2023

ASSIGNMENT 1 REPORT

INTRODUCTION TO ARTIFICIAL INTELLIGENCE (COS30019)

ANH TUAN DOAN

Student ID: 103526745
Tutorial class: Tuesday (12:30pm)

Assignment 1 Report

Anh Tuan Doan (Student ID: 103526745)

April 2023

Contents

1 Instruction

The implemented program has both the command line interface and a graphical user interface that allow the user to create a new maze and see the path-finding process happens in real-time. In order to use the command line interface, the user can type the command: *search* <filename> <method> (where "filename"

is the path to a .txt file that specifies the maze configuration including start, goals, and wall squares, and "method" is the search algorithm that will be used in the path finding process). There are 6 different methods, including Breadth-first search (BFS), Depth-first search (DFS), Greedy Best first search (GBFS), A* Search (AS), Bidirectional Search (CUS1), Iterative Deepening A* Search (CUS2) (Note: when running the program, you only need to specify the abbreviation of the algorithms, not the full name). If the arguments are valid, the program will run the search process and produce the sequence of moves from the start node to one of the goal nodes or the message "No solution found" if there is no solution.

On the other hand, the user can use the graphical interface to create a new maze configuration file and see the search process visually. To create a new maze, the user has to specify the row, column, start, goals, and wall cells. To select a cell, the user can simply click on a square in the maze and press the confirm button. Additionally, for the goals and walls, the user can select multiple squares and remove squares by clicking on the already selected cells.

After creating the maze, the user can save the file and export it to a location on the computer. Next, the user can go back to the homepage to import the maze.txt file that is just created and start the search process. Finally, on the search page, the user can select one of 6 algorithms to see the search process.

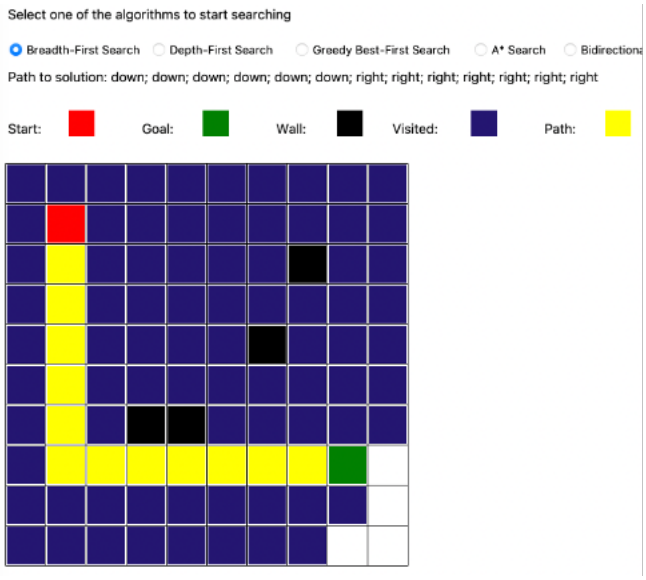

Figure 1. Graphical user interface for the program

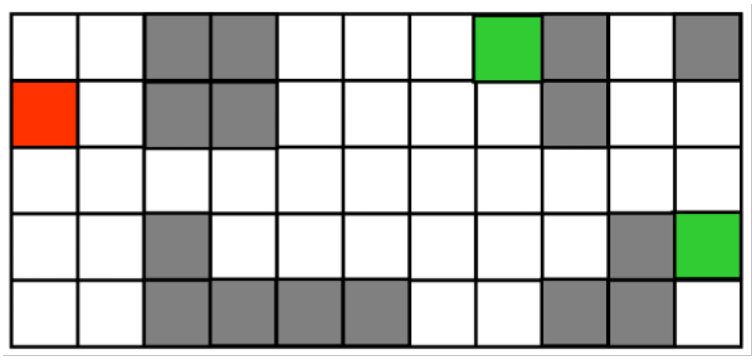

Figure 2. Robot Navigation Problem

2 Introduction

In the Robot Navigation problem, the environment is an N x M grid where N and M are both greater than 1. Moreover, there are a number of occupied wall cells that can not be reached. A robot is initially located in one of the cells and it is required to avoid the walls and find a path to reach one of the designated goals. To find a solution to this problem, we can use several search algorithms such as Breadth-first, and Depth-first search in order to find a correct path. Therefore, in this report, we will look at some common algorithms that have been used in the program and compare them based on their complexity, and memory consumption.

