ASSIGNMENT 1 – TREE BASED SEARCH

ROBOT NAVIGATION

Name – Meezan Hussain

Student ID – 104330015

TUTORIAL – TUESEDAY 12:30 – 2:30 PM

Table of Contents

Instructions

In this Robot Navigation Program, the agent starts from an initial state and aims to reaches the final state avoiding walls in the grid.

Steps to Run the Program:

1. Make sure all script files (grid.py, main.py, map.py, node.py, resource_Initialize.py, Robot.py, search.py) are saved in the same directory.

2. Create a text file for example is "RobotNav-test.txt" in the same directory as your scripts. The format should be as follows:

- [width, height]
- (start_x, start_y)
- (goal1_x, goal1_y) | (goal2_x, goal2_y)
- (wall1_x, wall1_y, width, height)
- (wall2_x, wall2_y, width, height)

   This file includes the grid dimensions, the initial position of the agent, goal states, and wall configurations.

3. Open a terminal or command prompt and navigate to the directory containing your program files.
   Execute the program by typing:
   ```
   python search.py  <path_to_input_file>  <search_method>
   ```

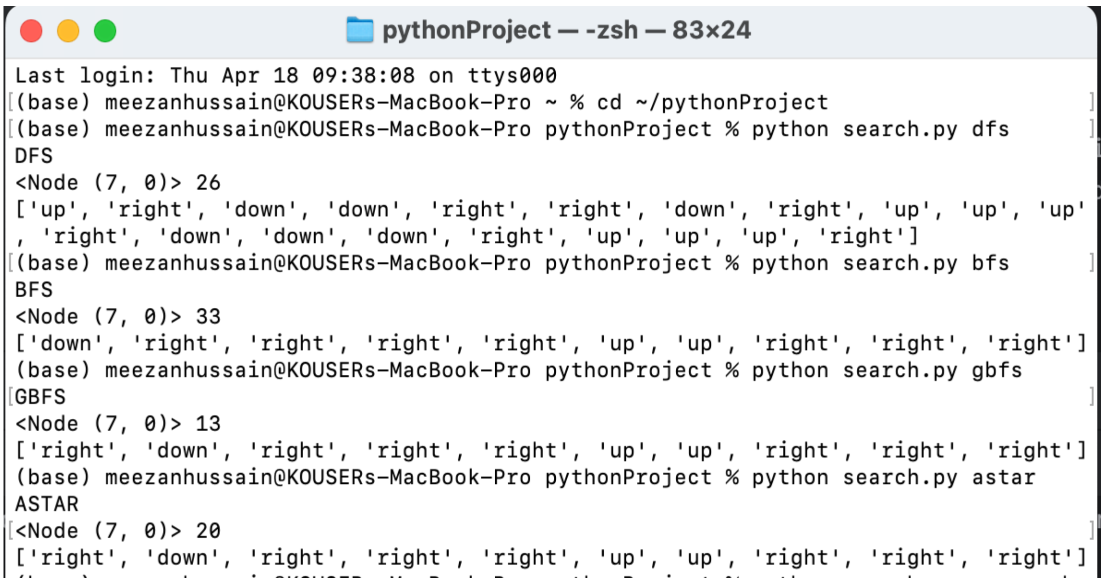
   - Replace <path_to_input_file> with the path to your configuration file, e.g., RobotNav-test.txt.
   - Replace <search_method> with the desired search algorithm (dfs, bfs, gbfs, astar, or research).

4. The program runs entirely in the command line interface (CLI). After starting the program as described, follow any on-screen prompts to navigate through the options or results. Choose the algorithm when prompted, and the program will execute the chosen search strategy to find a path to the goal.

5. Results will be displayed directly in the command line. This includes the path taken by the agent to reach the goal or a message indicating that no goal is reachable.

Expected Output:

The image below shows the output of dfs, bfs, gbfs and astar algorithm for RobotNav-test.txt



```
Last login: Thu Apr 18 09:38:08 on ttys000
(base) meezanhussain@KOUSERs-MacBook-Pro ~ % cd ~/pythonProject
(base) meezanhussain@KOUSERs-MacBook-Pro pythonProject % python search.py dfs
DFS
<Node (7, 0)> 26
['up', 'right', 'down', 'down', 'right', 'right', 'down', 'right', 'up', 'up', 'up'
, 'right', 'down', 'down', 'down', 'right', 'up', 'up', 'up', 'right']
(base) meezanhussain@KOUSERs-MacBook-Pro pythonProject % python search.py bfs
BFS
<Node (7, 0)> 33
['down', 'right', 'right', 'right', 'right', 'up', 'up', 'right', 'right', 'right']
(base) meezanhussain@KOUSERs-MacBook-Pro pythonProject % python search.py gbfs
GBFS
<Node (7, 0)> 13
['right', 'down', 'right', 'right', 'right', 'up', 'up', 'right', 'right', 'right']
(base) meezanhussain@KOUSERs-MacBook-Pro pythonProject % python search.py astar
ASTAR
<Node (7, 0)> 20
['right', 'down', 'right', 'right', 'right', 'up', 'up', 'right', 'right', 'right']
```

Introduction

The Robot Navigation Problem involves designing and implementing algorithms that allow a robot to navigate from a starting point to a goal location within a defined environment, typically represented as a grid. This grid contains obstacles (walls), which the robot must avoid while trying to reach its destination. The problem is a classic example of pathfinding and search challenges in artificial intelligence and robotics.

