COS30045 2024

Semester 1

Assignment 1 - Tree Based Search Report

104093389

Lee Chi Kit Aiken

Table of contents (TOC)

Instructions

This program implements 6 tree-based search algorithms in software to search for solutions to the Robot Navigation problem. Including Breath First Search, Depth First Search, Greedy Best First Search, A Star, Iterative Deepening Depth First Search, and Hill Climbing Search.

To use this program, we need to use a DOS command-line interface and execute the scripts. We need to go to the directory and type python search.py <filename> <method>. Notice that the method is in short form to save time by not needing to type in full form and it is not case-sensitive. It is listed below.

Full form	**Short form**
Breath First Search	bfs
Depth First Search	dfs
Greedy Best First Search	gbfs
A Star Search	as
Iterative Deepening Depth First Search	iddfs
Hill Climbing Search	hc

For example, if I want to execute an environment txt file that is in the same directory as the scripts name “RobotNav-test.txt” with Breath First Search. I will need to go to the directory and type **python search.py RobotNav-test.txt bfs** then execute it in the DOS command-line interface.

After execution, two things will appear. First, you will get an output in the terminal and a GUI display in the window.

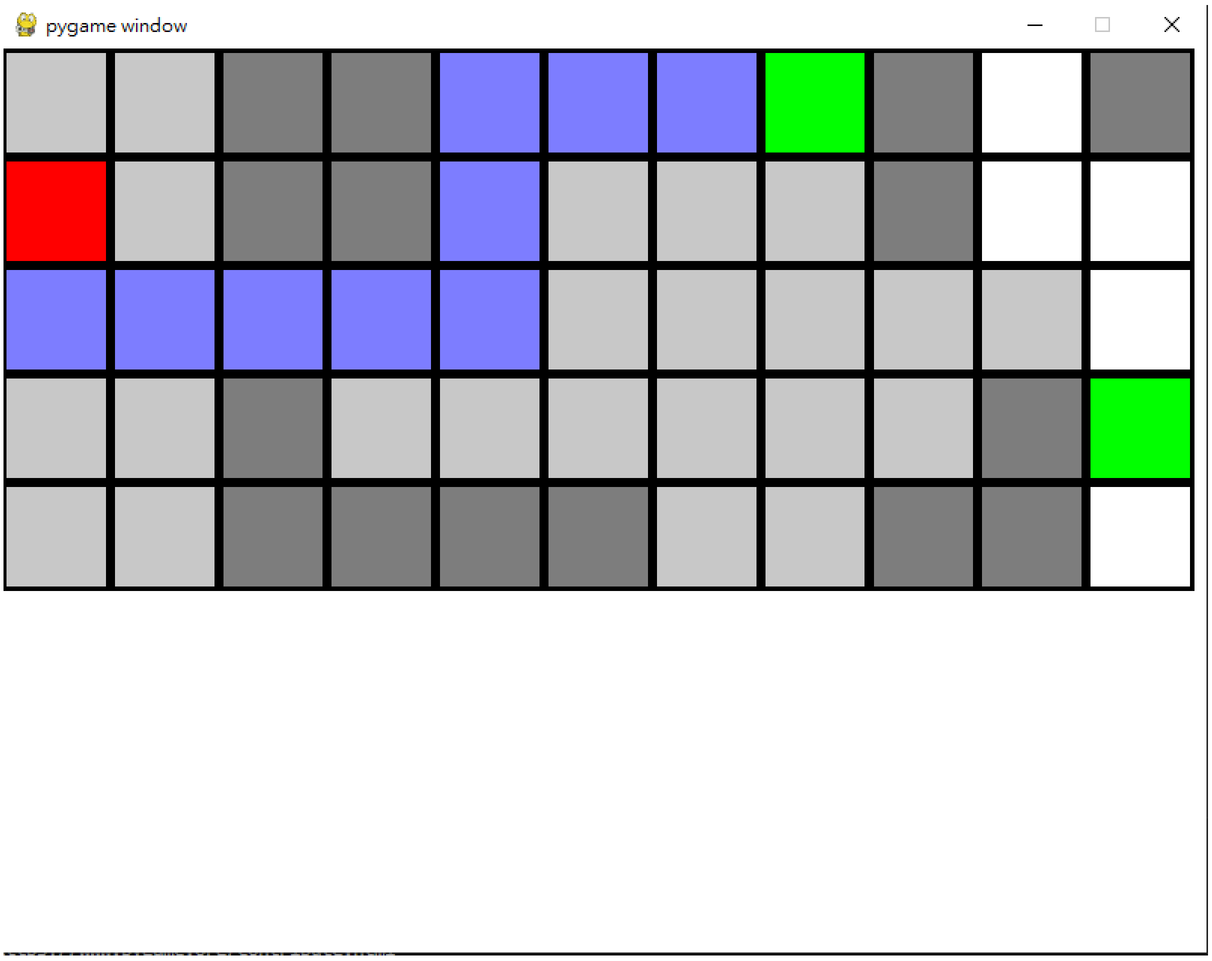
When a goal can be reached, the output will be:
filename method
goal number_of_nodes
path
When a goal can not be reached, the output will be:
filename method
No goal is reachable; number_of_nodes

