. The agent's objective is to find a path to one of the designated goal cells. The agent can take only one action at a time which includes moving up, left, down, or right. By analysing and comparing the advantages and disadvantages of tree-based search algorithms and uninformed search algorithms, we aim to provide insights that could lead to more efficient and effective navigation strategies for robots in complex environments such as mazes.

Glossary

- BFS: breadth-first search
- DFS: depth-first search
- GBFS: greedy best-first search
- Frontier: a data structure to keep a set of nodes that are going to be expanded. There are different types of frontier, but in this research there are 3 kinds, stack, queue, priority queue
- Priority queue: data structure that stores data in a ascending or descending value
- Heuristic function: is a function to calculate the distance between a node with the goal. There are different types of heuristic functions.
- Manhattan distance: the distance between two points calculated based on this formula: $abs(a.x - b.x) + abs(a.y - b.y)$

Search Algorithms

In this research, six different search algorithm is examined thoroughly in the terms of efficiency and memory usage. Breadth-first search (BFS), depth-first search (DFS), A*, greedy best-first search, bidirectional search, and bidirectional A* search will be introduced.

Breadth-First Search (BFS)

Overview

Breadth-first search is an uninformed search algorithm that begins by expanding the root node first then all of its successors. As it traverses each level of the search, the algorithm expands all nodes in the same layer before descending to the next layer. The algorithm uses first-in-first-out(FIFO) queue as a frontier so that all nodes can be expanded chronologically. A goal test will be applied to each expanded node to determine the termination of the search. The search also checks for repeated states by maintaining a list of visited nodes, so it will not add any visited node back to the frontier. Due to the nature of the algorithm, the search will always have the shallowest path to every node on the frontier (Russel and Norvig, 2010). Thus, it guarantees that a path will be found if the goal node is at some finite depth.

Complexity

- Time complexity: $O(b^d)$
- Space complexity: $O(b^d)$

  Where d is the depth of the tree

With the complexity of $O(b^d)$, the breadth-first search would result in a significant run time and memory consumption when encountering a tree with great depth. A computer can run the program for a long time, but an exponential space complexity can lead to the failure of the program (Russel and Norvig, 2010). With finite state space, the search is complete, and if the cost to traverse each is step is one then the search is also optimal.

Depth-First Search (DFS)

Overview

Depth-first search is also an uninformed search, but contrary to breadth-first search, it tries to expand the deepest node of the search tree first. The algorithm will run down the layers until there is no successor for a node, then it returns to the next deepest node that still has an unvisited successor (Russel, 2010). There are two main ways to implement depth-first search, recursive function, and stack as a frontier. Each approach has its own advantages and disadvantages. This problem will be discussed later. The depth-first search usually does not return the optimal path because it tries to go as deep as possible. However, it has an advantage over the breadth-first search which is the reduction in space complexity.

Complexity

- Time complexity: O(b^m)
- Space complexity: O(bm)

  Where m is the largest depth of any node

The time complexity is not improved compared to the breath-first search, but it saves space complexity. Especially with the recursive way, the program does not need to store all nodes to check for visited and nodes can be removed from memory when all their successors have been expanded. However, if m is larger than d, it can cause a terrible consequence. DFS is not complete when there is infinite-depth spaces and spaces with loops, and repeated states check can overcome the problem.

Bidirectional Search

Overview

The core idea of bidirectional search is to perform two searches simultaneously, one from the initial state and one from the goal. the search will terminate when they meet in the middle. The search implemented in those searches is breath-first search. Instead of a goal test, there is an intersection check to determine if two searches meet each other. The main improvement of bidirectional search is the reduction in run time. Moreover, it does not return the optimal path, because the intersection does not necessarily mean the optimal path, and it is complete with finite space.

Complexity

- Time complexity: O($b^{d/2}$)
- Space complexity: O($b^{d/2}$)

  This is a great improvement over the breadth-first search, and it can boost the speed of the search.

Greedy Best-First Search

Overview

This is an informed search where a heuristic function will be applied to find a node that is closest to the goal. There are different types of heuristic functions, and the one utilised in the robot navigation problem is the Manhattan distance. The goal then pushes the nodes into a priority queue based on heuristic value. The search will try to go to the nodes that have a more promising value. Nonetheless, it does not assure the optimal path, because the closer node to the goal based on some distance does not ensure that the search will always traverse the shortest path. A good heuristic function can enhance efficiency and reduce the complexity.