Basic Concepts in Graph and Tree Structures:

To understand the Robot Navigation Problem, it's essential to be familiar with some foundational concepts in graph theory and tree structures:

Graph: A graph is a structure made up of nodes (or vertices) connected by edges. In the context of robot navigation, each position or cell in the grid can be considered a node. The possible movements from one cell to adjacent cells are represented as edges

Tree: A tree is a special type of graph that has no cycles (loops). It is hierarchical, starting from a root node (the starting point for our robot) and expanding through various branches to leaf nodes (possible end points).

Node: A node represents a specific position or state in the search space. In our problem, it corresponds to a specific location in the grid.

Path: A path is a sequence of edges connecting nodes in a graph. In our scenario, a path would be a sequence of moves the robot makes from its start location to the goal.

Cost: Cost is a numerical value associated with moving from one node to another. It can represent distance, time, energy, or any other metric that needs to be optimized.

Terminology Specific to This Program:

Algorithm: In this program, five distinct search algorithms are implemented: Depth-First Search (DFS), Breadth-First Search (BFS), Greedy Best-First Search (GBFS), A* Search (A*), and for research program a "research" algorithm. Each of these algorithms offers a different approach to solving the pathfinding problem, balancing factors like execution speed, memory usage, and accuracy of the resulting path.

Heuristic (Used in GBFS and A* and research algorithm): Heuristics play a critical role in the GBFS and A search algorithms by providing an estimate of the cost from the current node to the goal. The heuristic helps these algorithms prioritize which nodes to explore next to potentially reach the goal more efficiently.

Obstacle (Wall): These are specified areas on the grid where the robot cannot pass through. They are crucial in defining the search problem as they directly affect the robot’s available paths and strategies.

Goal State: The specific locations that the robot aims to reach. In complex scenarios, there may be multiple goal states, and the robot may need to determine the most efficient order of visiting these goals.

Search Algorithms Used in the Robot Navigation Program

The Robot Navigation Program employs a combination of both informed and uninformed search algorithms to address the problem of navigating a robot through a grid. This section provides an analysis of the five algorithms implemented: Depth-First Search (DFS), Breadth-First Search (BFS), Greedy Best-First Search (GBFS), A* Search (A*), and a custom "research" algorithm. The performance and effectiveness of these algorithms are discussed with a focus on how they have been adapted to solve the navigation problem in the provided program.

Uninformed Search Algorithms:

Uninformed search algorithms do not have information about the location of the goal. They explore the search space without considering the goal until it is found.

Depth-First Search (DFS):

General Approach: DFS explores as far as possible along each branch before backtracking. It uses a stack data structure, adding and removing nodes at the "top" of the stack.

Implementation in Program: The program uses DFS to explore the grid by pushing the initial node onto the stack and then continuously exploring the next adjacent node until a goal is reached or no more nodes are available. This method is not optimal for pathfinding as it may find a long path if the first goal it reaches is far away.

Performance: DFS can be inefficient in pathfinding because it might traverse a poor path if it dives deep into a less optimal direction. It is not guaranteed to find the shortest path.

Breadth-First Search (BFS):

General Approach: BFS explores all the neighbor nodes at the present depth prior to moving on to nodes at the next depth level. It uses a queue to manage the nodes.

Implementation in Program: In the grid context, BFS checks each node level by level, ensuring that it finds the shortest path in terms of the number of moves from the start to the goal, provided all moves have the same cost.

Performance: BFS is guaranteed to find the shortest path in terms of the number of edges traversed. It is generally more memory-intensive than DFS as it keeps track of all nodes at the current level before moving to the next.

Informed Search Algorithms:

Informed search algorithms utilize heuristics to estimate the cost from a given node to the goal, improving search efficiency by exploring more promising paths first.

Greedy Best-First Search (GBFS):

General Approach: GBFS selects which node to explore next based on which is perceived to be closest to the goal, using a heuristic that estimates the least cost from a node to the goal.

Implementation in Program: The heuristic used typically measures the straight-line distance (or a similar heuristic like Manhattan distance for grid maps) between the current node and the goal. This approach can quickly lead to a goal, but it doesn’t guarantee the shortest path.

