

OCTOBER 10, 2021



ESP3201

SEARCH MINI-ASSIGNMENT

GOH KHENG XI, JEVAN

A0199806L

NUS Engineering

Contents

| | |
|---------------------------------------|----|
| 1. Introduction | 2 |
| 2. Code | 2 |
| 3. Breadth-First Search (BFS) | 2 |
| 3.1 Characteristics..... | 2 |
| 3.2 Data structure and algorithm..... | 3 |
| 3.3 Time and space complexity..... | 3 |
| 4. Depth-First Search (DFS) | 3 |
| 4.1 Characteristics..... | 3 |
| 4.2 Data structure and algorithm..... | 4 |
| 4.3 Time and space complexities | 4 |
| 5. Uniform-Cost Search (UCS) | 4 |
| 5.1 Characteristics..... | 5 |
| 5.2 Data structure and algorithm..... | 5 |
| 5.3 Time and space complexities | 5 |
| 6. A-Star Search (A*) | 5 |
| 6.1 Heuristic functions | 6 |
| 6.2 Characteristics..... | 7 |
| 6.3 Data structure and algorithm..... | 7 |
| 6.4 Time and space complexities | 7 |
| 7.Results..... | 7 |
| 7.1 Single Objective..... | 7 |
| 7.2 Multiple Objectives | 9 |
| 7.3 Other findings | 10 |
| Conclusion..... | 11 |
| 7. References | 12 |

1. Introduction

Search algorithms are techniques for finding a solution to achieve the goal state from the start state efficiently [1]. Common applications of search algorithms include pathfinding, games and machine learning [1].

This assignment explores 4 search algorithms which are breadth-first search (BFS), depth-first search (DFS), uniform-cost search (UCS) and A-Star search (A*). The skeleton codes are given for the programme but the functions corresponding to each of the search algorithm are to be completed.

2. Code

The mp1.py file contains the main() function that is executed when the file is ran. Maze.py provides the API to the Maze class which have the following methods:

1. `getStart()`: returns a tuple of x and y coordinates that belongs to the start position denoted by a capitalised 'P' on the map.
2. `getObjectives()`: returns a list of tuples representing the x and y coordinates of each objectives.
3. `getNeighbors(row, col)`: returns a list of tuples that represents the valid immediate up, down, left and right coordinates relative to the coordinates passed to the function. Valid coordinates are coordinates that are within the boundary of the map and are not occupied by obstacles.

3. Breadth-First Search (BFS)

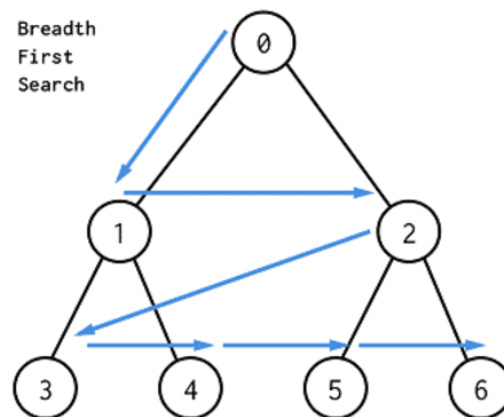


Figure 1. BFS algorithm

BFS is a uniformed or blind search where the algorithm has no additional information about the state space other than the given domain [2]. In this search, the agent explores the neighbouring nodes of the shallowest nodes first before exploring the nodes in the deeper layer as seen in Figure 1 [2].

3.1 Characteristics

BFS is complete which means it guarantees a solution should one exists, after expanding all the nodes shallower than the goal nodes.

Hence, BFS is also optimal if the cost function is a non-decreasing function of the goal node's depth. In this assignment, the cost function is set to be the number of nodes traversed from the starting to the current node and hence, the BFS algorithm implemented in this assignment is optimal.

3.2 Data structure and algorithm

A queue data structure is suitable for BFS as it follows a First-In-First-Out (FIFO) approach where the first set of paths is removed from the front of the data structure and a list of paths are generated by adding each valid neighbouring nodes into the removed path [3]. The new list of paths is then pushed into the back of the data structure and the algorithm continues until the goal state is found.