2.1 Glossary

- BFS: Breadth-first Search Algorithm
- DFS: Depth-first Search Algorithm
- A*: A* Search Algorithm
- GBFS: Greedy Best-First Search Algorithm
- IDA: Iterative Deepening A* Search Algorithm
- stack: a data structure that stores elements in a last-in-first-out fashion
- queue: a data structure that stores elements in a first-in-first-out fashion
- frontier: the list of nodes that will be visited in the future
- node: an item of the search tree and represent a cell in the maze.
- heuristic function: the function to calculate how likely a node will lead to the solution

- admissible heuristic function: a heuristic is admissible if it never overestimates the actual cost to get to the goal node

3 Search Algorithms

This section will look at some common search algorithms and compare their performance based on the data collected in the implemented program.

3.1 Uninformed Search Algorithm

Uninformed search algorithms are basic algorithms that can find a solution in a search tree using a brute-force method without utilizing any domain-specific knowledge to guide the search. Some common uninformed algorithms are Breadth-first search, Depth-first search, and Bidirectional Search:

- Breadth-first search (BFS): In BFS, the algorithm will start at the root node and keep expanding the shallowest unexpanded node until it finds a solution. Therefore, it will check all nodes at the same depth level before moving on to explore the nodes at the next depth. As a result, BFS can find the shortest path from the root node to the solution if the actions are unweighted. Furthermore, BFS store the nodes on the queue in a first-in-first-out principle to ensure the shallowest node is explored first. Its time complexity is $O(b^{d+1})$, where b is the branching factor, and d is the distance from the start node to the goal node. Moreover, because the BFS need to store all the nodes at the same depth; therefore, the memory consumption will be huge if the maximum width in the graph is large. Additionally, the breadth-first algorithm is complete if the branching factor is finite. When the implemented program run the BFS search with the maze configuration provided by the assignment, it explored a total of 34 nodes.

- Depth-first search (DFS): Instead of exploring the shallowest unexpanded node, the DFS will check the deepest node. The algorithm will start at the root node and recursively explore all the children nodes along the same branch as far as possible before backtracking and checking different branches. As a result, the found solution is not guaranteed to be the shortest path from the root node to the solution. To implement DFS, we can use a stack data structure in a last-in first-out faction to make sure the deepest node is explored first. For the time complexity, it is $O(b^m)$ where m is the maximum depth of the tree. For the memory consumption, because depth-first search only needs to keep track of the nodes in the chain; therefore it would typically consume less memory than BFS. In terms of completeness, DFS can get stuck in an infinite loop where it never finishes. Finally, in comparison with BFS, when the depth-first search is run with the maze configuration provided in the assignment, it explores a total of 41 nodes.

- Bidirectional search: Rather than starting at the start node and keep exploring the children nodes until we find a solution, bidirectional start searching simultaneously at both the start node and goal node and keep exploring until it finds a common node. If we use BFS as the method to explore the start and goal node, our solution will be the shortest path if the actions are unweighted. For the time complexity, as each search only need to check half of the depth, the time complexity is $O(b^{d/2} + b^{d/2})$. With the provided maze configuration, the bidirectional search explores 32 nodes which is slightly lower than BFS and DFS.

3.2 Informed Search Algorithms

On the other hand, informed algorithms use domain-specific knowledge to guide the search and find the solution; therefore, the search will be more efficient in terms of time and search space. Specifically, informed algorithms use heuristic functions to evaluate how "promising" a node is (i.e how likely that node will lead to the goal) and explore the most promising node in the current search tree. In the Robot navigation problem, we can evaluate the nodes by using the Manhattan distance which is the sum of the distance along the x and y axes between 2 points: $f(A, B) = |x_A - x_B| + |y_A - y_B|$ where $A(x_A,y_A)$, $B(x_B,y_B)$ are the coordinates of 2 points A and B. Notably, the Manhattan distance is admissible as it never overestimates the actual distance to reach the goal. The 3 common informed search algorithm that is used in the program are Greedy Best First Search, A* Search, and Iterative Deepening A* (IDA):

- Greedy Best First Search (GBFS): In GBFS, we only use the heuristic function to evaluate the nodes; therefore $f(n) = h(n)$. In particular, GBFS estimates the cost of each possible path to the goal and selects the path with the lowest cost. To do this, GBFS stores the nodes in a queue and sorts them based on their heuristic value to choose the node with the lowest value to explore. However, the solution produced by GBFS is not always the optimal path with the lowest cost as it only chooses the most "promising" node and does not consider the cost it has taken so far to reach that node from the start. The time and space complexity for GBFS is $O(b^m)$ with b as the branching factor and m as the maximum depth of the tree. When GBFS is run with the provided default maze, it expanded only 20 nodes which is the least amount of nodes in every algorithm.