With the exception of Iterative Deepening Depth First Search because it will also display the depth it went.

For the GUI, it will display a grid with the text file in blocks. Different color blocks represents different things.

Color	Meaning
White	The block is a path that can be reached
Deep Gray	The block is a wall that can not be reached
Green	The block is a goal that can be reached and will end the search once reached
Light Gray	The block is in the visited list, meaning the search algorithm once visited that node
Blue	The block is in the path list, meaning the search algorithm needs to visit that node to get to the goal
Red	The block is the starting point

The GUI will first display the grid that has goals, walls, and starting points. Then the GUI will showcase the visited list that the search algorithm visited. Finally, if the search algorithm finds a path, it will stop visiting nodes and show the path to get to the goal. **NOTICE THAT PATH WON'T BE DRAWN IF THERE IS NO SOLUTION.**
(if the display is too fast, we can change the delay number in the drawgui class. The higher the number is, the slower the block draws.)



Introduction

Robot Navigation Problem

The Robot Navigation problem involves guiding a robot through a defined space that is represented as a grid, to reach one or more specified destinations without collisions. In our scenario, the environment is an N×M grid where walls obstruct certain cells, and specific cells are designated as starting and goal points. The robot, starting from an initial position, must find a viable path to one of the designated goal cells, overcoming spatial constraints posed by walls and boundaries.

Tree-Based Search Algorithms

Search algorithms can be broadly categorized into uninformed (or blind) and informed (or heuristic-driven) methods. Uninformed search algorithms do not possess information about the goal's location relative to the current position. They explore paths indiscriminately. Informed search algorithms, however, utilize heuristics to gauge the desirability of paths, focusing efforts on more promising routes likely to lead to the goal.

Glossary	Explanation
Grid	The structured space is divided into nodes, through which the robot navigates.
Node	A point in the grid represents a possible position of the robot.
Step	The action of moving from one node to an adjacent node within the grid.
Path	A sequence of steps taken to move from the start node to the goal node.
Cost	The value assigned to traversing from one node to another, often used to determine the most efficient path.
Heuristic	A function used in certain search algorithms to help prioritize which nodes to explore next based on an estimated cost to reach the goal from each node.
Wall	A node that cannot be traversed, represents physical barriers in the robot's environment.
Frontier	A set of nodes in a search algorithm that represents the boundary of the explored area. These nodes have been reached but not yet expanded, containing potential paths for future exploration.

Search Algorithms

The core objective of employing search algorithms in AI is to traverse a problem space efficiently and effectively to find a solution. For the Robot Navigation Problem, we should be looking at the time complexity, space complexity, and if the solution is optimal. This page is a conclusion only, I will compare the algorithms on the testing page with test data to prove.

Depth-First Search(DFS)

Quality: DFS explores deep paths before backtracking, making it memory-efficient but not always optimal for shortest path finding.
Test Performance: DFS often took longer and produced longer paths in complex mazes, indicating its inefficiency in environments with many dead ends. The path it discovers is usually not optimal.

