COS30019

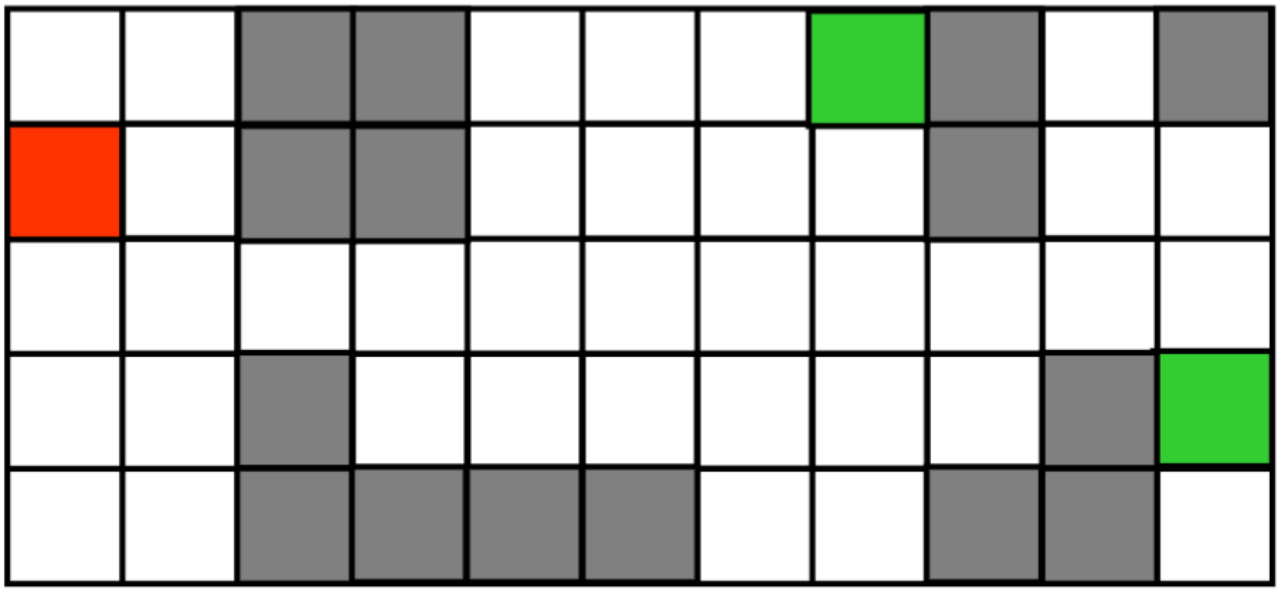
INTRODUCTION TO AI

SEMESTER: SPRING 2024

ASSIGMENT 1 – ROBOT NAVIGATION

The Robot Navigation Problem

In the lectures you have seen the Robot Navigation problem: The environment is an NxM grid (where N > 1 and M > 1) with a number of walls occupying some cells (marked as grey cells). The robot is initially located in one of the empty cells (marked as a red cell) and required to find a path to **visit one of the designated cells of the grid** (marked as green cells). For instance, the following is one possible environment:



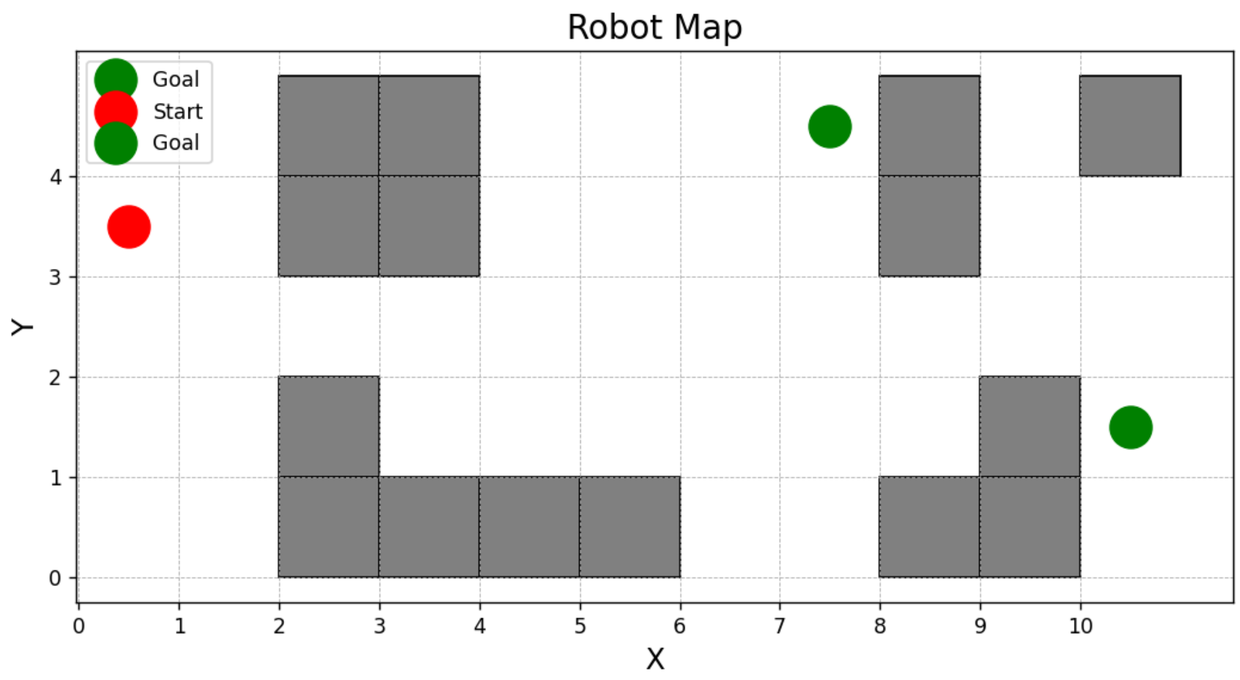
NGUYEN TUAN DUC

104189440

INTRODUCTION

In this assignment, my main task is to implement four pathfinding algorithms that I have learned in the course COS30019: Introduction to AI, taught by two instructors, Mrs. Nguyen Thi Kim Dung for lectures and Mr. Nguyen The Phuc for practical sessions. Additionally, I am required to research and discover two additional algorithms to fulfill the assignment's requirements.