Complexity

- Time complexity: O(b^m)
- Space complexity: O(b^m)
  Where m is the maximum depth of search space

The worst case in run-time has the same result with the depth-first search, but a good heuristic function can give considerable improvement. Memory consumption of this algorithm is large because it keeps all nodes in the memory. Hence, greedy best-first search is not optimal because it does not keep track of the current cost of the path and the heuristic function does not guarantee that the node will lead to the shortest path.

A*

Overview

A* is a popular informed search. It evaluates a node by combing g(n) the cost to the current node and h(n) the cost from that node to the goal which is estimated by a heuristic function.

f(n) = g(n) + h(n)

thanks to this improvement compared to the greedy best-first search, the search is complete and optimal. The problem with A* is that the heuristic function needs to be admissible, which never overestimates the cost to the goal. $h(n) \leq h^*(n)$, where $h^*(n)$ is the optimal cost to reach the goal. In the case of Robot navigation, Manhattan is sufficient to calculate the cost because of the feature of the grid.

Complexity

- Time complexity: O(b^d)
- Space complexity: all nodes are stored

A* is an optimal search where it can guarantee the shortest path, but it stores all the nodes. Generally, A* expand more nodes than greedy best-first search, so it consumes more memory to compute.

Bidirectional A*

Overview

This is a combination of A* and bidirectional search. Instead of the breadth-first search like in normal bidirectional search, it will use A* to search. This search is complete, but it is not optimal because the intersection check cannot promise the shortest path like the normal A*. It is not as effective as A* in terms of optimality, but it has faster speed and reduces the memory usage compared to A* and normal bidirectional search due to the leverage of a heuristic function.

Complexity

- Time complexity: O($b^{d/2}$)
- Space complexity: O($b^{d/2}$)

It still has the same time and space complexity as the normal bidirectional search, but it can work more effectively with the use of heuristic function

Comparison

With the test case provided in Figure 4, the results of each search algorithm can witness in the table

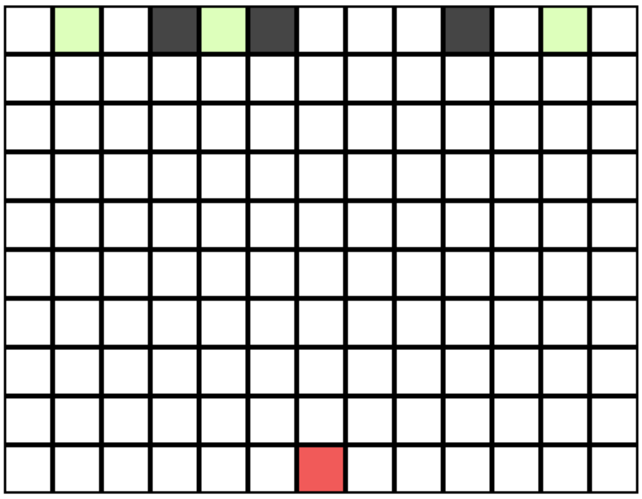


Figure 4. Sample maze.

Table 1. Performance comparison of search algorithms

Criterion	BFS	DFS	A*	GBFS	Bidirectional	Bidirectional A*
Time(ms)	1.999	0.9965	0.9978	0.5433	1.208	0.678
Node created	109	102	48	33	62	44

In order to yield objective results, each algorithm was run ten times and the average run time was calculated. Overall, it can be observed that informed searches have better performance than uninformed ones in terms of run time and the number of nodes created. DFS and GBFS share a similarity in which the run-time is faster than other approaches within the same category, but they are not optimal. On the other hand, bidirectional and bidirectional A* reduce the run time and space used in comparison to BFS and A* respectively, which is similar to what the theory proposes.

Implementation

The program is structured in an object-oriented way in which the grid contains all cells with all attributes for a maze like if it is a wall or goal. There are functions to search the path from the initial state to one of the goals, which are placed in a same file. The functions for bidirectional search and bidirectional A* search are named CUS1search and CUS2search functions. All the functions take grid as the input and return a list of cells that form the path.