Breadth-First Search (BFS)

Quality: BFS explores all nodes at a current depth before moving deeper, making it optimal for finding the shortest path in unweighted scenarios, especially in a scenario where all steps have the same cost as this one.
Test Performance: While reliable for shortest paths, its heavy memory usage makes it less suitable for large grids.

Greedy Best-First Search (GBFS)

Quality: GBFS uses a heuristic to prioritize closer nodes to the goal, aiming for faster solutions. It does not store the cost to get to the current node.
Test Performance: Effective in simpler scenarios by offering high speed and low space complexity but struggled with finding the most optimal route in more complex grids.

A* Search (A*)

Quality: Combines DFS and BFS benefits, using heuristics to guide search, which balances speed and accuracy.
Test Performance: Consistently found the shortest path efficiently across different environments, making it the best performer among the algorithms tested.

Iterative Deepening Depth-First Search (IDDFS)

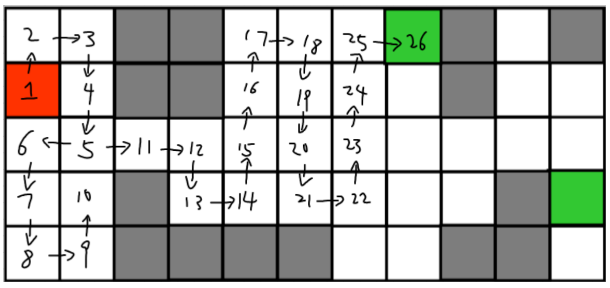
Quality: Blends DFS’s space efficiency with BFS's thoroughness by adding an increasingly limited depth to each search.
Test Performance: It has no impact on easy mazes but will be much more reliable in complex mazes due to the limited depth, but still does not always guarantee the shortest path.

Hill Climbing (HC)

Quality: Moves towards increasing goal proximity without storing other paths, making it memory-efficient but may get stuck at local maxima.
Performance: Quick in clear-path scenarios but unreliable in complex mazes.

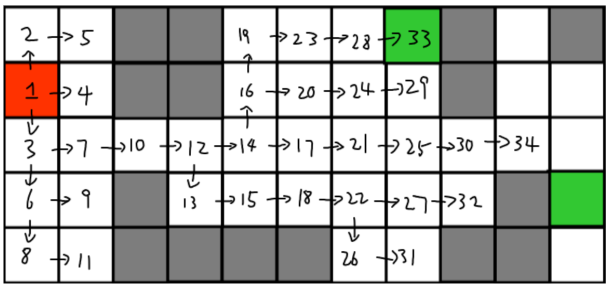
Implementation

Depth-First Search(DFS)



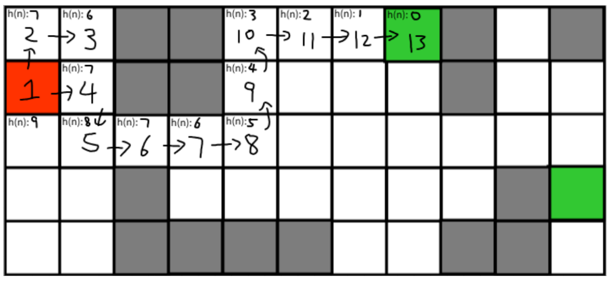
DFS is LIFO, therefore we will need a stack to store the nodes to expand. While the stack still has nodes, we should pop an item and get the info for it first, then the node will be checked if it is expanded already. If it is not, we expand it. Then, we do the goal check, ending the program if the current frontier is the goal. After that, I create a possible movement table using the position of the frontier node. Notice that we add TOP, then LEFT, then DOWN, then RIGHT. That is to ensure that the search method prioritizes the node based on the requirement. Since DFS is LIFO, when we use for loop to check the potential nodes, we should loop it in reverse so that the TOP node is added in the last. Then we start checking if the potential node is actually possible to reach if the node is expanded already, in the wall, or outside of the grid. If it is not in the above, we add it to our stack.