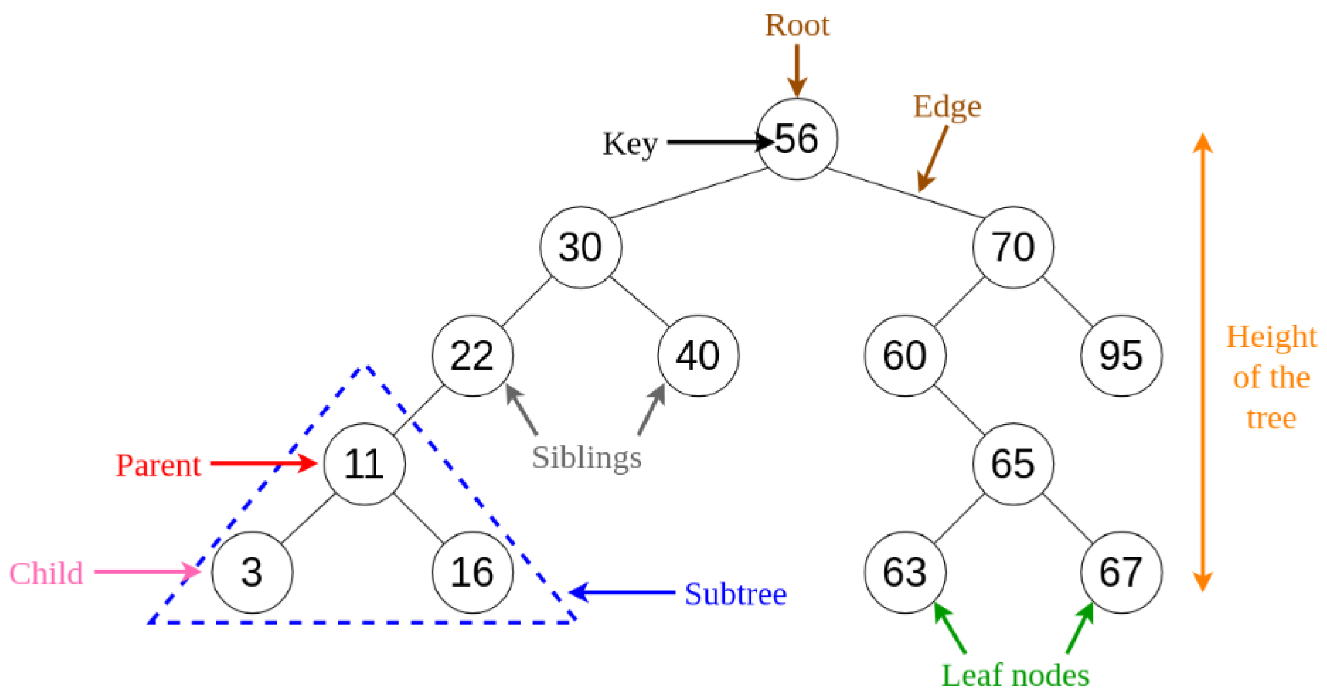
In this report, I will present the definitions and operational mechanisms of these algorithms, along with how I apply them to real-world projects.



The Robot Navigation challenge presents a classic maze problem, where various search algorithms are tested on search trees. Typically, there's a predefined starting and ending point, and the efficiency of different search algorithms is evaluated based on their ability to find a solution compared to each other.

In this task, we were provided with a starting point, destination, and several obstacles. The goal was to translate this information into a navigable environment for search algorithms. Subsequently, we had to implement four different search algorithms, with the option to devise two custom algorithms. These included Depth-First Search, Breadth-First Search, Greedy Best-First Search, and A* Search. The desired outcome was to select the test file name, method used, nodes explored, and the path taken.

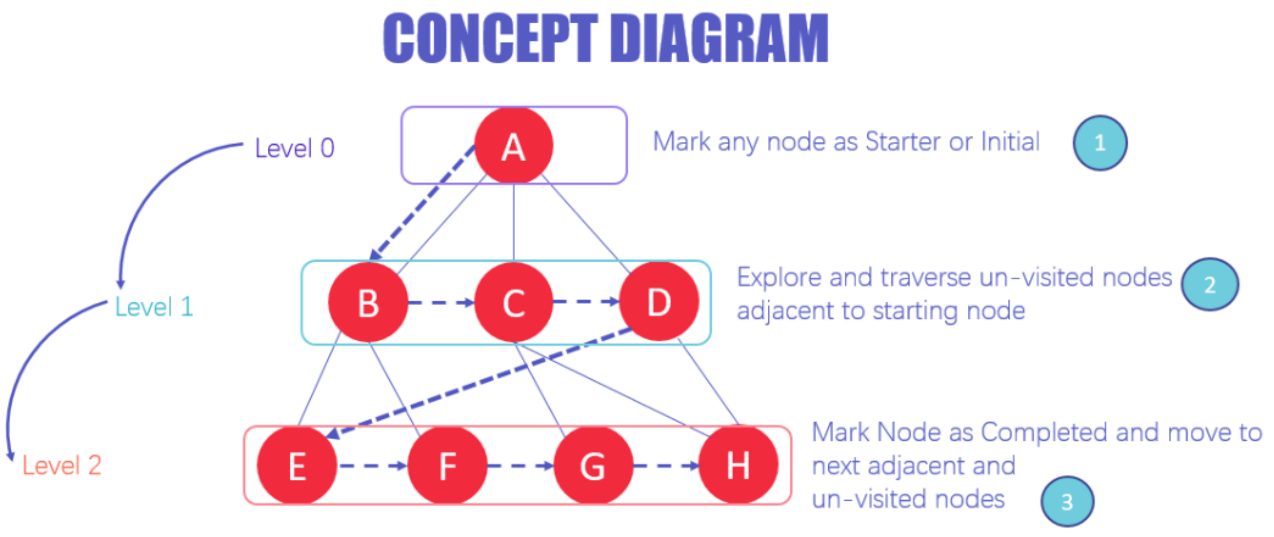
To tackle this challenge, a solid grasp of graph and tree principles was essential. The key concept is that a node may have multiple descendant nodes, and expanding all potential nodes results in a tree graph representing the entire dataset.

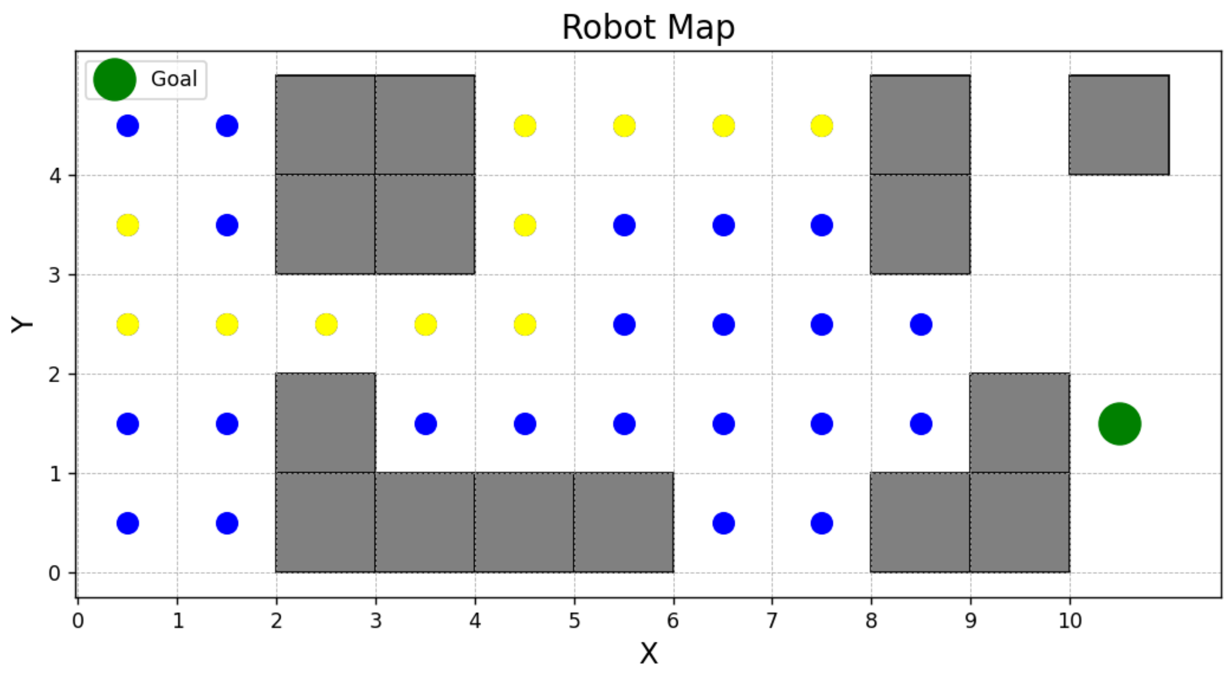


1. Breadth-First Search (BFS)

Breadth-First Search (BFS) algorithm takes a systematic and methodical approach to exploring a graph or a tree. Unlike Depth-First Search (DFS), which delves deep into one path before exploring other options, BFS adopts a broader perspective. It can be likened to systematically exploring a neighborhood, where every house on one street is thoroughly inspected before moving on to the next street. This method ensures that BFS thoroughly examines all nearby options before venturing further out into uncharted territory.