- A* Search (AS): In A* algorithm, we evaluate a node by combining its heuristic function h(n) and the cost that it has taken so far to reach that node g(n); therefore $f(n) = h(n) + g(n)$. In particular, A* stores the nodes in a priority queue and sorts them based on f(n) to select the path with the lowest cost. Because A* takes the travelled distance into account, it can avoid exploring the expensive path and search for other paths with less cost. Especially, the solution produced by A* is guaranteed to be the shortest path if the heuristic function h(n) is admissible. The time complexity of A* is dependent on the structure of the graph and the heuristic

function; in the worst case, the number of expanded nodes is exponential with the path length of the solution. In addition, one drawback of the A* algorithm is the memory consumption as it stores all the generated nodes. A* search explores 28 nodes in the provided maze configuration, which is slightly higher than GBFS.

- Iterative Deepening A* (IDA): IDA search is a variant of Iterative Deepening Depth First search that uses a heuristics function to evaluate the possible paths. In particular, IDA only explores the nodes with the f(n) value smaller than the threshold number and put the nodes with higher f(n) values into a pruned list. If there are no more nodes to explore and the solution is not found, it will update the threshold value to the smallest f(n) value in the pruned list and restart the whole search again. Similar to A*, IDA will find the optimal path to the goal if the heuristic function is admissible. However, IDA will use less memory than A* as it is a depth-first search algorithm. When the implemented program runs IDA search with the provided maze configuration, it only explores 27 nodes which is the second least number out of all algorithms.

4 Implementation

In this section, we will discuss the implementation of the mentioned search algorithms in pseudocode and analyze the differences between the algorithms in more detail. Firstly, each search algorithm is implemented in a class and derived from an abstract class called SearchAlgorithm that specifies the required attributes and methods. However, each algorithm will have a different implementation and search for the solution in different ways.

4.1 Breadth-first search

As the first step, the breadth-first algorithm starts the searching process by inserting the root node into the frontier queue, and marking it as visited. Next, it removes a node in the frontier queue to check if it is a goal. If the goal is found, the program will stop, otherwise, it will expand the current node by exploring its children. For each child of the node, if the child was not visited before, the program will add it to the frontier and mark it as visited. This process will be repeated until a solution is found or the program cannot find any solution. Notably, BFS use the queue data structure to store the node in a first in first out principle to ensure the shallowest node is expanded first.

4.2 Depth-first search

The implementation of Depth-first search is very similar to Breadth-first search; however, it uses a stack to store the nodes instead of a queue. In particular, the stack works in a last-in-first-out fashion; therefore, the most recently added

Algorithm 1 Breadth First Search

```
1: procedure BFS
2:     frontier ← Queue()
3:     visited ← List()
4:     frontier.queue(start node)
5:     visited.append(start node)
6:     while frontier is not empty do
7:         node ← frontier.dequeue()
8:         if node = goal node then
9:             found solution, exit
10:        else
11:            for children of node do
12:                if children not in visited then
13:                    frontier.queue(children)
14:                    visited.append(children)
15:                end if
16:            end for
17:        end if
18:    end while
19:    no solution found, exit
20: end procedure
```

Algorithm 2 Depth First Search

```
1: procedure DFS
2:     frontier ← Stack()
3:     visited ← List()
4:     frontier.push(start node)
5:     while frontier is not empty do
6:         node ← frontier.pop()
7:         visited.append(node)
8:         if node = goal node then
9:             found solution, exit
10:        else
11:            for children of node do
12:                if children not in visited then
13:                    frontier.push(children)
14:                end if
15:            end for
16:        end if
17:    end while
18:    no solution found, exit
19: end procedure
```

node will be explored first. This allows the Depth-first search to explore the deepest node in the frontier as far as possible until it hits the maximum depth.