Breath-First Search(BFS)



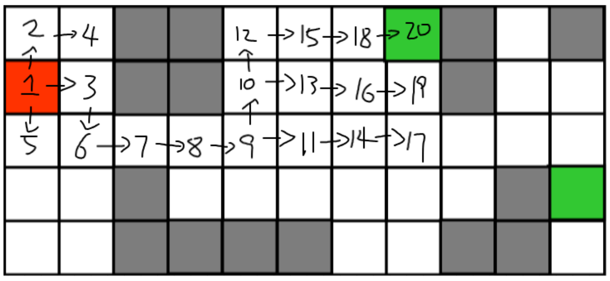
BFS is similar to DFS, only a few things have changed. First, the stack is changed to queue because BFS is FIFO, and we will need to be able to pop from the left to achieve this. If we pop from the left in the stack, that will take O(n) time complexity instead of O(1) in the queue because it has to pop every single element first and add it back in. Second, we loop the queue normally instead of looping it in reverse because BFS is FIFO. We have to add the TOP node first to ensure it got explored first. Finally, the node was expanded first before visiting because of how BFS works. For example, looking at the graph above, you can see even though a path is clear in 33, we still check the 34 nodes. The reason for this is that BFS will check every node in the same layer first before moving to the node. Since we don't check if the expanded node is the goal but instead the current node is the goal, we will have to visit the 34 nodes.

Greedy Best-First Search (GBFS)



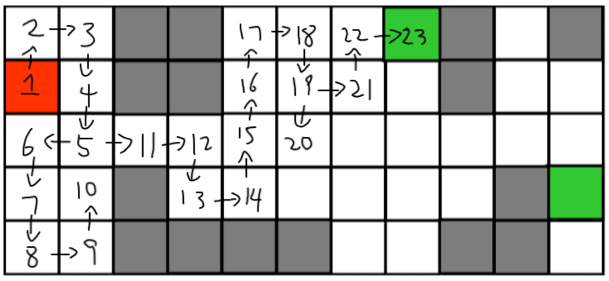
The design of GBFS is actually quite similar to BFS and DFS. However, since GBFS uses a heuristic function, we need to also store the heuristic cost of each node. On top of that, we should add a sorting method after we append the TOP, RIGHT, DOWN, LEFT nodes into the queue. So that we can prioritize visiting the lower heuristic cost node first rather than the higher cost node.

A* Search (A*)



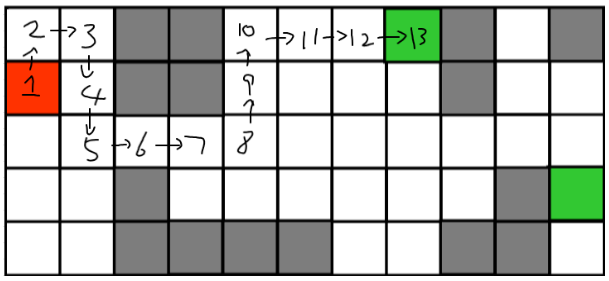
A* is basically GBFS with 1 difference. That is to use f(n) = h(n) + g(n) to choose the nodes instead of just h(n). g(n) in this case means the cost to get to the current node. In the code, we should add the f(n) cost into the queue instead. We should also have a dictionary to keep track of the actual cost to reach each node from the start node. The dictionary is updated whenever we discover a new node or a better path to get to that node. When we are sorting the queue, we should also sort it with the f(n) instead of the g(n).