The strength of BFS lies in its ability to efficiently find solutions that are likely to be nearby. For example, when searching for the quickest route in a simple map where all paths take the same time, BFS excels due to its organized and exhaustive exploration style. Additionally, BFS is widely used across various problem-solving scenarios due to its versatility and effectiveness in systematically traversing graphs and trees.

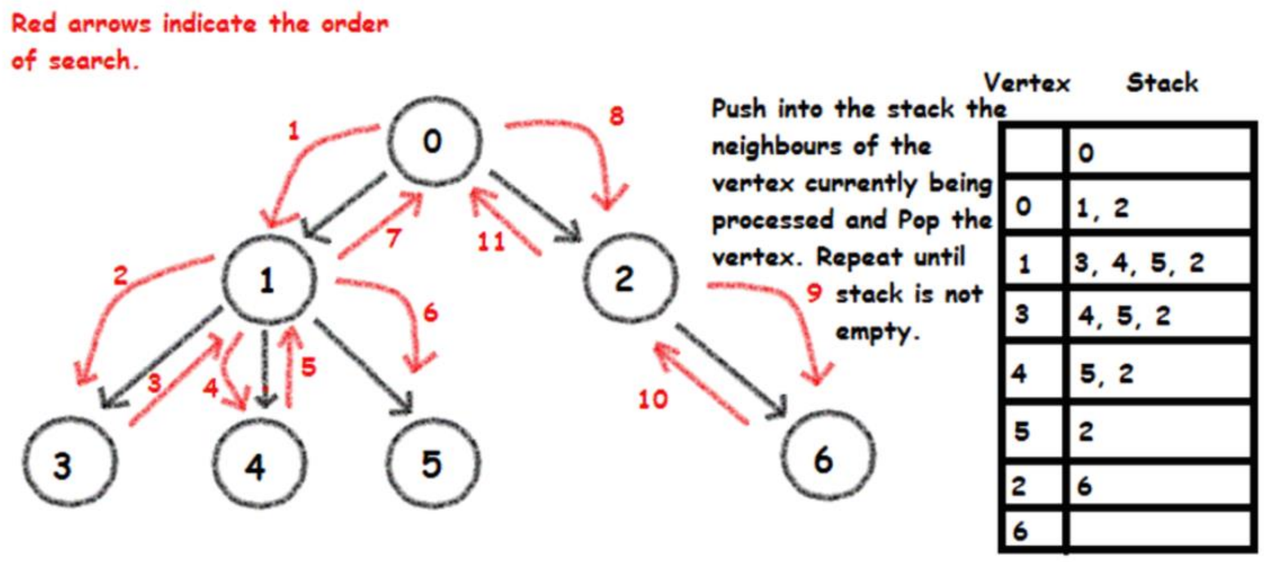

CONCEPT DIAGRAM
Level 0
Level 1
Level 2
A
B
C
D
E
F
G
H
Mark any node as Starter or Initial
1
Explore and traverse un-visited nodes adjacent to starting node
2
Mark Node as Completed and move to next adjacent and un-visited nodes
3


Robot Map
Goal
X
Y
0
1
2
3
4
5
6
7
8
9
10

2. DFS (Depth-First Search)

DFS (Depth-First Search) stands out for its distinctive search strategy. It plunges as far as possible along a single path within a maze-like structure before retracing its steps and exploring alternate paths. It achieves this by visiting a point and systematically examining each of its unvisited neighbors until reaching the target or exhausting all options.

While this method is well-suited for tackling complex problems, caution must be exercised to avoid getting ensnared in infinite loops. DFS operates akin to a determined adventurer navigating through a labyrinthine maze, venturing as deeply as feasible before branching out to explore new avenues. This careful balance ensures the traversal remains both productive and efficient.


Red arrows indicate the order of search.
Push into the stack the neighbours of the vertex currently being processed and Pop the vertex. Repeat until stack is not empty.
0
1
2
3
4
5
6
1
2
3
4
5
6
7
8
9
10
11

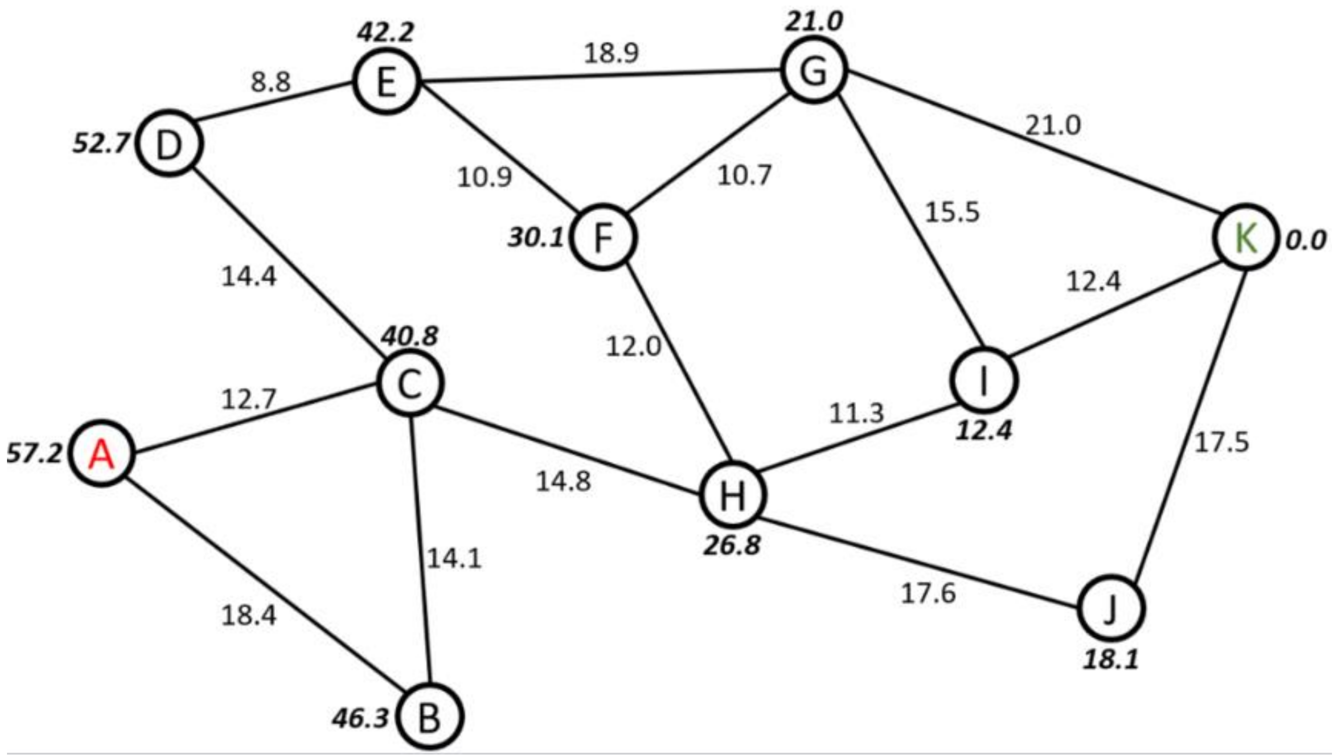
Vertex	Stack
	0
0	1, 2
1	3, 4, 5, 2
3	4, 5, 2
4	5, 2
5	2
2	6
6	