4.3 Bidirectional search

Algorithm 3 Bidirectional Search

```
1: procedure BidirectionalSearch
2:     sourceFrontier ← Queue()
3:     desFrontier ← Queue()
4:     sourceVisited ← List()
5:     desVisited ← List()
6:     sourceFrontier.queue(start node)
7:     sourceVisited.append(start node)
8:     desFrontier.queue(goal node)
9:     desVisited.append(goal node)
10:    while sourceFrontier and desFrontier are not empty do
11:        perform BFS on sourceFrontier
12:        perform BFS on desFrontier
13:        if sourceVisited and desVisited has a common node then
14:            found solution, exit
15:        end if
16:    end while
17:    no solution found, exit
18: end procedure
```

In the bidirectional search, we have 2 simultaneous searches, one starts from the root node while the other starts at the goal node. The program will keep searching until one of the frontiers is empty. If during the process, we find a common node in the sourceVisited and desVisited we stop the program and return the solution. If one of the frontiers is empty and there is no solution yet, we can conclude that there is no solution for the problem and stop the program. Notably, as we use the BFS algorithm for the searches, it is guaranteed that the solution will be the shortest path from the root node to the goal node.

4.4 Greedy best-first search

Algorithm 4 Greedy Best-First Search

```
1: procedure GBFS
2:     h(node) ← Manhattan Distance of node
3:     frontier ← PriorityQueue()
4:     visited ← List()
5:     frontier.queue(start node)
6:     visited.append(start node)
7:     while frontier is not empty do
8:         node ← the node with smallest h(n) value in the frontier
9:         if node = goal node then
10:            found solution, exit
11:        else
12:            for children of node do
13:                if children not in visited then
14:                    frontier.queue(children)
15:                    visited.append(children)
16:                end if
17:            end for
18:        end if
19:    end while
20:    no solution found, exit
21: end procedure
```

In informed search, we can use the heuristic function which is domain-specific knowledge to guide the search and make the process more efficient. In particular, we can assess the nodes by using the h(n) function which is the Manhattan distance from the current node to the goal. For every iteration, we evaluate every possible path and explore the most "promising" node based on the heuristic function. To implement this, the algorithm stores the nodes in a priority queue and sorts the nodes in the frontier based on their h(n) value. As a result, the search process will be more efficient than the uninformed methods. However, the GBFS algorithm does not always produce the optimal path, as it does not take the distance it takes to travel to the current node into account.

4.5 A* Search

Algorithm 5 A* Search

```
1: procedure AS
2:     f(node) ← Manhattan Distance of node + travelled cost
3:     frontier ← PriorityQueue()
4:     visited ← List()
5:     frontier.queue(start node)
6:     while frontier is not empty do
7:         newNode ← the node with smallest h(n) value in the frontier
8:         visited.append(newNode)
9:         if newNode = goal node then
10:            found solution, exit
11:        else
12:            for children of newNode do
13:                if there's a visited node with same location and f(node)<f(children) then
14:                    skip children
15:                else if there's a node in frontier with same location and f(node)<f(children) then
16:                    skip children
17:                else
18:                    frontier.queue(children)
19:                end if
20:            end for
21:        end if
22:    end while
23:    no solution found, exit
24: end procedure
```

In A* search, we evaluate a node by combining both the Manhattan distance and the total cost it has taken so far to reach that node: f(n) = h(n) + g(n). As a result, the solution produced by A* is guaranteed to be optimal as long as the heuristic function is admissible. Moreover, we also check the children node to see if it has been already visited before adding it to the frontier. If there is a node with the same location either in the visited or frontier list with a lower f(n) value than the children node, we can discard that children node and move on to the next node.

4.6 Iterative Deepening A* Search

Algorithm 6 Iterative Deepening A* Search

```
1: procedure IDA
2:     threshold ← None
3:     while True do
4:         f(node) ← Manhattan Distance of node + travelled cost
5:         prunedList ← List()
6:         frontier ← PriorityQueue()
7:         visited ← List()
8:         frontier.queue(start node)
9:         if threshold = None then
10:            threshold = f(start node)
11:        end if
12:        while frontier is not empty do
13:            newNode ← the node with smallest h(n) value in the frontier
14:            visited.append(newNode)
15:            if newNode = goal node then
16:                found solution, exit
17:            end if
18:            if f(newNode) > threshold then
19:                prunedList.apend(newNode)
20:                skip newNode
21:            else
22:                for children of newNode do
23:                    if there's a visited node with same location and
   f(node)<f(children) then
24:                        skip children
25:                    else if there's a node in frontier with same location and
   f(node)<f(children) then
26:                        skip children
27:                    else
28:                        frontier.queue(children)
29:                    end if
30:                end for
31:            end if
32:        end while
33:        if prunedList is empty then
34:            no solution found, stop program
35:        else
36:            threshold ← smallest f(n) value in prunedList
37:        end if
38:    end while
39: end procedure
```

Iterative Deepening A* search is a variant of Iterative Deepening Depth-first search and uses a heuristic function to evaluate the node. In particular, IDA only explores the node whose h(n) value is smaller than a threshold number and adds all the nodes with the higher value into a pruned list. The algorithm will keep searching until the frontier is empty and if the solution is not found and the prunedList is not empty, it will update the threshold value to be the smallest f(n) value in the prunedList and restart the whole search process again until it finds a solution. If both the frontier and prunedList are empty, then we can conclude that there is no solution for the problem and stop the program.