Iterative Deepening Depth-First Search (IDDFS)



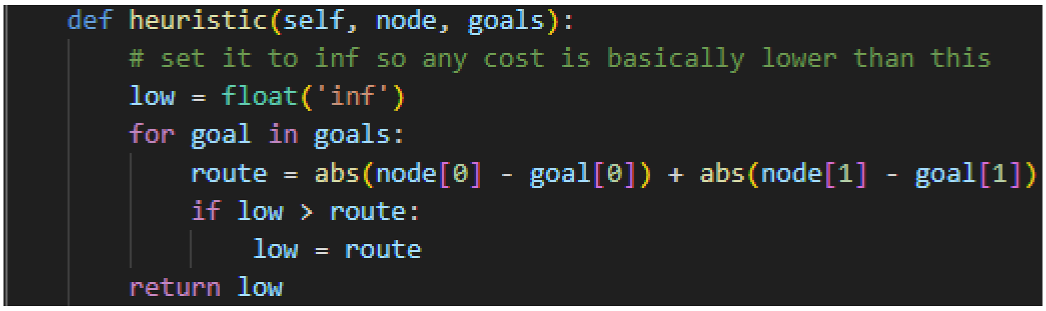
IDDFS is a DFS with a limited depth that gets updated every time that depth can not find a result. So we should add a loop that scales infinitely and limits the DFS with a depth, when the depth is hit, we instead focus on the other nodes first until the DFS updated with new depth. This presents some worst-case scenarios. Every time we do a DFS, we should clear the pre-existing expended nodes first.

Hill Climbing (HC)



The only difference between Hill Climbing and GBFS is that in Hill Climbing, you can't go back after jumping to a node. Therefore we shouldn't store a node in a queue. We should instead just replace the current node with the new node. In code, we can achieve this by storing a tuple for the position, 2 arrays for the path and direction we went and that's it. The other part will most likely be the same compared to GBFS.

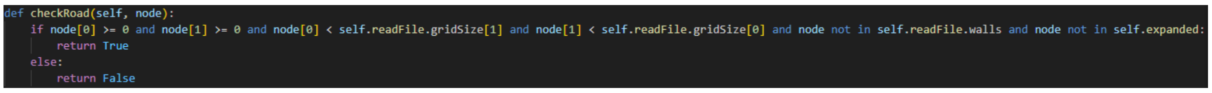
Heuristic Function



```python
    def heuristic(self, node, goals):
        # set it to inf so any cost is basically lower than this
        low = float('inf')
        for goal in goals:
            route = abs(node[0] - goal[0]) + abs(node[1] - goal[1])
            if low > route:
                low = route
        return low
```

The Heuristic Function is simple. We are only getting the absolute difference in the position between the node and each goal. After that, we should use the lowest cost as the heuristic cost to make sure that we don't overestimate the cost.

Check Road Function

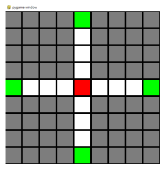
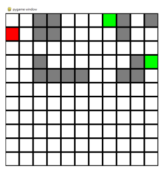
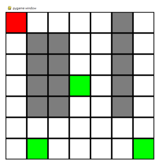
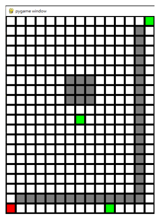
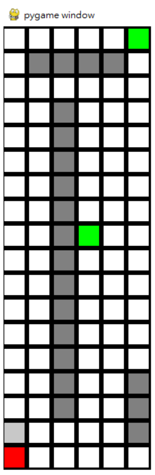


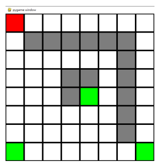
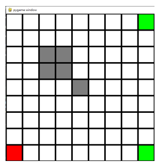
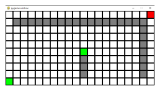
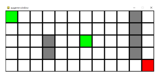
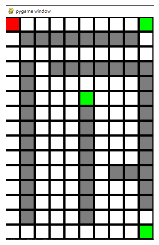
```python
def checkRoad(self, node):
    if node[0] >= 0 and node[1] >= 0 and node[0] < self.readFile.gridSize[1] and node[1] < self.readFile.gridSize[0] and node not in self.readFile.walls and node not in self.expanded:
        return True
    else:
        return False
```

Check road is used in all search methods. This function just checks if the node we are going to is available and returns the answer. If a node is outside of the grid, is a wall, or is a node that is already expanded. It is not a node that we should explore anymore.

Testing

After creating the search algorithms, we shouldn't only test it on the default data. We should also create some test cases to test its **LIMIT**, **LOGIC,** and **PERFORMANCE**. Therefore, I created 10 test cases for different scenarios to see if the algorithms are working as intended, and determine the best algorithms out of these 6.