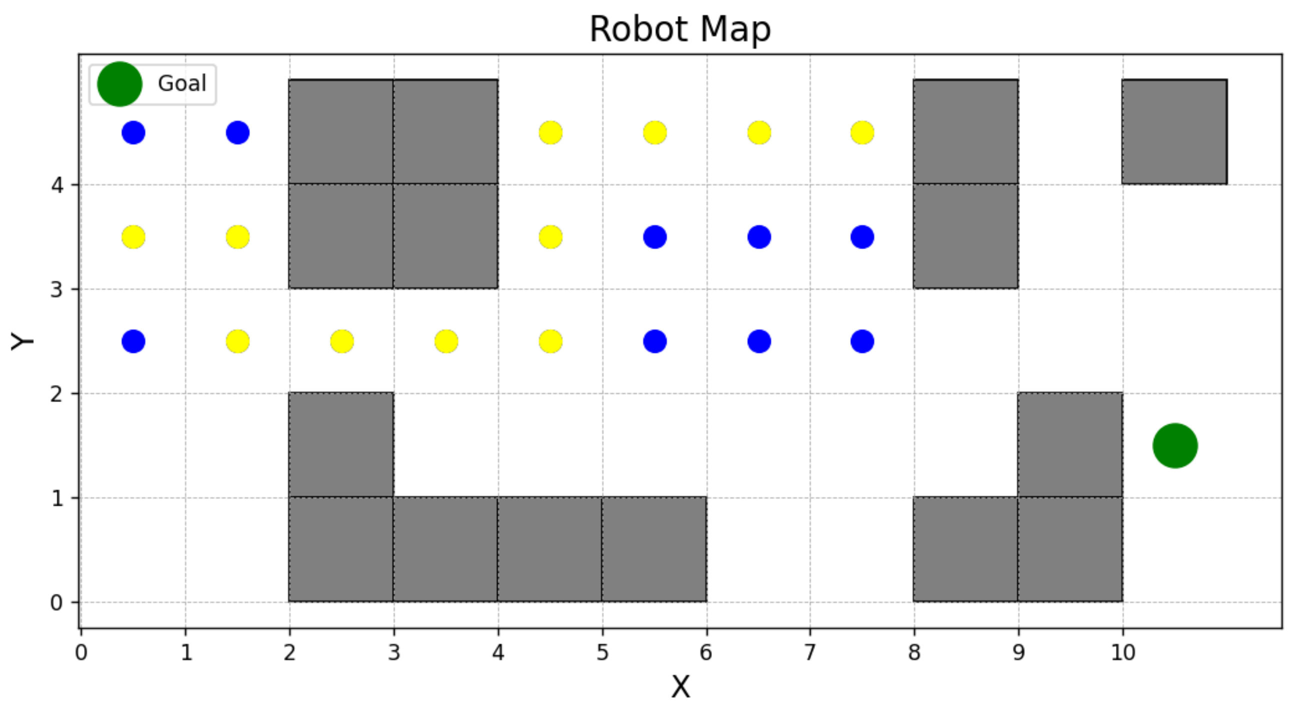
Depth First Search

3. A_STAR(A*)

A* Search is a remarkable algorithm that combines the strengths of two different search strategies. Similar to Uniform Cost Search, it considers the distance traveled thus far, while also incorporating elements of Greedy Best-First Search by evaluating proximity to the goal.

Utilizing a heuristic, A* Search estimates the potential of each option and selects the most promising one overall. This intelligent approach prevents the algorithm from becoming trapped in unfavorable paths, ensuring it efficiently identifies the optimal route. Analogous to possessing a map and compass in a maze, A* Search provides both a sense of past progress and future direction. Consequently, it emerges as a standout performer in numerous scenarios requiring pathfinding or solution discovery, particularly in maze-like environments and other network-related problems.