As nodes are expanded, the visited nodes are to be recorded as there is no need to explore visited nodes again, improving the space and time complexity of the algorithm. The python dictionary data structure which uses a hash table is suitable for this operation as it provides a fast look up of $O(1)$ complexity.

3.3 Time and space complexity

Assuming the maximum number of valid neighbouring nodes for each node, as the agent traverse through the graph (map), the number of nodes explored will be multiplied by the branching factor (b), which is the number of neighbouring nodes for each node, for each depth layer, d . Hence the time complexity for the algorithm is $b + b^2 + b^3 + \dots + b^d = O(b^d)$ or $O(n)$, where n is the number of nodes, assuming that every expansion is a $O(1)$ operation.

Since, every visited node are to be stored in the memory, the space complexity for the algorithm is $O(b^d)$ as well.

4. Depth-First Search (DFS)

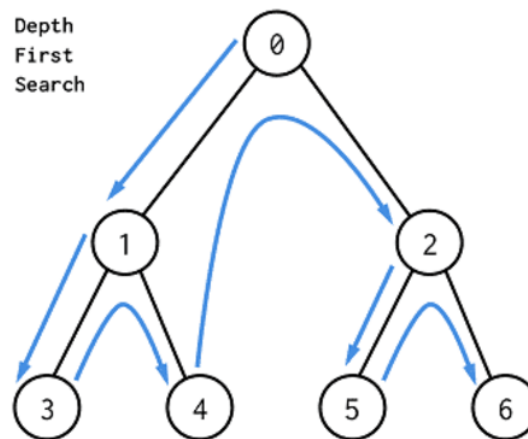


Figure 2. DFS algorithm

DFS is also a uniformed search algorithm the agent explores as far into a branch as possible until it reaches the deepest node before exploring the other branches to find the goal node [2]. This algorithm is illustrated in Figure 2 above.

4.1 Characteristics

As the agent is always exploring the frontier nodes, DFS is only complete if the environment is constrained by a boundary which is the case in this assignment and only in a graph search where

the visited nodes are stored, otherwise, the agent will be exploring the same branch for an infinite amount of time.

DFS is also not optimal, as it returns the first solution found according to the order which the branches of the node are explored which may or may not be the shallowest or shortest path to reach the goal state.

4.2 Data structure and algorithm

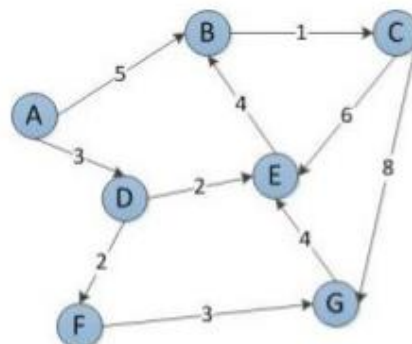
A stack data structure is suitable for DFS as it follows a Last-In-First-Out (LIFO) approach where the last set of paths is removed from the back of the data structure and a list of paths are generated by adding each valid neighbouring nodes into the removed path [3]. The new list of paths is then pushed into the back of the data structure and the algorithm continues until the goal state is found.

Similar to BFS, the visited nodes are to be recorded as the environment is explored to improve the time and space complexity of the algorithm.

4.3 Time and space complexities

The time complexity of DFS is $O(b^m)$ where m is the maximum depth of the environment. Its space complexity is also $O(bm)$ due to the need to store every visited node.

5. Uniform-Cost Search (UCS)



Solution
Explored : A D B E F C
path: A to D to F to G
Cost = 8

Figure 3. UCS algorithm

UCS is another uniformed search algorithm where the agent explores the node with the least cumulative cost also known as the weight of the edges from the starting position iteratively until the goal node is found as depicted in Figure 3 [2]. In this assignment, the cost of expanding a node is assumed to be constant and hence, the cumulative cost of a node is simply the number of nodes in the path which corresponds to the number of coordinates returned in the list by the `ucs()` function in the programme.