Test Case	Lowest cost	DFS	BFS	GBFS	A*	IDDFS	HC
	4	nodes:5 cost:4	nodes:17 cost:4	nodes:5 cost:4	nodes:14 cost:4	nodes:5 cost:4	nodes:5 cost:4
	10	nodes:64 cost:50	nodes:67 cost:10	nodes:13 cost:10	nodes:20 cost:10	nodes:44 cost:14	nodes:13 cost:12
	6	nodes:8 cost:7	nodes:19 cost:6	nodes:8 cost:7	nodes:10 cost:6	nodes:14 cost:6	nodes:8 cost:7
	10	nodes:12 1 cost:120	nodes:67 cost:10	nodes:11 cost:10	nodes:11 cost:10	nodes:41 cost:10	nodes:11 cost:10
	12	nodes:23 cost:22	nodes:53 cost:12	nodes:32 cost:22	nodes:32 cost:12	nodes:47 cost:12	nodes:23 cost:22

	7	nodes:8 cost:7	nodes:27 cost:7	nodes:8 cost:7	nodes:8 cost:7	nodes:8 cost:7	nodes:8 cost:7
	8	nodes:67 cost:58	nodes:47 cost:8	nodes:9 cost:8	nodes:9 cost:8	nodes:54 cost:16	nodes:9 cost:8
	22	nodes:29 cost:28	nodes:105 cost:22	nodes:29 cost:28	nodes:61 cost:22	nodes:83 cost:24	nodes:29 cost:28
	7	nodes:20 cost:15	nodes:28 cost:7	nodes:10 cost:7	nodes:16 cost:7	nodes:17 cost:7	nodes:6 cost:No goal is reachable
	9	nodes:39 cost:37	nodes:43 cost:9	nodes:10 cost:9	nodes:10 cost:9	nodes:37 cost:9	nodes:10 cost:9

Depth-First Search(DFS)

Depth-first search exceeds in the simple maze. Being effective and offering good space complexity. It is a better choice than BFS in that scenario. However, when we start scaling the maze and adding complexity, the disadvantage of DFS will start showing up. The time complexity scales up exponentially. On top of that, the path that DFS discovers in those scenarios is extremely inefficient. Therefore, I think DFS is only effective on simple tasks, even in situations where space complexity is important, there are still better choices to go for.

Breadth-First Search (BFS)

Breadth-First Search has bad space complexity, while not being very time efficient. However, it is guaranteed to find the optimal path if all step has the same cost. This makes it somewhat useful even in a more complex scenario. Looking at the tests, BFS usually wasted the most amount of time finding a path. Unlike DFS, BFS is actually worse in simple cases compared to other algorithms. But it will always have the lowest cost no matter what, so it is suitable for most scenarios where the only thing that matters is the optimal path.

Greedy Best-First Search (GBFS)

Greedy Best-First Search is very fast. The space complexity is in the middle. Although it is not guaranteed to find the optimal path. With the help of the heuristic function, GBFS can still get the optimal path or a path that is close to the optimal path most of the time. I originally thought A* is much better than GBFS, but I can see a situation where GBFS is preferred over A*, like a situation where the optimal path is not as important as the time complexity.

A* Search (A*)

A* is a good all-rounder. It has a relatively fast speed compared to other algorithms, especially when the heuristic function is good. The space complexity is not excellent usually, but still better than BFS. And A* guaranteed optimal path. It is the best choice in these 6 algorithms in my opinion if you want an algorithm that can scale well and offers an optimal path.

Iterative Deepening Depth-First Search (IDDFS)

IDDFS is very interesting, and I'm glad I picked this as one of my custom algorithms. It is a DFS at its core but also has the identity of a BFS. By adding a depth limit in the search, it helps with the scalability of DFS. Whenever the depth is hit, it forces the node to stop expanding and prioritize another path first to prevent the worst case. Looking at the test, it has the best of both worlds in my opinion. Even though it is not guaranteed an optimal path, it has better time complexity than DFS on average, better space complexity than BFS, and is better at finding lower costs than DFS.