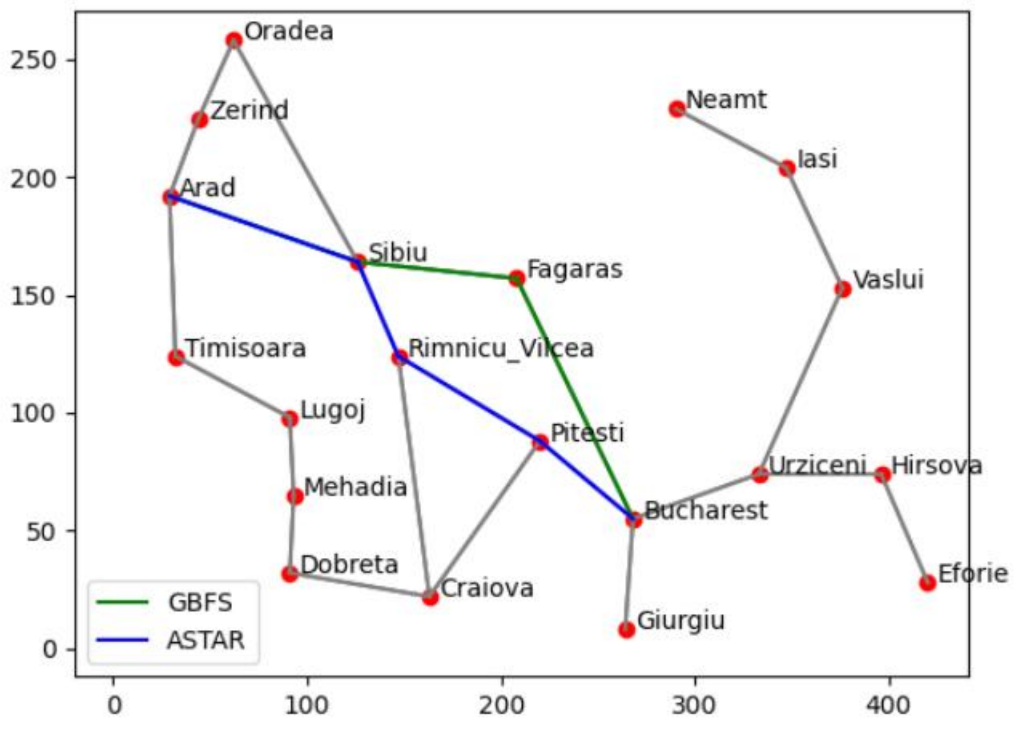


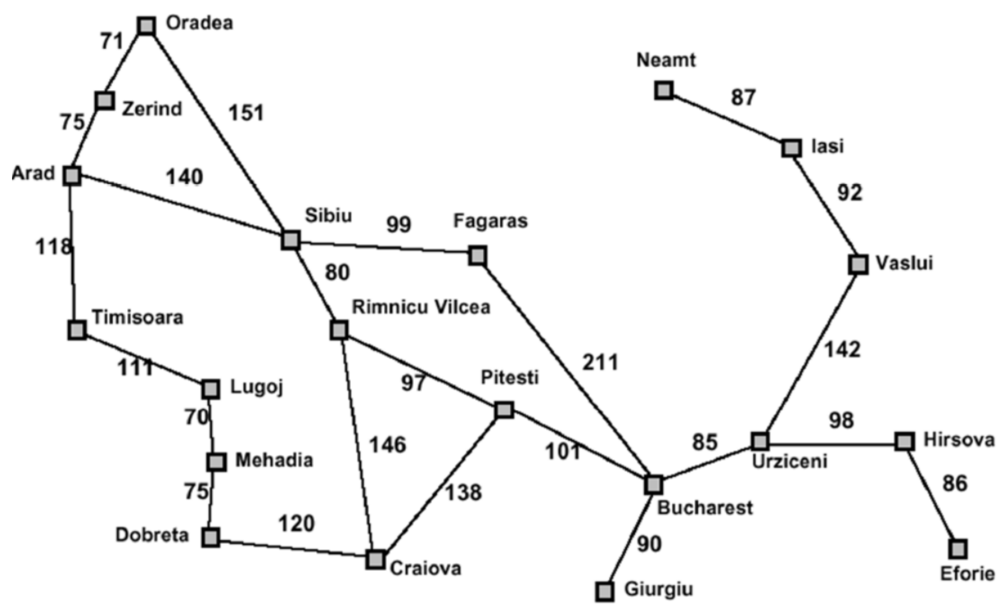


4. Greedy Best-First Search (GBFS)

In contrast to traditional search methods, Greedy Best-First Search prioritizes exploration based on an educated guess of how close each option is to the goal. It always chooses the path that currently appears most promising for reaching the destination quickly. This approach involves making locally optimal choices at each step, focusing on minimizing the estimated cost to the goal, rather than considering the entire problem structure.

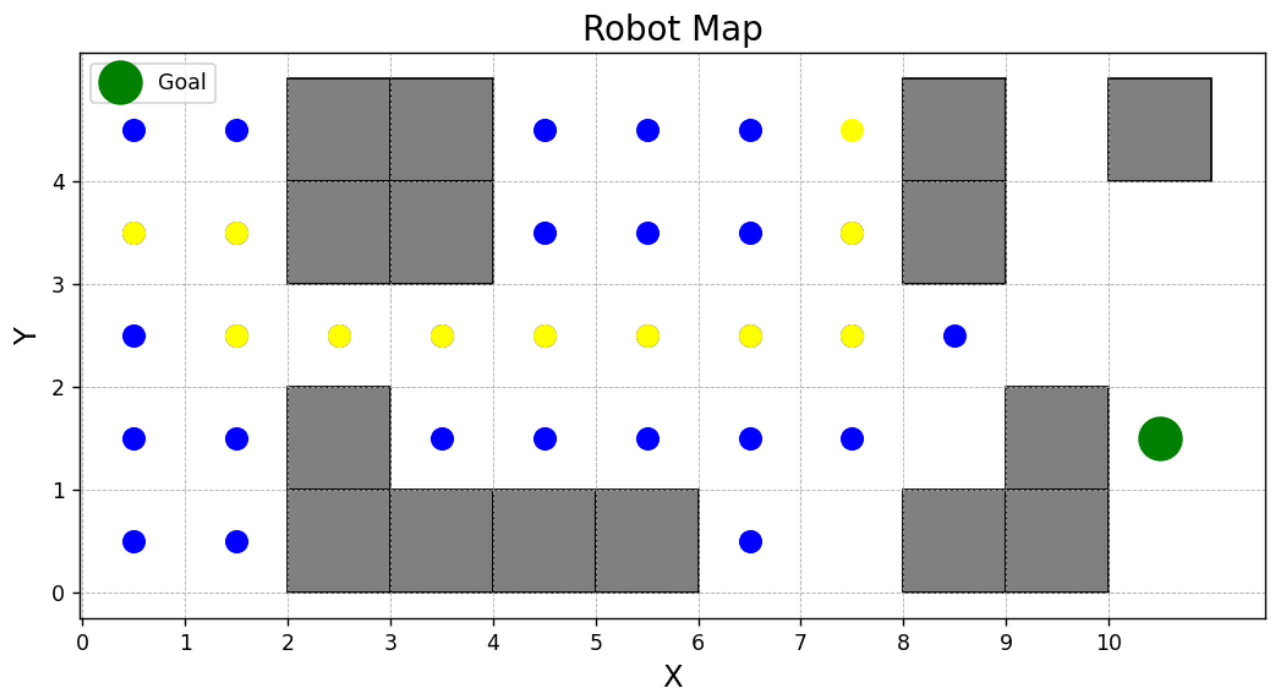
While Greedy Best-First Search can be efficient in specific situations, its reliance on estimations can lead to suboptimal solutions if the estimations are inaccurate or fail to capture the complete problem complexity. Comparatively, A* Search, another popular algorithm, combines the strengths of both Greedy Best-First Search and Uniform Cost Search. A* Search incorporates a heuristic to estimate the cost of reaching the goal from each node, allowing it to make informed decisions about which paths to explore. This enables A* Search to strike a balance between efficiency and optimality, often outperforming Greedy Best-First Search in finding the optimal solution while still maintaining computational efficiency.





