ROBOT NAVIGATION

Assignment 1- Tree Based Search

Md Radif Rafayet Chowdhury (103539316)
103539416@student.swin.edu.au

Table of Contents

Abstract

The present report examines the implementation and comparison of various tree-based search algorithms within a robot navigation simulation. By constructing a simulation environment based on the provided grid specifications, various uninformed and informed search algorithms have been implemented and evaluated. The algorithms explored include Depth-First Search (DFS), Breadth-First Search (BFS), Greedy Best-First Search (GBFS), A*, Iterative Deepening Search (IDS), and Beam Search.

The comparative analysis is focused on assessing the algorithms' performance in terms of path optimality, computational efficiency, and completeness. The report further discusses the implementation details, highlights the advantages of heuristic-informed searches, and provides a comparative analysis of the experimental results obtained from the test cases run against a diverse set of grid-based challenges. The findings aim to contribute to the optimization of robotic pathfinding in artificial intelligence applications.

Instructions

To successfully run the robot navigation program, please follow the detailed instructions outlined below:

Prerequisites:

1. Ensure that Python is installed on your system. Python can be downloaded from python.org.
2. The Pygame library is required for visualization purposes. It can be installed using pip, Python's package installer, with the following command: **`pip install pygame`**.

Setting Up the Environment:

1. Gather the necessary Python scripts **(`algorithm.py`, `grid.py`, `search.py`, `test.py`)** and the **`RobotNav-test.txt`** file. These files must be placed in the same directory to allow the scripts to interact with each other properly.
2. The **`RobotNav-test.txt`** file should be formatted correctly to describe the grid environment. It includes the grid dimensions, initial state, goal states, and obstacles. This file is crucial for the program to understand the navigation space.

Running the Program:

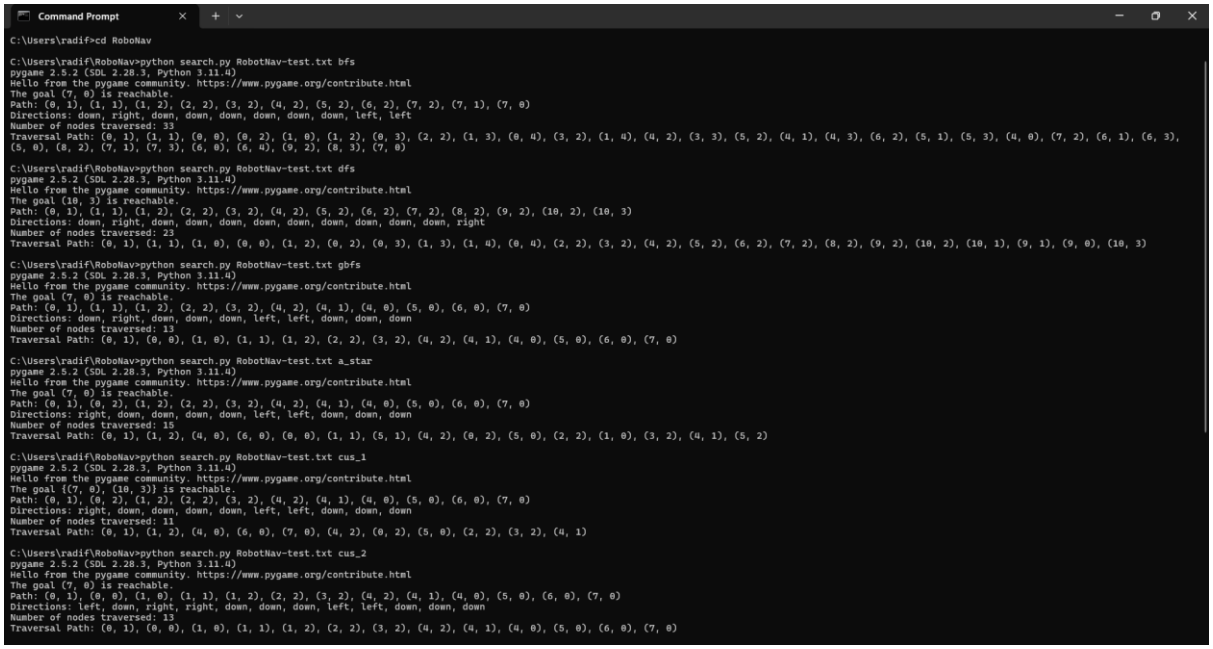
1. Open a terminal or command prompt window.
2. Navigate to the directory containing the program files using the cd command. For example: **`cd path_to_directory`.**
3. Run the **`search.py`** script with two arguments: the filename of the grid configuration and the chosen search method. The syntax is as follows: **`python main.py <map_filename> <search_method>`**. Replace **`<map_filename>`** with the name of your grid configuration file (e.g., **`RobotNav-test.txt`**) and **`<search_method>`** with one of the available methods **(`'dfs'`, `'bfs'`, `'gbfs'`, `'a_star'`, `'cus_1'`** for Iterative Deepening Search, or **`'cus_2'`** for Best First Search).