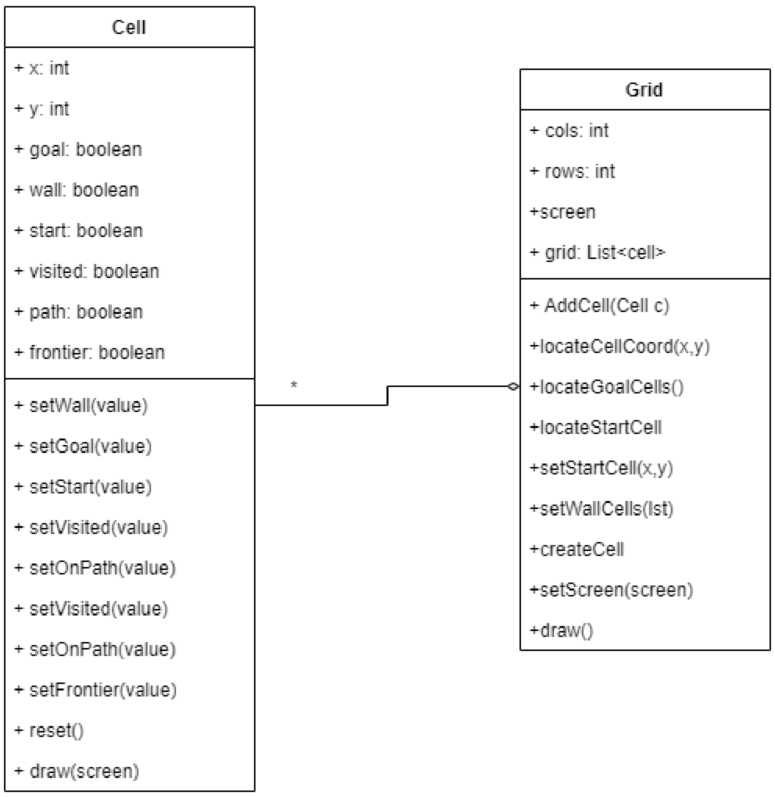


Figure 5. Cell and grid UML diagram

Breadth-First Search

Breadth-first search utilises a first-in-first-out queue data structure. The program pushes the root node first into the queue. Then it uses a while loop to loop until the queue is empty. In the while loop, it will pop out the first element of the queue, then check if it is the goal. If yes, it will return the path and terminate the function. Then, it loops through all possible neighbours of the cell and checks if they are wall cells or visited. If they satisfy the requirements, they will be pushed into the queue. The program will terminate when it finds the goal or there is no cell left in the queue. If there is no cell left in the queue, the function will return None which means that there is no path found.

```
procedure BFSsearch(grid):
    startCell = grid.locateStartCell()
    queue = Queue()
    queue.append(startCell)
    path = {startCell: [startCell]}
    while queue is not empty:
        current = queue.pop()
        if current is goalCell:
            return path[current]
        end if
        adjancents = grid.locateNeighbour(current)
        for adjacent in adjancents:
            if adjacnet.isVisited == true or adjacent.isWall == true:
                continue
            end if
            adjacent.isVisited = true
            path[adjacent] = path[current] + [adjacent]
            queue.append(adjacent)
         end for
    end while
    return None
end procedure
```

Figure 6. BFS pseudocode

Depth-First Search

Depth-first search is implemented similarly to breadth-first search. The main difference is the data structure used for the frontier.

```
procedure DFSsearch(grid):
    startCell = grid.locateStartCell()
    stack = Stack
    stack.append(startCell)
    path = {startCell: [startCell]}
    while stack is not empty:
        current = stack.pop()
        current.isVisited = true
        if current is goalCell:
            return path[current]
        end if
        adjancents = grid.locateNeighbour(current)
        for adjacent in adjancents:
            if adjacnet.isVisited == true or adjacent.isWall == true:
                continue
            end if
            path[adjacent] = path[current] + [adjacent]
            stack.append(adjacent)
         end for
    end while
    return None
end procedure
```

Figure 7.DFS pseudocode

Greedy Best-First Search

The idea of greedy best-first search is quite similar to breadth-first search, and the difference is that it uses a priority queue to store nodes. Priority queue ensures that nodes are sorted based on their cost and it always pops out node with the smallest cost first. In the robot navigation problem, Manhattan distance is leveraged as the heuristic function to calculate the cost from a node to either of the node. In case there are multiple goal, it will calculate the distance between the node and each of the goals then return the smallest distance one.

```
procedure GBFSsearch(grid):
    startCell = grid.locateStartCell()
    queue = PriorityQueue()
    queue.append((0,startCell))
    path = {startCell: [startCell]}
    while queue is not empty:
        current = queue.pop()
        if current is goalCell:
            return path[current]
        end if
        adjancents = grid.locateNeighbour(current)
        for adjacent in adjancents:
            if adjacnet.isVisited == true or adjacent.isWall == true:
                continue
            end if
            adjacent.isVisited = true

            path[adjacent] = path[current] + [adjacent]
            queue.append((heuristic(adjacent,grid.goalCells),adjacent))
         end for
    end while
    return None
end procedure
```