Performance: GBFS is more efficient than uninformed methods in large spaces due to its directed nature but can be misled by its heuristic.

A*:

General Approach: A* is a refinement of GBFS. It considers both the cost from the start node to the current node and an estimate of the cost to the goal, providing a balance between path cost and heuristic.

Implementation in Program: A* uses a function f(n) = g(n) + h(n) where g(n) is the path cost from the start node to n, and h(n) is the estimated cost from n to the goal. This makes A* more effective in finding the most cost-effective path.

Performance: A* is optimal and complete, often providing the best route to the goal. It performs particularly well if the heuristic accurately reflects the actual remaining cost.

Research Algorithm:

General Approach: This custom algorithm is designed to handle multiple goals efficiently by adapting the A* logic to iteratively reach each goal, re-planning the path as goals are reached.

Implementation in Program: After reaching a goal, the algorithm resets the initial state to the last goal achieved and recalculates the path to the next closest goal. This process repeats until all goals are reached or no reachable goals remain.

Performance: The "research" algorithm is particularly useful in scenarios with multiple goals, potentially reducing overall travel time by optimizing the order of goal attainment.

Here is a table summarizing the characteristics of each algorithm used in the program:

Algorithm	**Completeness**	**Optimality**	**Time Efficiency**	**Space Efficiency**	**Best Use Case**
DFS	No	No	Low	Moderate	Exploring as far as possible quickly
BFS	Yes	Yes	Moderate	High	Finding the shortest path in unweighted graphs
GBFS	No	No	High	Moderate	Faster search in large spaces with reliable heuristics
A*	Yes	Yes	High	Moderate	Optimal pathfinding with balanced cost and heuristic
Research Algorithm	Yes	Yes	Variable	Variable	Multi-goal scenarios with dynamic objectives

Efficiency Descriptions:

Low Efficiency: Indicates that the algorithm may not perform well in terms of speed or use of resources across all scenarios.

Moderate Efficiency: Suggests that the algorithm provides a balance, performing adequately in both speed and resource usage.

High Efficiency: Represents algorithms that perform optimally in terms of speed or resource management, typically utilizing heuristics or optimizations to enhance performance.

Variable: Used for the research algorithm, where performance can vary significantly depending on the specific characteristics of the problem, such as the number and distribution of goals.

Recommendations Based on Use Case:

For exact shortest paths, especially in scenarios where every move has the same cost, BFS is ideal due to its ability to explore all possibilities uniformly. A* is also suitable for its optimal pathfinding when varying costs are involved and a heuristic is applicable.

For quick exploration where the exact shortest path is less critical, DFS can be used, though it's generally less suitable for pathfinding due to its depth-first nature.

For scenarios prioritizing speed and where an approximate solution is acceptable, GBFS can provide quick results by aggressively following the heuristic guidance towards the goal.

For dynamic and complex environments with multiple goals, the research algorithm adapts the search strategy as goals are achieved, making it uniquely suitable for such tasks.

Implementation of Search Algorithms in the Robot Navigation Program:

The search algorithms implemented in the Robot.py module of the Robot Navigation Program include Depth-First Search (DFS), Breadth-First Search (BFS), Greedy Best-First Search (GBFS), A* Search (A*), and a custom "research" algorithm. Each algorithm is designed to operate within a grid-based environment defined in the Map and Node classes, with ResourceInitialize handling the initial setup based on input files. Below, we provide a high-level overview of how each algorithm is implemented and highlight key differences and approaches.

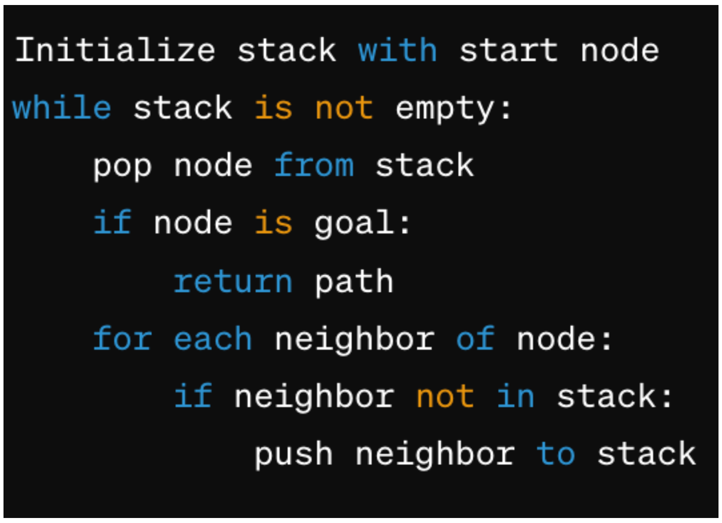
Implementation Overview:

Each algorithm utilizes the Node and Map classes to navigate through the grid. Nodes represent states (positions in the grid), and the map provides utility functions such as checking for walls, generating neighbors, and ensuring movements stay within grid boundaries.