Hill Climbing (HC)

Looking at the test data, I don't think Hill Climbing should be used in most cases other than giant data as it takes a lot of space to store all the potential. It uses a heuristic function, and although it is very fast, it doesn't guarantee an optimal path. Sometimes, the performance is worse than some of the uninformed algorithms. The only good side of it is the O(1) space complexity at ALL TIME. It doesn't store the potential path to take in a queue. It only goes to the nearest node based on the heuristic costs. Furthermore, in an extreme case like test 9, it goes into a dead end since the node around it is visited. It will just report no path found even though there is clearly a path to take. I think unless the situation demands an O(1) space complexity algorithm, this shouldn't be used as a main way of finding a solution to a problem.

Conclusion

Based on the test, I think that If a heuristic function is available, then an informed method should preferred because of its performance. For different search methods, it came with its own pros and cons. We have to decide which one is more important. Is it speed? Space efficient? Optimal path? Furthermore, based on the problems, the method we should use might differ. If we don't have a heuristic function today and there is a problem that needs an optimal solution, I will choose BFS. But in an easy problem that doesn't need an optimal solution but needs less space usage, I might prefer DFS. The "Best method" is pretty subjective, but I would say in general consideration, A* will be the best because it's an all-rounder.

Features/Bugs/Missing

Features:
All 6 search methods are done
GUI for display
Good scalability with a singleton pattern
User input for the algorithm name is not case-sensitive
Raise ValueError when the algorithm name is incorrect
Multiple class scripts for modular programming
Comments in every function for easier understanding

Bugs:
Even though the outline of each block changes to 1 from 3 when the block count reaches 2500. If the grid size is insanely big (500 * 500), the display of the GUI might be very hard to see.

Missing:
A user interface that can interact with a mouse click might be useful for the GUI.
Potential Upgrade
A better heuristic function that takes obstacles into account as well.

Research

For the research initiatives, I've picked the GUI display as my research. Due to the knowledge I have in Unity, Ruby Gosu, and SplashKit. For how to understand the GUI, I've put the instructions on the instructions page. So I would instead explain the code here.

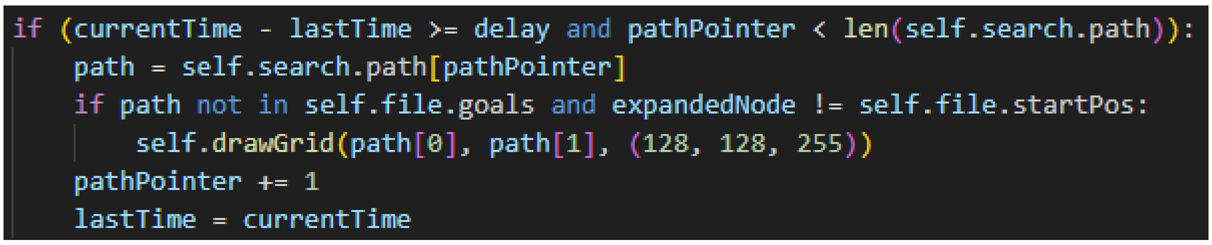
When I was building the program, the file reader actually consumed a lot of my time. Because I want the file reader to process every single data that other scripts need, walls, starting POS, etc. However, it pays off massively because the GUI becomes extremely easy to implement. All I have to do is create a draw grid function, and use that function to draw the empty grid with 2 for loop for the x and y axis. And then just paint the individual block with the information I get from reading the data file and algorithms. If it's a wall, paint the block gray, if it's a goal, paint it green, etc.(for what all the color means, refer back to page 4 of the GUI instructions.)

The hardest part of this is to create a delay for each block since we need to display how the robot finds its path. Luckily, I have learned how to create a bullet delay with the space shooter game I've created with Unity in my free time[1]. I can just borrow the code there (C#) and adopt it in this program. It is not as easy as I expected, pygame does not separate the fps and the action. For every frame in pygame, we run a loop. And since pygame will try to run as fast as possible. That means how fast or slow the program will go is totally depending on your computer power. In order for the delay to be the same on every single computer. We can't have code that does something like this:

```
if counter == 10
    do something
    reset counter
else
    counter+=1
```

We have to use the time offered by pygame. So that we delay the display based on the time, not how many frames it runs.