Figure 8.GBFS pseudocode

```
procedure ManhattanDistance(node, goals):
    min = +∞
    for goal in goals:
        temp = abs(node.x - goal.x) + abs(node.y - goal.y)
        if temp < min
            min = temp
         end if
    end for
    return min
end procedure
```

Figure 9.Manhattan distance implementation

A*

A* is nearly identical to greedy best-first search in terms of the core concept, and the difference is how it evaluates the costs of a node. Instead of using only the heuristic function, A* combines the cost to reach the current node with the heuristic value of that node.

```
procedure ASsearch(grid):
    startCell = grid.locateStartCell()
    queue = PriorityQueue()
    queue.push((0,startCell))
    path = {startCell: [startCell]}
    cost = {startCell: 0}
    while queue is not empty:
        current = queue.pop()
        if current is goalCell:
            return path[current]
        end if
        adjancents = grid.locateNeighbours(current)
        for adjacent in adjancents:
            if adjacnet.isVisited == true or adjacent.isWall == true:
                continue
            end if
            newCost = cost[current] +1
            if adjacent not in cost or newCost < cost[adjacent]:
              cost[adjacent] = newCost
              priority = newCost + heuristic(adjacent, grid.goalCelss)
              queue.push((priority, adjacent))
              path[adjacent] = path[current] + [adjacent]
            end if
        end for
    end while
    return None
end procedure
```

Figure 10. A pseudocode*

Bidirectional Search

The implementation of bidirectional search is like BFS, but it searches from both ends instead of just one way from the initial point. But this search is uninformed so there is possibly a case in which there is no path to one of the goals. Hence, the search needs to loop through the list of goal cells until it finds a path. If there is no path found to all of them, it returns none.

```
procedure BidirectionalSearch(grid):
    startCell = grid.locateStartCell()
    goalCells = grid.locateGoalCells()
    while goalCells is not empty:
        forwardQueue = Queue()
        forwardQueue.append(startCell)
        forwardVisited = {startCell: [startCell]}

        backwardQueue = Queue()
        goalCell = goalCells.pop(0)
        backwardQueue.append(goalCell)
        backwardVisited = {goalCell: [goalCell]}
        visitedIntersection = None
        while forwardQueue is not empty and backwardQueue is not empty:
            forward = forwardQueue.pop()
            adjacents = grid.locateNeighbours(forward):
            for adjacent in adjancents:
                if adjacent in forwardVisited or adjacent.isWall == true:
                    continue
                end if
                forwardQueue.append(adjacent)
                forwwardVisited[adjacent] = forwardVisited[forward] + [adjacent]
                if adjacent in backwardVisited:
                    visitedIntersection = adjacent
                    break
                end if
            end for
            if visitedIntersection is not None:
                break
            end if

            backward = backwardQueue.pop()
            adjacents = grid.locateNeighbours(backward):
            for adjacent in adjancents:
                if adjacent in backwardVisited or adjacent.isWall == true:
                    continue
                end if
                backwardQueue.append(adjacent)
                backwardVisited[adjacent] = backwardVisited[forward] + [adjacent]
                if adjacent in forwardVisited:
                    visitedIntersection = adjacent
                    break
                end if
            end for
            if visitedIntersection is not None:
                break
            end if
        end while
```

Figure 11. Bidirectional search pseudocode

Bidirectional A*

This algorithm is similar to A*, but it searches from both ends. Bidirectional A* expands nodes based on the evaluation function like A* which sums the path cost and the heuristic value. Bidirectional A* does not guarantee the optimal path when the termination condition of the search is meeting in the middle. However, there is some remedies to that problem which will be discussed later.

Features/Bugs

Features

Features implemented in the program:

- There is a command line interface that read the path to text file and the chosen search method.
- There are 6 search algorithms, BFS, DFS, GBFS, A*, bidirectional search, and bidirectional A*. The last two algorithms are custom searches, one of which is uninformed while the other is an informed search. Moreover, there is also DFS recursive to compare the performance.
- There is a GUI to visualise different pathfinding algorithms. As seen in Figure 12 , CUS1 and CUS2 stand for custom search 1 and custom search 2 respectively.
- The red cell is the initial point while the light green is the goal. Cells with black colour are the walls. The dark green colour stands for the cells that have been visited by the algorithm while the orange ones have been pushed into the frontier but not visited yet.