5.1 Characteristics

If the cost of each edge of the graph is homogeneous as assumed in this assignment, UCS behaves identically to BFS except that it checks for the minimal cost when it reaches the goal instead of returning the goal path. Hence, it is a complete search algorithm if the cost in each expansion is positive. It is also an optimal search algorithm as the path with the smallest cumulative cost is chosen for every expansion.

5.2 Data structure and algorithm

A priority queue data structure that is built upon a binary heap is suitable for BFS as it allows the enqueueing of item in $O(\log n)$ and the dequeuing of the highest priority item in $O(\log n)$ [4]. The priority in UCS is the lowest cost function item which in this assignment, is the number of coordinates in the path.

After the highest priority path which corresponds to the lowest cost path is dequeued from the priority queue, a list of paths is generated by adding each valid neighbouring nodes that are not visited into the removed path. For visited nodes are to be stored in a separate container which is preferably a hash table like the python dictionary. The new list of paths is then enqueued back into the priority queue and the algorithm continues until the goal state is found.

5.3 Time and space complexities

The number of expansions performed in the optimal path is $\frac{C}{\epsilon} + 1$ where C is the cumulative cost from the initial state to the end state in the solution, ϵ is the minimum additional cost of expanding a node and the additional 1 value represents the initial step to get to the initial state. Since for each layer, the number of expansions performed is multiplied by the maximum number of branches per node, the time complexity is $O(b^{\frac{C}{\epsilon}+1})$.

In this assignment where the cost of expanding each node is assumed to be 1, the number of expansions performed in the solution path, $\frac{C}{\epsilon} + 1$ is equals to the number of steps in the optimal path, d . Hence the time complexity of the implemented UCS is $O(b^d)$.

Since, visited nodes are also stored, the space complexity for this algorithm is $O(b^{\frac{C}{\epsilon}+1})$ which in the UCS implemented in this assignment is $O(b^d)$.

6. A-Star Search (A*)

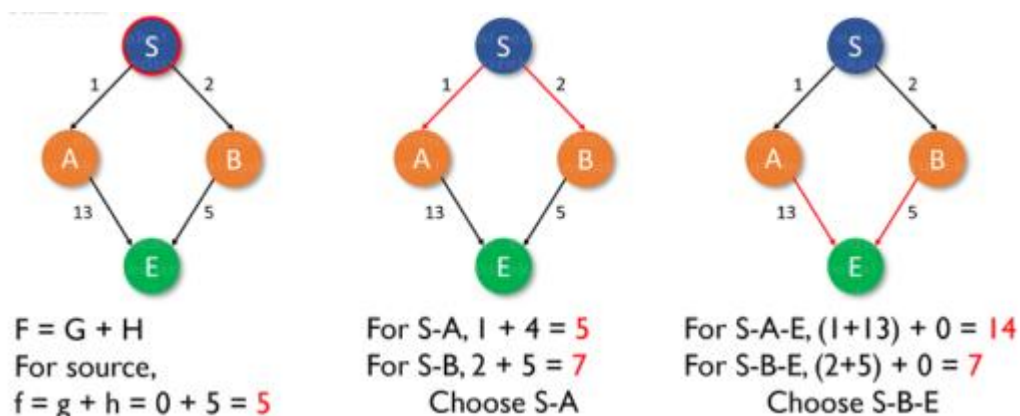


Figure 4. A* search algorithm

Unlike BFS, DFS and UCS, A* search is an informed search which means that it chooses the expansion of nodes in a heuristic approach until the goal is found [5]. In this search method, the cost function is $f(n) = g(n) + h(n)$ where $g(n)$ is the cost function of the node from the initial state and $h(n)$ is a heuristic function that estimates the cost of the of the goal node from that node [5]. This is shown in Figure 4 above.