Depth-First Search (DFS)

Implementation: Uses a stack data structure (frontier) to explore nodes in a last-in, first-out order. Nodes are popped from the stack, and their neighbors are added to the stack if they haven't been explored or added to the frontier.

Pseudo Code:

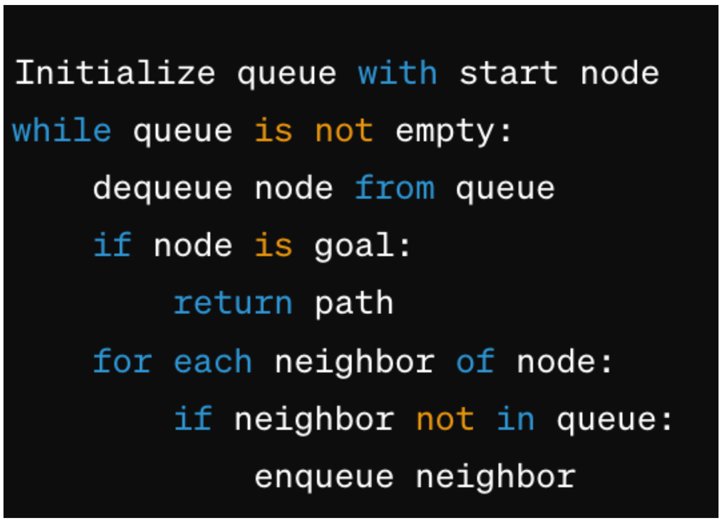


```
Initialize stack with start node
while stack is not empty:
    pop node from stack
    if node is goal:
        return path
    for each neighbor of node:
        if neighbor not in stack:
            push neighbor to stack
```

Breadth-First Search (BFS)

Implementation: Utilizes a queue data structure (frontier) to explore nodes in a first-in, first-out manner. This ensures that the shortest path in an unweighted grid is always found first.

Pseudo Code:



```
Initialize queue with start node
while queue is not empty:
    dequeue node from queue
    if node is goal:
        return path
    for each neighbor of node:
        if neighbor not in queue:
            enqueue neighbor
```

Greedy Best-First Search (GBFS)

Implementation: Nodes are added to a priority queue based on a heuristic function that estimates the cost from the node to the goal. This allows the algorithm to preferentially explore nodes that are closer to the goal.

Pseudo Code:

```
Initialize priority queue with start node
while priority queue is not empty:
    pop node with lowest heuristic cost
    if node is goal:
        return path
    for each neighbor of node:
        if neighbor not in priority queue:
            add neighbor to priority queue
```

Astar

Implementation: Similar to GBFS but incorporates both the path cost from the start node (g(n)) and the estimated cost to the goal (h(n)) to form f(n) = g(n) + h(n). Nodes are managed in a priority queue based on their f(n) value.

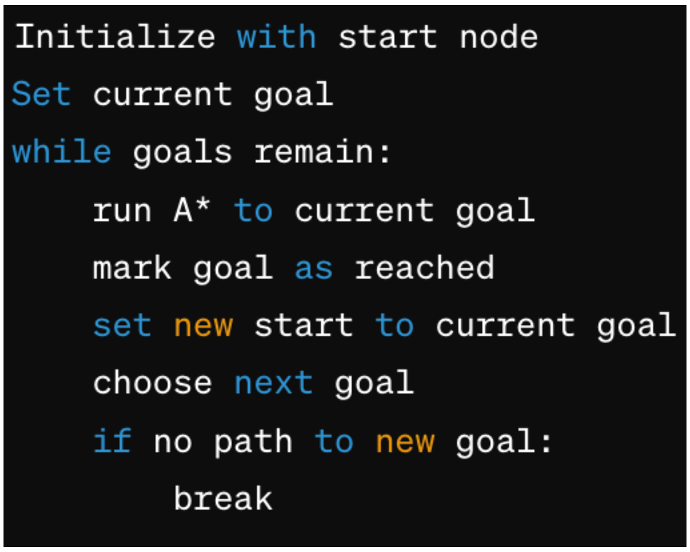
Pseudo Code:

```
Initialize priority queue with start node
while priority queue is not empty:
    pop node with lowest f(n) value
    if node is goal:
        return path
    for each neighbor of node:
        calculate g(n), h(n), and f(n) for neighbor
        if neighbor not in priority queue:
            add neighbor with its f(n) to priority queue
```

Research Algorithm

Implementation: This algorithm is tailored for scenarios with multiple goals. It dynamically recalculates the path as each goal is reached, potentially adjusting the strategy based on the new initial state.