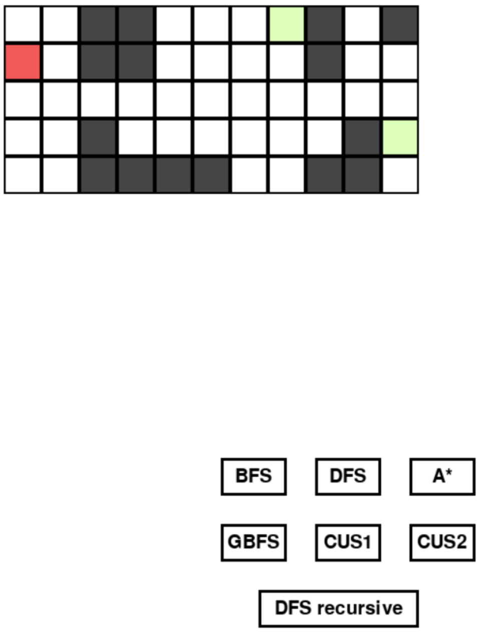


Figure 12. GUI

Bugs

There are some issues with the GUI program. The design for the GUI program is not very effective, so when facing a complicated test case, it tends to be unstable. Furthermore, the size of the cell is fixed, so when the maze is too large, the GUI cannot draw all the maze.

Research

Visualisation

The program utilises the pygame library to draw the interface. The program runs on a single thread, so there may be some performance issues while running the GUI program

Optimisation

There are two enhancements implied on DFS and bidirectional A* to make them more effective. Firstly, there are two popular versions of DFS, one that is recursive and the other that involves using a list to keep track of visited nodes. The recursive method uses less memory because it does not save the nodes and deletes them from the branch after traversing but not finding a path. Therefore, it will consume less memory. To compare memory usage of them, node counting is not applicable because the recursive approach does not actually create any node, but we can count the number of states that are visited by the recursive function. This can be observed from the real test case. With the test case same as Figure 4, the recursive function is just recalled for 50 times while the other creates up to 102 nodes, which is a significant reduction in memory usage. Despite the advantage of memory usage, there is no notable difference in the run time between the two versions. The recursive version can be run from the command line as the “dfsr” method.

Secondly, bidirectional A* can ensure the improvement over A* in terms of run-time, but it does not guarantee the optimal route when meeting in the middle is the terminal condition. This can be explained by the fact that the heuristic function may force the tree to expand the nodes that are closer when comparing the heuristic value between the node and the goal. However, there may be many obstacles if taking that path, so it would not be optimal. In order to counter that problem, there are some suggestions. The one implemented in the program is inspired by a study (Whangbo & Taeg-Keun, 2007). The main idea is to use a threshold mu to keep track of the path cost when finding a complete path. This value will be checked and updated whenever the program finds a path, but it just keeps the smallest value. The condition to stop the search is when it is unlikely to find a shorter path. The program compares the maximum value of the estimated cost between the node from the forward queue and backward queue with the threshold. If that value is larger than the threshold, the search is unlikely to yield a better result.

Conclusion

In general, a thorough analysis of six different search algorithms has been delivered through the report. All of the algorithms are considered within the Robot Navigation problem to help define which one would be more effective. Moreover, the report covers the keys feature implementation of the program, GUI, and console program. To define the most effective use, it should be categorised into two groups, uninformed and informed search. In terms of uninformed search, the bidirectional search can help reduce both time and space complexity, which is suitable for a complicated and large case. However, if the developer wants to prioritise the memory usage over the shortest path, DFS recursive would be an appropriate solution. With regard to informed search, Bidirectional A* should be applied with the improvement initiative to find the optimal path instead of the terminal condition of meeting in the middle. This way would reduce the run time significantly. To improve efficiency, multi-threading could take advantage to boost the performance, especially in the case of GUI version.

Acknowledgement/Resources

The textbook of *Artificial Intelligence: A modern approach* by Russel has helped me gain an insight into how the search algorithm works and profoundly understand the disadvantages

and advantages of each of them. Moreover, the research by Whangbo and Taeg-Keun gave me the idea of how to improve the Bidirectional A* to make it find the optimal path.

Reference

1. Russell, S.J. and Norvig, P. (2010) *Artificial Intelligence: A modern approach*. Upper Saddle River: Prentice-Hall.
2. Whangbo, T.-K. (2007) "Efficient modified bidirectional A * algorithm for optimal route-finding," *New Trends in Applied Artificial Intelligence*, pp. 344–353. Available at: https://doi.org/10.1007/978-3-540-73325-6_34.