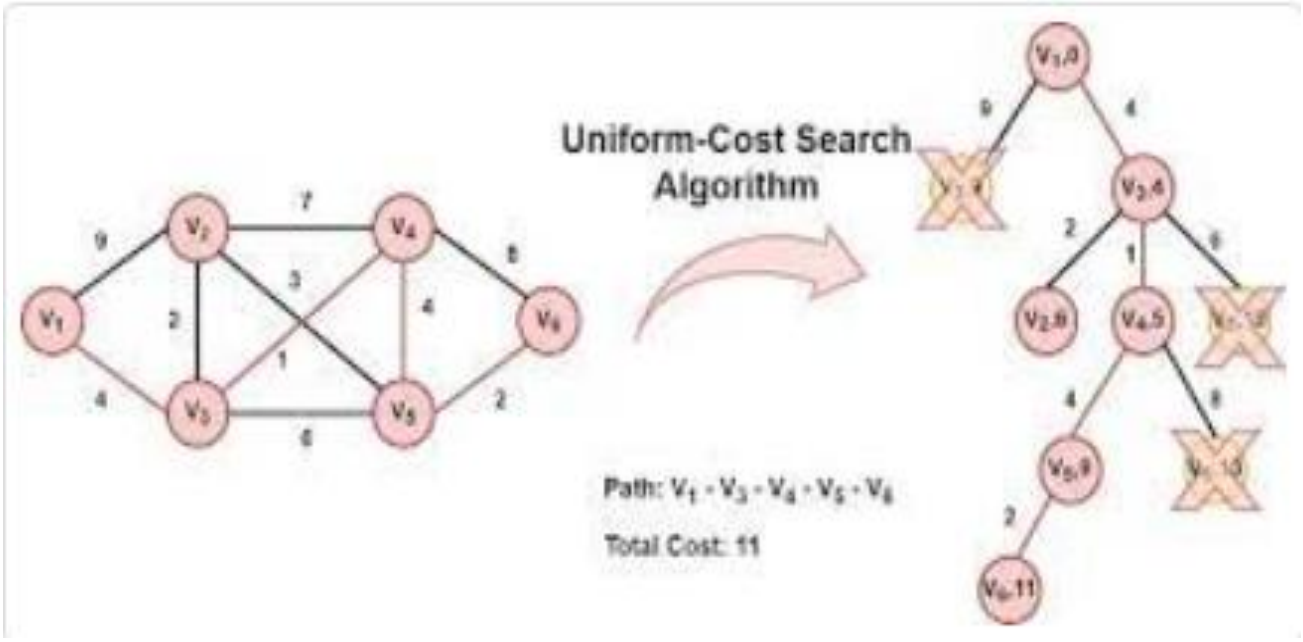
Straight−line distance to Bucharest

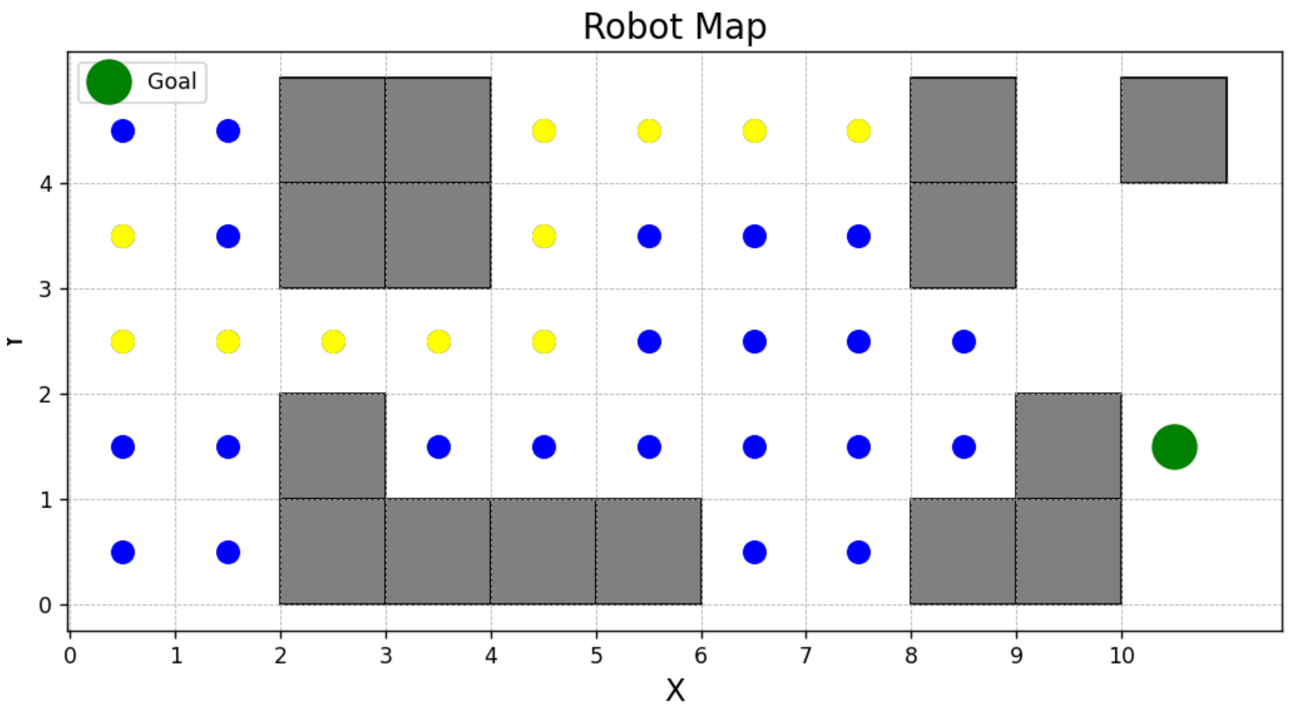
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



5. Uniform Cost Search (UCS)

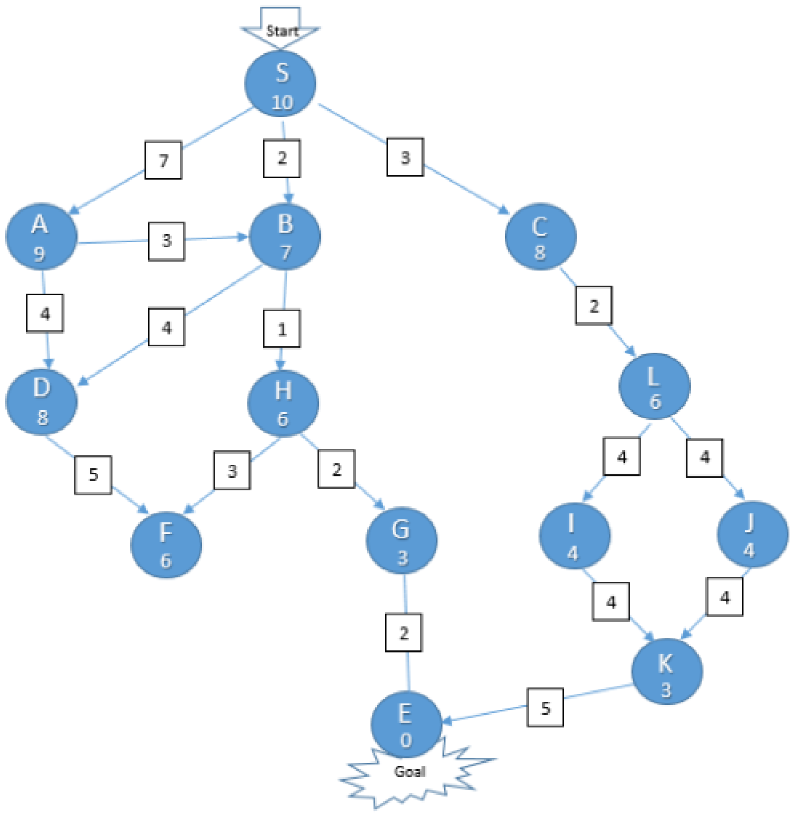
Uniform Cost Search (UCS) is a systematic algorithm that explores paths in a manner that prioritizes those with the lowest cumulative cost. Unlike Greedy Best-First Search, which focuses solely on proximity to the goal, UCS considers the actual cost incurred at each step. It meticulously evaluates all possible paths, gradually expanding outward from the starting point while always selecting the path with the lowest accumulated cost. This method ensures that UCS systematically explores the search space, guaranteeing that it finds the optimal solution with the lowest total cost. However, this exhaustive exploration can be computationally expensive, particularly in scenarios with large search spaces or numerous potential paths.