Since the cost of expanding a node is assumed to be 1, $g(n)$ is simply the number of steps from the starting node to the current node. $h(n)$ should be a function that tries to model the actual optimal cost from the node to the goal state as much as possible. Common functions include the Euclidean distance the Manhattan distance and the from the node to the objective, the Manhattan distance and the Chebyshev distance [6].

6.1 Heuristic functions

The Minkowski distance is the general form of the Euclidean, Manhattan and the Chebyshev distances [7], and its formula is given by:

$$D = \left(\sum_{i=1}^n |x_i - y_i|^p \right)^{\frac{1}{p}}$$

Where x and y are the i -coordinate of the goal position and the current node, and n is the dimension of the state space.

Manhattan Distance

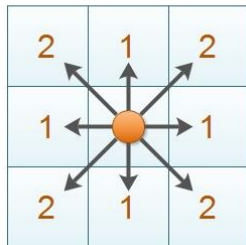


Figure 6. Manhattan Distance

Euclidean Distance

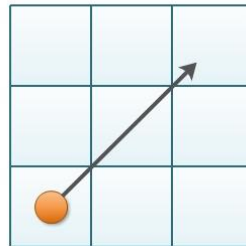


Figure 5. Euclidean Distance

Chebyshev Distance

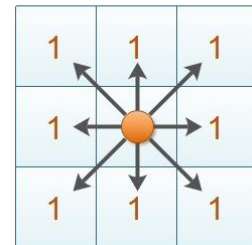


Figure 7. Chebyshev Distance

When $p = 1$, the Minkowski distance is the Manhattan distance between the node and the goal position which corresponds to the formula $D = |x_1 - y_1| + |x_2 - y_2|$. Figure 5 shows the visual representation of the Manhattan distance between 2 nodes.

When $p = 2$, the Minkowski distance is the Euclidean distance between the two nodes which corresponds to the formula $D = \sqrt{|x_1 - y_1|^2 + |x_2 - y_1|^2}$. Figure 6 shows the visual representation of the Euclidean distance between 2 nodes.

The limit of the Minkowski function as p tends to infinity is the Chebyshev distance that corresponds to the formula $D = \lim_{n \rightarrow \infty} (\sum_{i=1}^n |x_i - y_i|^p)^{\frac{1}{p}}$. It is analogous to the movements in a game of chess where diagonal movement of a chess piece is considered as a single step.

In this assignment, the agent can only move up, down, left and right, hence, $h(n)$ is chosen to be the Manhattan distance as Euclidean and Chebyshev distances will cause a larger underestimation of the actual cost.

6.2 Characteristics

A-star search is complete and optimal if the heuristic function does not overestimate the cost of the goal from each node and the heuristic function is always equal to or smaller than the number of steps for the node to reach the goal.

6.3 Data structure and algorithm

The priority queue is also suitable for A* search where the highest priority item is the path with the least $f(n)$ cost. As the highest priority path is dequeued from the data structure, a list of paths is generated by adding each valid neighbouring node that are not visited into the dequeued path. The list of newly generated set of paths is enqueued back into the priority queue and the newly visited nodes are stored in a dictionary. The algorithm continues until the goal state is found.

6.4 Time and space complexities

Time complexity of A* search is proportional to the number of expansions perform and hence it is of $O(b^{\epsilon d})$ time complexity where ϵ is the relative error of heuristics given by $\frac{h^* - h}{h^*}$ where h^* is the heuristic cost of a path and h is the optimal cost. As the visited nodes are also stored in a container, the space complexity is $O(b^d)$.






7.Results














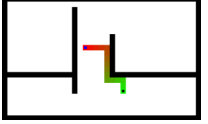

Each of the 4 search algorithms are then tested on all the single-objective map and the multiple-objectives map to observe the different solutions provided by each algorithm and study their efficiency.

7.1 Single Objective

There are a total of 4 single-objective mazes given which are the small, medium, large sized mazes and an open maze with larger open area. In addition, a custom-designed maze was crafted to allow for more comparison between the search algorithms.