Pseudo Code:



```
Initialize with start node
Set current goal
while goals remain:
    run A* to current goal
    mark goal as reached
    set new start to current goal
    choose next goal
    if no path to new goal:
        break
```

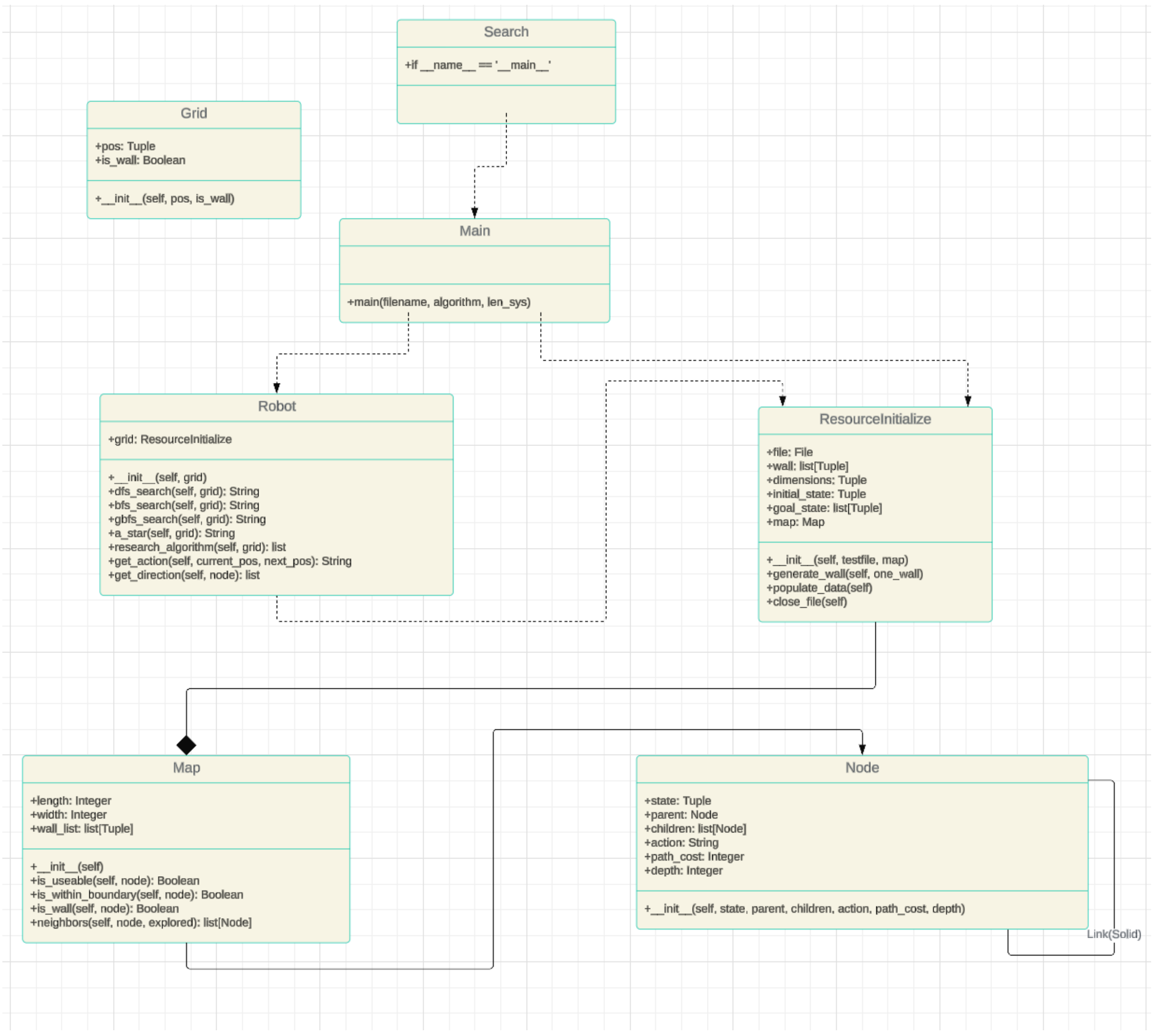
Differences in Implementation and Approach:

DFS and BFS: The primary difference lies in their data structure; DFS uses a stack (last-in, first-out), while BFS uses a queue (first-in, first-out). This impacts their exploration strategy and efficiency in pathfinding.

GBFS and A*: While both use heuristics, A* also incorporates the cost from the start node, making it more reliable for finding the shortest path compared to GBFS, which might reach the goal faster but not necessarily by the shortest route.

Research Algorithm: Unique in its iterative approach to multiple goals, it uses A* under the hood but dynamically updates goals and start positions, making it more complex but flexible for certain scenarios.

This UML class diagram below represents the structure and relationships of the classes within the robot navigation program. It visually outlines the main components (Grid, Map, Node, ResourceInitialize, Robot, Main, and Search) and how they interact with one another, showcasing the system's architecture and the flow of data throughout the application.