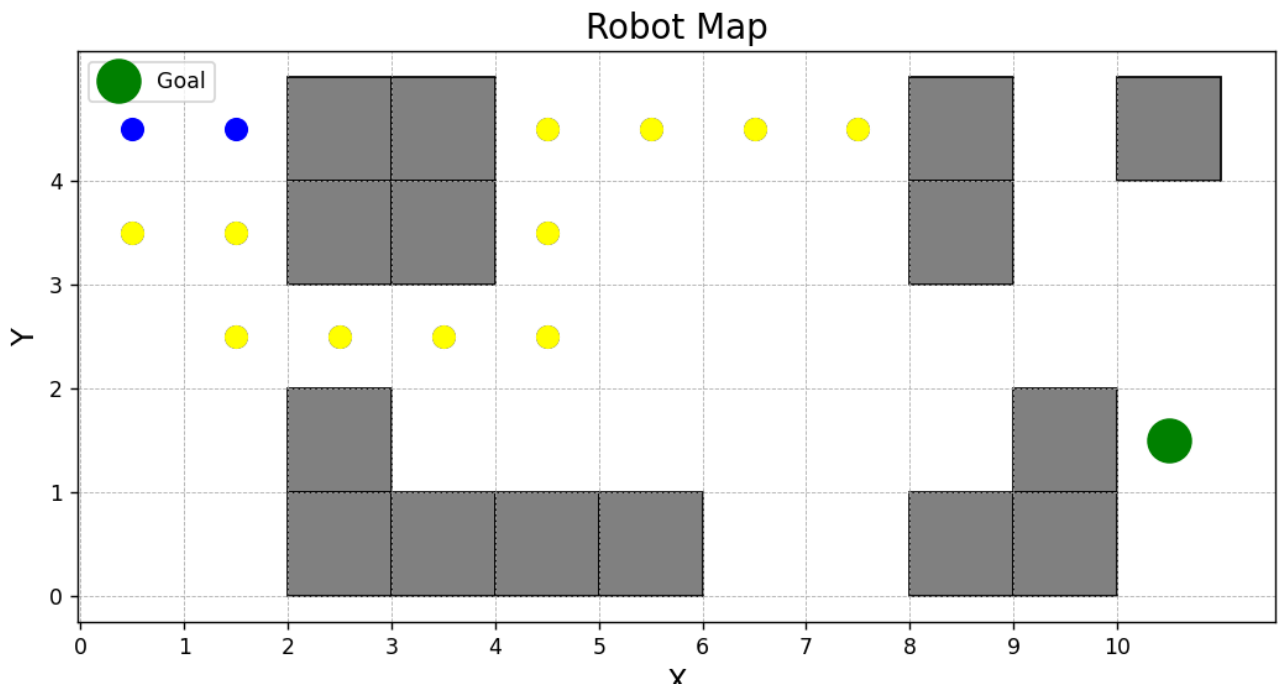




6. Best-First Search

Best-First Search, on the other hand, operates on a heuristic-based approach, aiming to identify the most promising path to the goal at each step. Unlike UCS, which evaluates paths based on their cumulative cost, Best-First Search makes decisions solely based on the estimated distance to the goal. It prioritizes exploration of paths that seem to be closer to the goal, making it highly efficient in situations where proximity to the goal is crucial. However, this approach does not guarantee finding the optimal solution, as it may overlook potentially better paths in pursuit of immediate gains. Best-First Search is particularly useful in scenarios where computational resources are limited, or where finding a reasonably good solution quickly is more important than finding the absolute best solution.


Start
S 10
A 9
B 7
C 8
D 8
H 6
L 6
F 6
G 3
I 4
J 4
K 3
E 0
Goal
7
2
3
3
4
4
1
2
5
3
2
4
4
2
4
4
5



7: IMPLEMENTATION DETAILS

In this chapter, we delve into the specifics of the application's implementation, guided by the provided code snippets. The application showcases a robust framework for map reading, visualization, and pathfinding utilizing various algorithms. Below, we detail each feature based on the supplied code.

4.1 Map Reading and Verification

The application begins with the create_grid_map function from create_and_draw_map.py, which reads and verifies the map's data from the input file. This function meticulously parses the text file to determine the grid's dimensions, starting point, goal positions, and obstacle placements. It then constructs a grid map represented as a 2D list, marking the respective positions for the robot, goals, and obstacles.

Error handling is integrated to ensure the data format's validity, especially for obstacle representations, ensuring robustness against incorrect file formats or data inconsistencies. This step sets the groundwork for the pathfinding process by establishing a clear, error-free representation of the map.

4.2 Algorithm Selection and Implementation

The main program prompts users to select a search algorithm from options such as A*, BFS, DFS, Greedy Best First Search, Uniform Cost Search, and Best First Search. Each algorithm is meticulously implemented to adhere to its standard operational principles:

A Search (asm1_a_star_search)**: Implements the A algorithm using a priority queue, evaluating nodes based on the sum of actual cost from the start and the estimated cost to the goal (heuristic). The method ensures optimal pathfinding by prioritizing nodes with lower total costs.

Breadth-First Search (asm1_bfs): Utilizes a queue to explore the grid level by level, ensuring that the search radiates evenly around the start position. This algorithm is especially effective in finding the shortest path on unweighted grids.

Depth-First Search (asm1_dfs): Employs a stack (implicit with recursion in this case) to dive deep into each path before backtracking, useful in scenarios where the full depth of the map needs exploration.

Greedy Best First Search (asm1_gdf_search): Guides the search toward the goal using only a heuristic function, prioritizing nodes that seem closer to the goal, disregarding the actual cost so far.

Uniform Cost Search (ucs_search): Expands the least cost node, ensuring that when the goal is found, the path is guaranteed to be the cheapest among all possible paths.

Best First Search (best_first_search): A variant of the breadth-first search that uses a priority queue to improve efficiency by guiding the search based on heuristic values.

The application facilitates a comprehensive approach to pathfinding by supporting multiple algorithms, allowing for comparative analysis and educational exploration.

4.3 Visualization and Path Reconstruction

Post-algorithm execution, the application visualizes the final grid map with the found path and visited nodes using matplotlib. The draw_map function illustrates obstacles, the start and end points, visited nodes, and the final path. This feature not only aids in understanding the algorithm's functionality but also provides immediate visual feedback on its effectiveness and efficiency.

Furthermore, path reconstruction is an integral part of the A* and Greedy Best First Search implementations, where the final path is traced back from the goal to the start position, ensuring that users can easily interpret the route taken by the algorithm.

4.4 User Interface and Error Handling

The user interface is straightforward, requesting the file path and the choice of algorithm through console inputs, ensuring ease of use. Additionally, robust error handling mechanisms, such as file existence checks and validation of algorithm names, enhance user experience by preventing common errors and guiding the user through the correct steps.

In summary, the implementation section meticulously outlines the development of a comprehensive pathfinding application. Each feature is designed with precision, from map reading and algorithm implementation to visualization and user interaction, ensuring a robust, user-friendly tool that effectively demonstrates various pathfinding strategies.

8: FEATURES/BUGS/MISSING

In this chapter, we evaluate the developed application, emphasizing features, identifying potential bugs, and discussing missing elements or improvements. The evaluation is based on the functionality reflected in the provided code and the designed algorithms.

8.1 Features

Node Position Display and Interpretation: The application is designed to display the x and y values of nodes within the found path. This facilitates a clear understanding of the path's trajectory. Post-pathfinding, the application interprets these coordinates into directional moves (up, down, left, right), providing an intuitive representation of the path for users. This feature enhances the user's comprehension of the algorithm's output, making the path's progression easily understandable.

Initial and Final Map Display: The code allows for the visualization of the initial map setup and the final result after pathfinding. Using matplotlib, the initial obstacles, start, and goal positions are displayed, and upon completion of the search algorithm, the final path and visited nodes are highlighted. This feature is crucial for a comprehensive understanding of the pathfinding process and for verifying the correctness of the algorithm's output.

8.2 Bugs and Potential Issues

While the code is well-structured and covers the fundamentals of pathfinding, there might be potential bugs or issues not immediately evident without extensive testing. For example:

Error Handling: While initial error checks are in place (such as verifying file existence and the validity of selected algorithms), additional error handling may be required to cover edge cases, such as incorrect file formats or invalid grid configurations.

Performance and Optimization: Depending on the size of the grid and the complexity of the obstacles, the application's performance can vary. Optimization techniques might be required to handle larger maps or more complex scenarios efficiently.

8.3 Missing Elements or Improvements

Real-Time Algorithm Visualization: Currently, the application does not provide a real-time view of the algorithm in action. Integrating a step-by-step visualization feature, where users can observe how the algorithm explores the grid and expands the search frontier, would significantly enhance the educational value and user engagement.