Table 1. Single-objective results

| | Small Maze | Medium Maze | Big Maze | Open Maze | Customed Maze |
|-----|--|--|--|--|---|
| BFS |  <p>Path Length: 20</p> <p>States Explored: 90</p> <p>Total time 0.0s</p> |  <p>Path Length: 69</p> <p>States Explored: 267</p> <p>Total time 0.000500s</p> |  <p>Path Length: 211</p> <p>States Explored: 616</p> <p>Total time 0.00150s</p> |  <p>Path Length: 16</p> <p>States Explored: 182</p> <p>Total time 0.0s</p> |  <p>Path Length: 90</p> <p>States Explored: 499</p> <p>Total time 0.00100s</p> |

| | | | | | |
|-----|---|--|--|---|---|
| DFS |  Path Length: 50 States Explored: 73 Total time 0.0s |  Path Length: 131 States Explored: 144 Total time 0.000501s |  Path Length: 211 States Explored: 426 Total time 0.00100s |  Path Length: 42 States Explored: 433 Total time 0.00100s |  Path Length: 160 States Explored: 227 Total time 0.000500s |
| UCS |  Path Length: 20 States Explored: 91 Total time 0.000500s |  Path Length: 69 States Explored: 268 Total time 0.00100s |  Path Length: 211 States Explored: 619 Total time 0.00150s |  Path Length: 16 States Explored: 194 Total time 0.000500s |  Path Length: 90 States Explored: 496 Total time 0.00100s |
| A* |  Path Length: 20 States Explored: 52 Total time 0.0s |  Path Length: 69 States Explored: 219 Total time 0.000500s |  Path Length: 211 States Explored: 542 Total time 0.00150s |  Path Length: 16 States Explored: 15 Total time 0.0s |  Path Length: 90 States Explored: 383 Total time 0.00100s |

From table 1, it can be observed that while the 4 search algorithms are complete, not all of them are optimal. BFS, UCS and A* search always gave the optimal path with the least path length, while DFS always overestimates the optimal cost. States explored is an estimate for the space complexity while total time is an estimate for the time complexity of each search algorithm.

In most environment, DFS is very space efficient relative to BFS and UCS except for the open maze where it makes many redundant searches in the open area. Furthermore, it takes a shorter to find a solution or at worst similar time than the other algorithms except in the open environment where it took a longer time due to the extra redundant exploration.

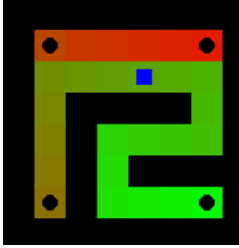


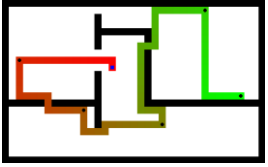
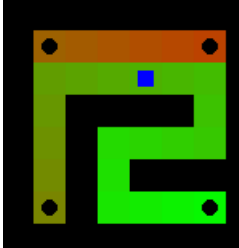



As mentioned in section 5.1, since the cost of expanding each node is constant in this assignment, UCS behaves identically to BFS and therefore they both produce very similar results in all the mazes. They both produce an optimal solution with similar number of states explored. The difference in the solutions (but still optimal) produced by the 2 algorithms is due to the use of priority queue in UCS where the paths are not sorted stably which means that paths of the same cost are not guaranteed to keep their order in the data structure. Furthermore, it can be observed that UCS took a longer time than BFS in general to produce a solution. This is once again attributed to the use of priority queue where the enqueueing and dequeuing operations are of $O(\log n)$ compared to the use of normal queue in BFS where the operations are $O(1)$. Additionally, computational resources are required to import the heapq module at the start of the function.