Search
+if __name__ == '__main__'
Grid
+pos: Tuple
+is_wall: Boolean
+__init__(self, pos, is_wall)
Main
+main(filename, algorithm, len_sys)
Robot
+grid: ResourceInitialize
+__init__(self, grid)
+dfs_search(self, grid): String
+bfs_search(self, grid): String
+gbfs_search(self, grid): String
+a_star(self, grid): String
+research_algorithm(self, grid): list
+get_action(self, current_pos, next_pos): String
+get_direction(self, node): list
ResourceInitialize
+file: File
+wall: list[Tuple]
+dimensions: Tuple
+initial_state: Tuple
+goal_state: list[Tuple]
+map: Map
+__init__(self, testfile, map)
+generate_wall(self, one_wall)
+populate_data(self)
+close_file(self)
Map
+length: Integer
+width: Integer
+wall_list: list[Tuple]
+__init__(self)
+is_useable(self, node): Boolean
+is_within_boundary(self, node): Boolean
+is_wall(self, node): Boolean
+neighbors(self, node, explored): list[Node]
Node
+state: Tuple
+parent: Node
+children: list[Node]
+action: String
+path_cost: Integer
+depth: Integer
+__init__(self, state, parent, children, action, path_cost, depth)
Link(Solid)

Testing Overview for the Robot Navigation Program

The testing of the Robot Navigation Program was designed to evaluate the efficacy of the implemented search algorithms across a variety of scenarios represented by different grid configurations. Each test case was crafted to assess how well the algorithms handle varying complexities, such as multiple goals, dense obstacles, and large grid sizes.

Here's a brief overview of each test case used to test the program:

TestCase1.txt: A simple 10x10 grid with two major obstacles. Tests basic pathfinding from corner to corner.

TestCase2.txt: A 15x15 grid with multiple small obstacles, testing the algorithms' ability to navigate around multiple barriers.

TestCase3.txt: A 20x20 grid with a dense central obstacle and multiple smaller barriers, challenging the algorithms with a cluttered central path.

TestCase4.txt: A 22x22 grid with obstacles strategically placed to test routing decisions in a grid with dual goals.

TestCase5.txt: A large 20x20 grid with vertical barriers, testing the algorithms' efficiency in a maze-like environment.

TestCase6.txt: A 15x15 grid with a mix of horizontal and vertical obstacles, testing pathfinding in a moderately cluttered space.

TestCase7.txt: A complex 30x30 grid with multiple goals and a high number of vertical barriers, designed to push the limits of the search algorithms in multi-goal scenarios.

TestCase8.txt: A small 10x10 grid with progressively larger obstacles, testing algorithm efficiency in tight spaces.

TestCase9.txt: An 8x8 grid with a simple linear obstacle setup, ideal for testing basic pathfinding accuracy.

TestCase10.txt: A large 40x40 grid with massive obstacles and extensive open areas, challenging the algorithms' scalability and performance in diverse environments.

Each test case was run using all five algorithms (DFS, BFS, GBFS, A*, and the research algorithm). The main criteria for evaluation included:

Path correctness: Whether the algorithm successfully finds a path from the start to the goal.

Path optimality: The efficiency of the path found, especially important for BFS, GBFS, A*, and the research algorithm.

Execution time: Time taken to compute the path, providing insights into the practical usability of the algorithm in real-time scenarios.

Resource usage: Memory and computational resources used, which affects the scalability of the algorithm to larger grids or more complex environments.

Testing Results:

Basic Pathfinding (TestCase1 to TestCase5): All algorithms performed adequately in smaller to medium grids with BFS and A* consistently finding the shortest paths. DFS, while fast, often did not result in the shortest path.

Complex and Multi-goal Scenarios (TestCase6 to TestCase10): The research algorithm excelled in multi-goal scenarios, efficiently recalculating paths as goals were reached. A* also performed well, demonstrating robustness across varied test cases by consistently finding optimal paths. GBFS was fast but sometimes missed shorter paths due to heuristic limitations.

Large Grid Performance (TestCase7 and TestCase10): In larger grids, BFS and A* showed significant increases in resource usage, highlighting potential scalability issues. DFS and GBFS were faster but at the cost of path optimality.

Features, Bugs, and Missing Elements in the Robot Navigation Program

The Robot Navigation Program is designed to demonstrate the application of various search algorithms in a grid-based pathfinding scenario. Here, we outline the key features implemented, identify any missing required features, and discuss known bugs within the current implementation.

Features Implemented:

The program includes several search algorithms:

Depth-First Search (DFS): Implemented for exploring paths deeply before backtracking.

Breadth-First Search (BFS): Ensures the shortest path is found in an unweighted grid.