5 Features/Bugs/Missing

For the assignment, the program has implemented all the required features as well as a graphical interface to allow the user to create, import maze configuration and see the search process.

- Breadth-first search algorithm: the uninformed algorithm will produce a sequence of moves to reach the goal node from the start node
- Depth-first search algorithm: the algorithm will produce a solution for the maze if one exists.
- Greedy best-first search algorithm: an informed search algorithm that will evaluate the possible paths using a heuristic function and return a solution if one exists
- A* search algorithm: an informed search algorithm that is guaranteed to produce the shortest path.
- Uninformed custom search (Bidirectional search): This algorithm will return the shortest solution from the start node to the first goal node specified in the maze file.
- Informed custom search (Iterative Deepening A* Search): the algorithm will produce the shortest path to reach one of the goals
- Command line interface: given the maze file, and the search method, this program will return a series of actions to reach the goal node from the start node if one exists.
- Create a new maze configuration using a graphical user interface: the user can create a new maze configuration using a graphical interface
- Import maze configuration using a graphical user interface: the user can also import an existing maze configuration to the GUI program
- Visualize the search process using a graphical user interface: Finally, the user can see the search process happens in real-time using the graphical interface

6 Research

In addition to the command line interface, the program also includes a graphical interface that allows the user to create new maze configurations, import existing maze files and see the search process happening in real-time. In terms of implementation, the graphical interface program was written entirely in Python and heavily uses the Tkinter library which is a cross-platform framework that provides a fast and easy method to create graphical applications. In this section, we will discuss the main functionalities of the GUI program and provide some screenshots to demonstrate them.

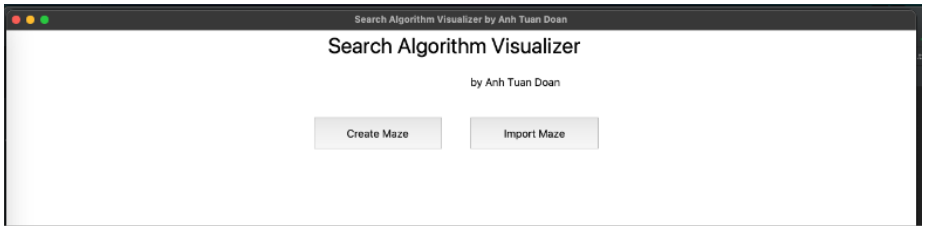

Figure 3. Homepage

On the homepage, the user can either create a new maze configuration or import an existing maze file to the program. If the user clicks on the "Create Maze" button, they will have to specify the maze configuration including the number of rows, column, start, goals, and wall squares.

After specifying the required information, the user can click on the "save" button which will open a save dialog box that allows the user to specify where they want to save the maze configuration. Next, to solve the maze and start the searching process, the user needs to go back to the homepage and import the .txt file they just created.