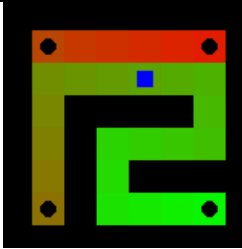


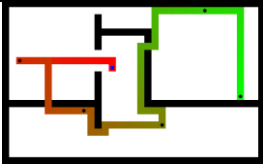
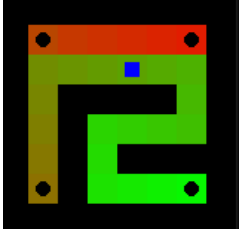


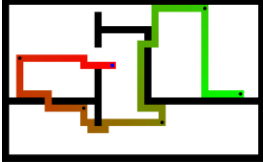
A* search is more space efficient than BFS and UCS and in certain mazes, such as the small maze and the open maze, it took lesser explorations than DFS to find the optimal solution. In all the mazes, it also took the least amount of time, or at most similar time relative to the 3 other algorithms to find the solution, despite the $O(\log n)$ operations of priority queue and the need to import heapq module.

7.2 Multiple Objectives

There are a total of 3 multi-objectives mazes given which are the tiny, medium and big sized corner mazes. Similarly, a customised maze with multiple objectives was crafted for more comparison between the search algorithms. The customised map takes inspiration from the open maze given in the single objective mazes as it allows the comparison of the search algorithm in an openly spaced environment.

Table 2. Multi-objectives results

| | Tiny Corners | Medium Corners | Big Corners | Customised |
|-----|---|---|--|--|
| BFS |  Path Length: 36 States Explored: 52 Total time 0.000499s |  Path Length: 110 States Explored: 349 Total time 0.000500s |  Path Length: 166 States Explored: 1198 Total time 0.00200s |  Path Length: 95 States Explored: 1340 Total time 0.00200s |
| DFS |  Path Length: 42 States Explored: 39 |  Path Length: 201 States Explored: 374 Total time 0.000501s |  Path Length: 338 States Explored: 637 |  Path Length: 448 States Explored: 508 Total time 0.00150s |

| | | | | |
|-----|---|---|--|--|
| | Total time 0.0s | | Total time 0.00100s | |
| UCS |  Path Length: 36 States Explored: 53 Total time 0.0s |  Path Length: 110 States Explored: 347 Total time 0.000500s |  Path Length: 166 States Explored: 1176 Total time 0.00200s |  Path Length: 95 States Explored: 1347 Total time 0.00250s |
| A* |  Path Length: 36 States Explored: 40 Total time: 0.0s |  Path Length: 110 States Explored: 248 Total time 0.00100s |  Path Length: 166 States Explored: 347 Total time 0.00100s |  Path Length: 95 States Explored: 356 Total time 0.00150s |

From table 2, it is observed once again that all 4 algorithms are complete and they are optimal with the exception of DFS as shown by the longer path length of the solution.

Similar to the single-objective results, DFS took little exploration and time to find a solution and managed to outperform BFS and UCS in the custom-design-maze of a relatively more open environment than the other 3 mazes. However, the solution is overestimated drastically as evident by the path length in the medium corners, big corners and the custom-designed maze. It is also observed to performed the worst in the medium corners maze.


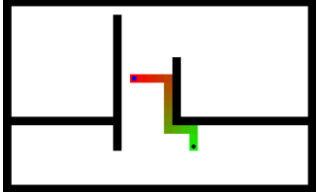
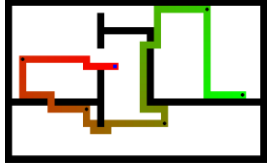

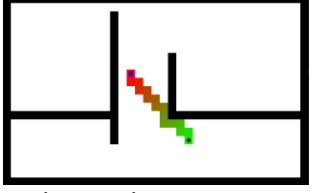
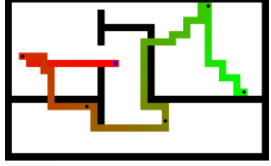
BFS and UCS performed similar to each other but took the most exploration and time to produce an optimal solution with the exception of the medium size map where DFS performed the worst.

A* search is able to produce optimal solutions in the multi-objective mazes which gives it an advantage over DFS. Furthermore, it has the best performance in terms of states explored and time taken to produce a solution.