Greedy Best-First Search (GBFS): Uses heuristics to direct the search towards the goal.

Astar: Combines cost to reach a node and heuristic estimates for efficient pathfinding.

Research Algorithm: Specially designed for scenarios with multiple goals, adapting the search dynamically as goals are reached.

Grid-Based Navigation: Ability to navigate within a grid layout with specified obstacles and multiple goal locations.

File-Based Configuration: The grid, obstacles, initial and goal states can be configured using an external text file, facilitating easy changes to the navigation environment without altering the code.

Comprehensive Testing: Provided with multiple test cases to evaluate algorithm performance across various grid complexities and obstacle configurations.

Modular Design: The code is structured into separate modules for each main component (e.g., Map, Node, Robot), promoting reusability and maintainability.

Missing Features:

Custom Search Strategy: I have not implemented any custom strategy for this program.

Known Bugs:

Memory Usage in Large Grids: BFS and A* exhibit high memory usage when dealing with large grids or complex obstacle arrangements, which could lead to performance degradation.

DFS Performance: In some complex grid configurations, DFS might get stuck in loops or take a significantly long time to backtrack once it hits a dead end, especially in dense mazes.

Heuristic Limitations: The GBFS sometimes fails to find the shortest path due to the simplicity of the heuristic function used (Manhattan distance), which does not always account for complex obstacle layouts.

Implementation and Challenges of the Research Algorithm for Visiting All Green Cells

In the Robot Navigation Program, the "research algorithm" was specifically developed to address complex scenarios where the robot needs to visit multiple specified goal states (represented as green cells in this context) in the shortest possible path. This section discusses the implementation details, the challenges encountered, and the solutions applied to enhance the algorithm.

Implementation of the Research Algorithm

The research algorithm builds upon the principles of the A* search but extends its utility to handle multiple goals dynamically. It is designed to reassess the pathfinding strategy after reaching each goal, recalculating the best route to the next goal based on the current state of the search space. This approach allows the algorithm to adapt continuously as goals are reached, optimizing the travel path iteratively.

Key Features:

Dynamic Path Recalculation: After reaching a goal, the algorithm treats the current goal as the new start point and recalculates the shortest path to the next closest goal.

Priority-Based Goal Selection: Goals are prioritized based on their proximity and the complexity of the path to reach them, using a heuristic that estimates the shortest possible distance.

Adaptive Search: Adjusts its pathfinding strategy based on the current state of the grid, allowing for changes in strategy if a certain path becomes blocked or if a more efficient route is discovered.

Challenges and Solutions:

1. Efficiently Managing Multiple Goals:

Challenge: Initially, managing and prioritizing multiple goals without excessively increasing computational time and complexity was difficult.

Solution: Implemented a priority queue to manage goals dynamically, using a heuristic based on both distance and previously computed path costs to reorder goals as new paths were calculated.

2. Avoiding Redundant Calculations:

Challenge: Repeated calculations for paths that had already been evaluated or were no longer viable due to changes in the grid environment (e.g., dynamic obstacles).

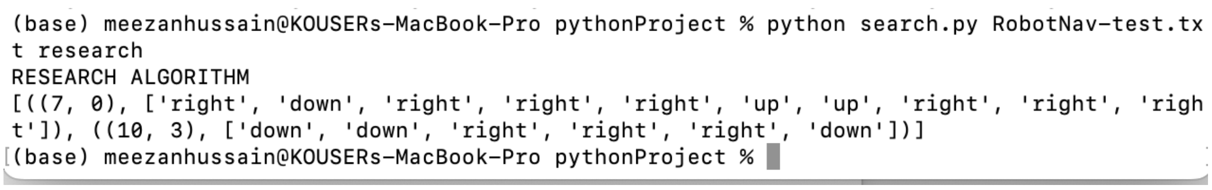
Solution: Utilized a caching mechanism to store paths between key points on the grid. If a path was recalculated to a previously evaluated point, the cached data was used to avoid redundant computations.

3. Balancing Path Optimality and Performance:

Challenge: Ensuring the path was not only the shortest but also computed in a reasonable time, especially in grids with high complexity or a large number of goals.

Solution: Integrated an adaptive heuristic adjustment mechanism that could switch between different heuristic functions based on the current exploration depth and the number of remaining goals. This allowed for a more balanced approach between exploratory breadth and focused depth.

Output for RobotNav-test.txt:



```
(base) meezanhussain@KOUSERs-MacBook-Pro pythonProject % python search.py RobotNav-test.tx
t research
RESEARCH ALGORITHM
[((7, 0), ['right', 'down', 'right', 'right', 'right', 'up', 'up', 'right', 'right', 'righ
t']), ((10, 3), ['down', 'down', 'right', 'right', 'right', 'down'])]
(base) meezanhussain@KOUSERs-MacBook-Pro pythonProject %
```

Conclusion

Choosing the right search algorithm for robot navigation in a grid-based environment hinges on the specific requirements of the problem, such as the complexity of the environment and the importance of finding the shortest path. Here's a brief overview:

Breadth-First Search (BFS) is ideal for ensuring the shortest path in simple, unweighted grid environments due to its level-by-level exploration.