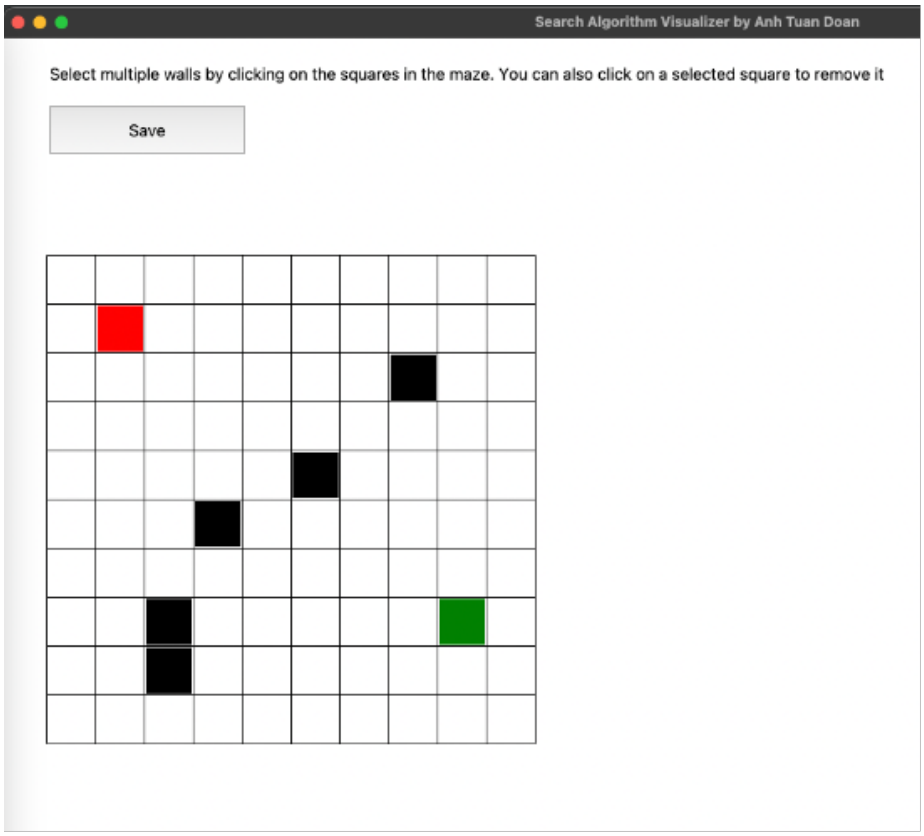

Figure 4. Create maze configuration

After importing the maze configuration, the user can finally see the search process by pressing the "Search" button. Now, the user can select one of the six algorithms including Breadth-first, Depth-first, Greedy best-first, A*, Bidirectional, and the Iterative Deepening A* algorithms to see the searching process happens in real-time. If the algorithm finds a solution, the sequences of moves to reach the goal node will be printed out on the screen; otherwise, a message saying "No solution found" will be displayed:

Figure 5. Search process in real-time

7 Conclusion

Every search algorithm has its own benefits and drawbacks and can be applied to different scenarios. However, in a problem such as Robot Navigation, A* algorithm would be an ideal choice as it can use a heuristic function such as the Manhattan distance to guide the search and reduce time and space complexity. Moreover, A* can also find the shortest path to the goal if the heuristic function is admissible. In conclusion, the A* algorithm would be the best method to use in this application as it can find the shortest path in a reasonable amount of time. To improve the performance for A*, we should use a priority queue to store the nodes in the frontier instead of a normal list. If a normal list is used, every time

we need to get the value with the lowest f(n) value, we need to iterate through the list and search for the minimum value. However, when a skew heap priority queue is used, it will improve the performance of the algorithm, especially when the frontier queue keeps increasing.

8 Acknowledgements/Resources

Below are the list of resources used for this assignment:

- `https://www.geeksforgeeks.org/iterative-deepening-a-algorithm-ida-artificial-intelligence/`: I use this resource to understand the Iterative Deepening A* search algorithm.
- `https://www.geeksforgeeks.org/a-search-algorithm/`: I use this resource to understand how A* algorithm works
- `https://realpython.com/python-gui-tkinter/`: I use this resource to learn about the Tkinter framework in order to create the GUI application.
- `https://stackoverflow.com/questions/10393886/tkinter-and-time-sleep`: I use this resource to learn how to delay time in Tkinter library.
- `https://takinginitiative.wordpress.com/2011/05/02/optimizing-the-a-algorithm/`: I use this resource to learn how to optimize the A* algorithm.

9 References

- GeeksforGeeks (no date) A* Search Algorithm. Available at: `https://www.geeksforgeeks.org/a-search-algorithm/`.
- GeeksforGeeks (no date) Iterative Deepening A* algorithm (IDA*) – Artificial intelligence. Available at: `https://www.geeksforgeeks.org/iterative-deepening-a-algorithm-ida-artificial-intelligence/`.
- Real Python (2023) Python GUI programming with Tkinter. Available at: `https://realpython.com/python-gui-tkinter/`.
- StackOverflow (2012) Tkinter and time.sleep. Available at: `https://stackoverflow.com/questions/10393886/tkinter-and-time-sleep`.
- Bobby (2011) Optimizing the A* algorithm, Taking Initiative. Available at: `https://takinginitiative.wordpress.com/2011/05/02/optimizing-the-a-algorithm/`.