7.3 Other findings

For A* search, different heuristics are tested, and the results were compared. It is found that positive scaling of the heuristic function has no implications on the results (i.e $f(n) = g(n) + k \cdot h(n)$ where $k \in \mathbb{R}^+$) and the exact path, time taken and number of states explored remain the same. However, a change of heuristic function does produce different results.

Table 3. Change of heuristic function

| | Large maze | Open maze | Customed-designed multi-objective maze |
|--------------------|---|---|---|
| Manhattan Distance |  <p>Path Length: 211</p> <p>States Explored: 542</p> <p>Total time 0.00150s</p> |  <p>Path Length: 16</p> <p>States Explored: 15</p> <p>Total time 0.0s</p> |  <p>Path Length: 95</p> <p>States Explored: 356</p> <p>Total time 0.00150s</p> |
| Euclidean Distance |  <p>Path Length: 211</p> <p>States Explored: 556</p> <p>Total time 0.00150s</p> |  <p>Path Length: 16</p> <p>States Explored: 60</p> <p>Total time 0.0s</p> |  <p>Path Length: 95</p> <p>States Explored: 578</p> <p>Total time 0.00300s</p> |

From table 3, an underestimate of the heuristic function may indeed produce a different set of paths as seen in the open maze and the customed designed maze. Although it still produce optimal results, the states explored increased significantly especially for the open and customed-designed mazes in addition to the increased time taken due to the additional exploration.

Conclusion

Different search algorithms are suitable for different tasks and it is instrumental to choose the appropriate algorithms for them. If the goal is not far from the starting position like in the open maze scenario, it might be more practical to use BFS rather than DFS, but if the problem allows for many actions (i.e. many branches for each node), BFS might not be appropriate due to the huge memory required. If the cost of each action is not uniform, UCS might be a better choice as it ignores the steps and finds the solution that is most optimal in the defined cost function.

From the results in this assignment, the A* search is seemingly a perfect algorithm; however, its performance is highly dependant on the estimate of the heuristic cost. If the heuristic function over-estimates or underestimates the optimal cost severely, it might take a longer time and more exploration to produce a solution.

7. References

- [1] Tutorials Point, "AI - Popular Search Algorithms," Tutorials Point, [Online]. Available:
] https://www.tutorialspoint.com/artificial_intelligence/artificial_intelligence_popular_search_algorithms.htm. [Accessed 10 October 2021].
- [2] S. J, "Analytics Vidhya," Data Science Blogathon, 7 February 2021. [Online]. Available:
] <https://www.analyticsvidhya.com/blog/2021/02/uninformed-search-algorithms-in-ai/>. [Accessed 10 October 2021].
- [3] Simplilearn, "The Ultimate Guide to Stacks And Queues Data Structures," Simplilearn, 15 September 2021. [Online]. Available: <https://www.simplilearn.com/tutorials/data-structure-tutorial/stacks-and-queues>. [Accessed 10 October 2021].
- [4] Geeks for geeks, "Priority Queue | Set 1 (Introduction)," Geeks for geeks, 28 June 2021. [Online].
] Available: <https://www.geeksforgeeks.org/priority-queue-set-1-introduction/>. [Accessed 10 October 2021].
- [5] B. Roy, "A-Star (A*) Search Algorithm," Towards Data Science, 29 September 2019. [Online].
] Available: <https://towardsdatascience.com/a-star-a-search-algorithm-eb495fb156bb>. [Accessed 10 October 2021].
- [6] A. Patel, "Heuristics," Stanford Theory, [Online]. Available:
] <http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html>. [Accessed 10 October 2021].
- [7] National Institute of Standards and Technology, "MINKOWSKI DISTANCE," National Institute of Standards and Technology, 31 August 2017. [Online]. Available:
] <https://www.itl.nist.gov/div898/software/dataplot/refman2/auxillar/minkdist.htm>. [Accessed 10 October 2021].