Basically, I will have 3 variables, current time, last time, and a delay. We will do an if-check on every loop we done. and if the difference between the last time we've displayed the block and the current time is bigger than the delay. We display 1 block and set the current time to be the new last time.



```
if (currentTime - lastTime >= delay and pathPointer < len(self.search.path)):
    path = self.search.path[pathPointer]
    if path not in self.file.goals and expandedNode != self.file.startPos:
        self.drawGrid(path[0], path[1], (128, 128, 255))
    pathPointer += 1
    lastTime = currentTime
```

[1] https://sharemygame.com/@aiken2001/laser-defender

Conclusion

When addressing the Robot Navigation Problem, the choice of search algorithm can greatly influence both the effectiveness and efficiency of finding a solution. Given the problem's inherent complexities—such as varying grid sizes, the presence of obstacles, and multiple goals—the best type of search algorithm often depends on the specific requirements of the task, such as the need for the shortest path or the fastest search time.

Recommended Search Algorithm

For general purposes, A Search (A-star)* stands out as the most effective approach for the Robot Navigation Problem. A* efficiently balances between the breadth-first and depth-first extremes, leveraging heuristics to estimate the cost from the current node to the goal. This makes A* particularly powerful for finding the shortest path in a varied terrain grid, as it can dynamically prioritize paths that seemingly lead closer to the goal, thus reducing the total computation time.

Improving Performance

Optimized Heuristic Functions

The performance of A* largely depends on the quality of the heuristic function. Using a more accurate heuristic, such as taking the obstacles into account when providing the cost can greatly increase the performance.

Bi-directional Search

While researching for the custom search method. I also came across the Bi-directional Search method which can be applied in all of the search methods above. Implementing a bi-directional version of A* could further improve performance. This technique involves simultaneously running two instances of the algorithm—one from the start node and one from the goal node—and stopping when they meet. This method can potentially halve the search space, leading to faster solutions.

Memory Optimization

By researching and testing the IDDFS search method. It makes me realize that for memory-limited scenarios, using a less memory-intensive algorithm might be necessary. We can apply Iterative Deepening to the A*, increasing the space efficiency.

Final Thoughts

As I mentioned earlier on the testing page. While A* provides a robust framework for navigating complex environments, the no-one-size-fits-all solution to the Robot Navigation Problem underscores the importance of context in algorithm selection.

Acknowledgments/Resources

To complete this assignment, I go through a lot of resources online. The lectures were too hard for me to understand so I would go through the topic of the lectures first before watching the lectures. So here are some acknowledgments on it.
Michael Sambol on Youtube:
https://www.youtube.com/@MichaelSambol
This guy helps me understand a lot of fundamental concepts in the unit. He assisted me in designing the BFS and DFS of my project with his algorithm in 4-minute videos.
Neetcode:
https://neetcode.io/
This is originally just a website I watched sometimes so I can learn to do some leetcode. But the lectures there did go through different search algorithms and linked some leetcode about the topics so I can get some practice on it.
Geeksforgeeks:
https://www.geeksforgeeks.org/iterative-deepening-searchids-iterative-deepening-depth-first-searchiddfs/
https://www.geeksforgeeks.org/introduction-hill-climbing-artificial-intelligence/
This is the website that I use to understand the custom search algorithm that I chose

Reference

GeeksforGeeks. (2023, February 20). *Iterative deepening search(ids) or iterative deepening depth first search(iddfs)*. GeeksforGeeks. https://www.geeksforgeeks.org/iterative-deepening-searchids-iterative-deepening-depth-first-searchiddfs/

GeeksforGeeks. (2023, April 20). *Introduction to hill climbing: Artificial intelligence*. https://www.geeksforgeeks.org/introduction-hill-climbing-artificial-intelligence/