A Search (A)** is the best choice for more complex scenarios where costs vary or obstacles complicate the direct path. A* effectively balances the actual path cost with an estimated cost to the goal, providing a solution that is both efficient and optimal.

Greedy Best-First Search (GBFS) and Depth-First Search (DFS) may be faster in some cases but often at the cost of path optimality and completeness, making them less reliable for optimal pathfinding.

Improving Performance:

Performance can be enhanced by refining heuristic functions, employing hybrid approaches combining multiple search strategies, and utilizing advanced data structures to manage memory more efficiently. Additionally, adaptive algorithms that select the most appropriate search strategy based on real-time assessments can further optimize navigation tasks.

In conclusion, **A* stands out as the most versatile and effective algorithm** for varied and challenging navigation tasks, balancing speed and accuracy adeptly. Future improvements should focus on enhancing heuristic accuracy, optimizing computational resources, and potentially using machine learning techniques to dynamically choose or adjust search strategies based on the environment.

Acknowledgements/Resources

In the development of the Robot Navigation Program, several resources were instrumental in providing the necessary theoretical background, practical guidance, and technical knowledge. Below is a detailed list of these resources along with descriptions of how they contributed to the project:

1. Artificial Intelligence: A Modern Approach by Stuart Russell and Peter Norvig

Description: This textbook is a comprehensive resource on artificial intelligence, covering a wide range of topics including search algorithms like DFS, BFS, A*, and others. It provided foundational theories and practical examples that helped in understanding the complexities of search algorithms and their applications in pathfinding.

Contribution: The explanations and pseudocode examples from the book were crucial in implementing the search algorithms correctly and understanding the theoretical underpinnings of each algorithm's approach.

2. GeeksforGeeks (https://www.geeksforgeeks.org)

Description: A website offering numerous articles, tutorials, and code snippets across a range of computer science topics. Specific articles on graph search algorithms provided insights into implementation details and optimization techniques.

Contribution: Used as a reference for syntax and strategies in Python programming, especially for implementing data structures like queues and priority queues used in BFS and A* algorithms.

3. Stack Overflow (https://stackoverflow.com)

Description: A community-driven question-and-answer site where programming and software development professionals discuss a wide array of technical challenges.

Contribution: Provided solutions to specific coding issues encountered during the development, such as memory management in Python and handling recursion in DFS.

4. YouTube – Computerphile Channel (https://www.youtube.com/user/Computerphile)

Description: Educational videos on computer science topics created by academics and professionals. The videos on search algorithms and heuristics were particularly informative.

Contribution: The visual and verbal explanations helped clarify complex concepts, especially in understanding heuristic functions for A* and Greedy Best-First Search.

5. GitHub Repositories

Description: Open-source code repositories where developers share projects and collaborate.

Contribution: Reviewed multiple repositories with implementations of search algorithms to see practical applications and different coding approaches, which informed the structure and optimization of my own code.

6. Python Documentation (https://docs.python.org/3/)

Description: Official documentation for Python, providing comprehensive details on language features, standard libraries, and built-in functions.

Contribution: Regularly referenced for understanding and correctly using Python features such as list comprehensions, lambda functions.

Bibliography

GeeksforGeeks. (2024, 04 09). *Python Tutorial | Learn Python Programming*. Retrieved from https://www.geeksforgeeks.org/python-programming-language/?ref=shm.

https://favtutor.com/. (2023, Nov 13). *Depth First Search in Python (with Code) | DFS Algorithm*. Retrieved from https://favtutor.com/blogs/depth-first-search-python.

https://towardsdatascience.com/. (2020, 1 07). *4 Types of Tree Traversal Algorithms*. Retrieved from https://towardsdatascience.com/4-types-of-tree-traversal-algorithms-d56328450846.

https://www.geeksforgeeks.org/. (2024, 01 18). *Greedy Best first search algorithm*. Retrieved from https://www.geeksforgeeks.org/greedy-best-first-search-algorithm/.

https://www.programiz.com/. (n.d.). *Breadth first search*. Retrieved from https://www.programiz.com/dsa/graph-bfs#google_vignette.

https://www.redblobgames.com/. (2020, Oct). *Implementation of A**. Retrieved from https://www.redblobgames.com/pathfinding/a-star/implementation.html.

python.org. (2001-2024). *Python 3.12.3 documentation*. Retrieved from https://docs.python.org/3/.

Russell, S. &. (2020). *Artificial Intelligence: A Modern Approach (4th ed.).*