Path Optimization Feedback: The application successfully identifies a path but does not provide insights into its optimality or compare it against other potential paths. Introducing functionality to compare different algorithms' paths, time efficiency, and node exploration could provide valuable insights into each algorithm's effectiveness.

Interactive Map Configuration: Enhancing the application to allow users to dynamically create and modify maps within the interface, including placing obstacles, start, and goal positions, would greatly improve usability and experimentation.

Additional Pathfinding Algorithms: Including more diverse algorithms, such as Dijkstra's or Jump Point Search, would broaden the application's scope and educational value.

Comprehensive Documentation and Help Section: Providing users with detailed documentation on how to use the application and an explanation of each algorithm's mechanics and applications could make the tool more accessible and educational.

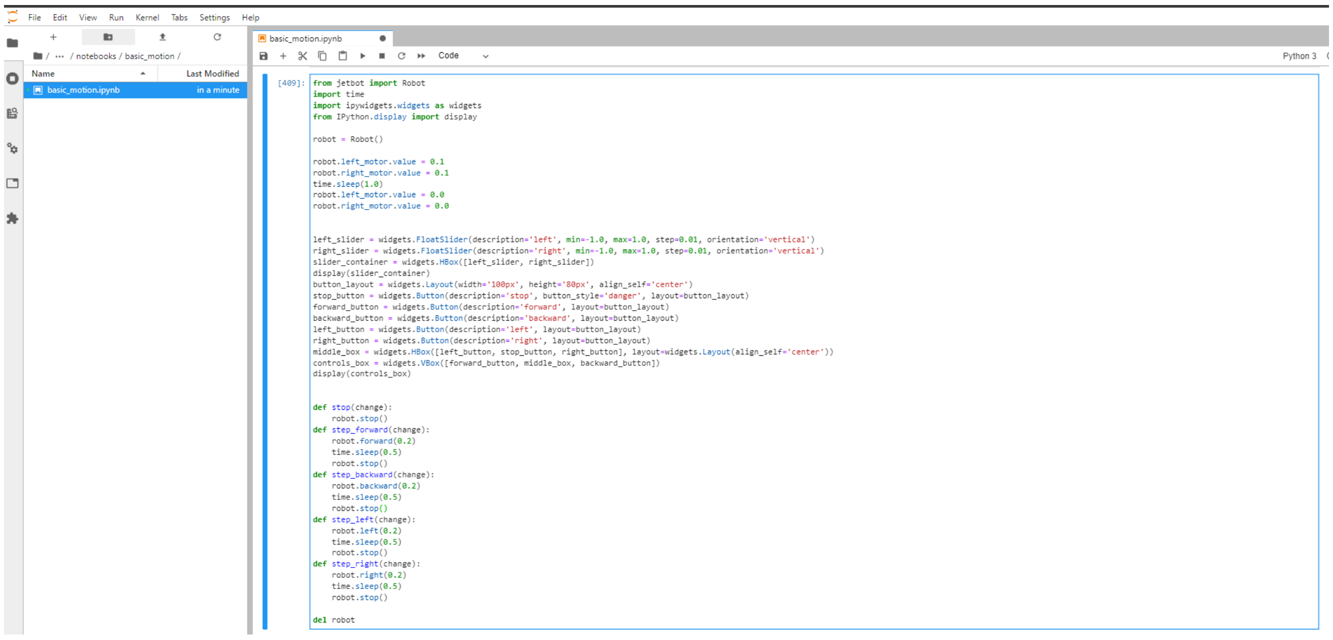
By addressing these missing elements and potential improvements, the application can be significantly enhanced in terms of usability, educational value, and functionality.

9. Robot Navigation: robot-navigation

The demonstration linked above showcases the successful navigation of the robot. The process involves testing the functionality of the implemented algorithms on a physical setup. Following this successful test, the subsequent task is to record the discovered path into a text file. This text file will later serve as input for the IoT students to generate a motor file. Surprisingly, this step requires minimal effort compared to the initial estimation.

Having already programmed the system to store the optimal path as a list and to print out the coordinates of each node along this path, only two additional lines of code are necessary. These lines will specify the destination file and write the computed results into it.

It's important to emphasize that the actual movement of the robot, guided by the algorithm's output, lies beyond the scope of my project. This aspect will be addressed by the IoT students. The linked video demonstrates the correctness of the path displayed on the map, serving solely for monitoring the robot's movement during testing and not as part of the final implementation.



```
[409]: from jetbot import Robot
import time
import ipywidgets.widgets as widgets
from IPython.display import display

robot = Robot()

robot.left_motor.value = 0.1
robot.right_motor.value = 0.1
time.sleep(1.0)
robot.left_motor.value = 0.0
robot.right_motor.value = 0.0


left_slider = widgets.FloatSlider(description='left', min=-1.0, max=1.0, step=0.01, orientation='vertical')
right_slider = widgets.FloatSlider(description='right', min=-1.0, max=1.0, step=0.01, orientation='vertical')
slider_container = widgets.HBox([left_slider, right_slider])
display(slider_container)
button_layout = widgets.Layout(width='100px', height='80px', align_self='center')
stop_button = widgets.Button(description='stop', button_style='danger', layout=button_layout)
forward_button = widgets.Button(description='forward', layout=button_layout)
backward_button = widgets.Button(description='backward', layout=button_layout)
left_button = widgets.Button(description='left', layout=button_layout)
right_button = widgets.Button(description='right', layout=button_layout)
middle_box = widgets.HBox([left_button, stop_button, right_button], layout=widgets.Layout(align_self='center'))
controls_box = widgets.VBox([forward_button, middle_box, backward_button])
display(controls_box)


def stop(change):
    robot.stop()
def step_forward(change):
    robot.forward(0.2)
    time.sleep(0.5)
    robot.stop()
def step_backward(change):
    robot.backward(0.2)
    time.sleep(0.5)
    robot.stop()
def step_left(change):
    robot.left(0.2)
    time.sleep(0.5)
    robot.stop()
def step_right(change):
    robot.right(0.2)
    time.sleep(0.5)
    robot.stop()

del robot
```

REFERENCES:

Hassanzadeh, Mahdi. “Hassanzadehmahdi/Romanian-Problem-Using-Astar-And-GBFS.” *GitHub*, 22 Feb. 2024, github.com/hassanzadehmahdi/Romanian-problem-using-Astar-and-GBFS. Accessed 7 Mar. 2024.

Soularidis, Andreas. “Uniform Cost Search (UCS) Algorithm in Python.” *Plainenglish.io/Blog/Uniform-Cost-Search-Ucs-Algorithm-In-Python-Ec3ee03fca9f*, 21 Feb. 2021, plainenglish.io/blog/uniform-cost-search-ucs-algorithm-in-python-ec3ee03fca9f.

“View Source for A-Star Algorithm - Cornell University Computational Optimization Open Textbook - Optimization Wiki.” *Optimization.cbe.cornell.edu*, optimization.cbe.cornell.edu/index.php?title=A-star_algorithm&action=edit. Accessed 7 Mar. 2024.

Walker, Alyssa. “Breadth First Search (BFS) Algorithm with EXAMPLE.” *Www.guru99.com*, www.guru99.com/breadth-first-search-bfs-graph-example.html.

“What’s the Difference between Best-First Search and A* Search?” *Stack Overflow*, stackoverflow.com/questions/34244452/whats-the-difference-between-best-first-search-and-a-search. Accessed 7 Mar. 2024.