Example command:

C:\Users\radif>cd RoboNav

C:\Users\radif\RoboNav>python search.py RobotNav-test.txt dfs

Here is the expected output:



Understanding the Output:

1. Upon execution, the program performs the selected search algorithm on the grid described in the **`RobotNav-test.txt`** file. The path found by the algorithm (if any) will be printed to the console, including the sequence of moves and the total number of steps.
2. Simultaneously, the Pygame window will visualize the navigation of the robot on the grid. The robot's movement from the starting point to the goal will be animated based on the search algorithm's path.

Introduction

The field of Artificial Intelligence (AI) holds many practical challenges, one of which is navigating an environment efficiently. This report introduces a simulation where a robot, or agent, finds its way through a grid. The grid, detailed in the RobotNav-test.txt file, is an arrangement of spaces where the robot starts at one point and aims to reach a target location, all while avoiding blocks that represent obstacles.

To guide the robot, different search algorithms are put to the test. These include simpler methods like Depth-First Search (DFS) and Breadth-First Search (BFS), which explore without any directional

guidance, and more advanced methods like Greedy Best-First Search (GBFS) and A*, which use a strategy known as heuristics to make smarter choices about which way to go.

The exploration doesn't stop at common techniques; it extends to Iterative Deepening Search (IDS) and Best First Search, adding layers of complexity and potentially offering better ways to solve the navigation problem. The success of these methods is evaluated by seeing how they work on a set of example grid layouts, examining how fast and effectively they reach the goal.

A key part of this project is not only running these algorithms but also watching them in action. Using the Pygame library, the simulation visually tracks the robot's path across the grid, offering an immediate understanding of each method's approach to problem-solving.

This report aims to simplify the comparison of these methods, offering insight into their performance and laying the groundwork for further study. The goal is to make sense of how these algorithms can be applied to real-life AI challenges and what they tell us about building intelligent systems that can navigate the world around them.

Search Algorithms

In addressing the complexities of robotic navigation within a structured grid, our examination has encompassed a variety of search algorithms. Each algorithm is distinguished by its unique approach to traversing the grid and locating the goal amidst obstacles.

Uninformed Search Algorithms:

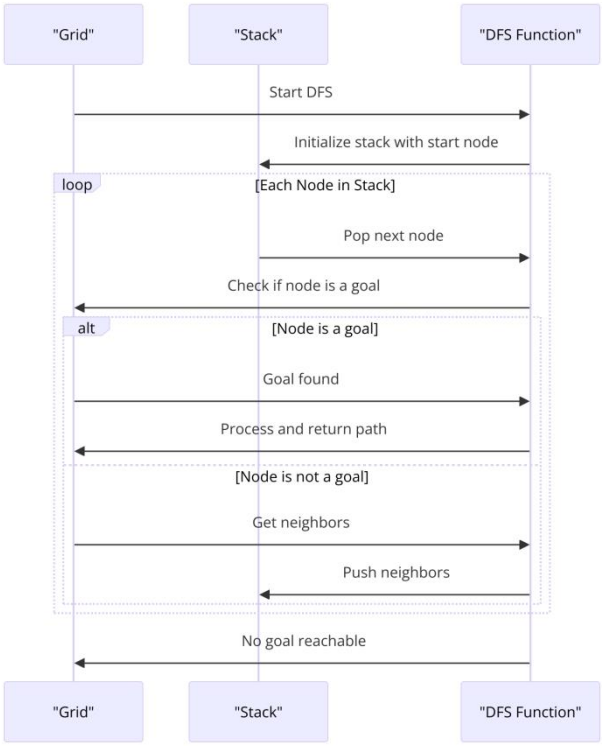
Uninformed search algorithms, also known as blind search algorithms, do not have any information about the goal's location or the distance to the goal. They only have access to the problem's structure and must explore the search space through trial and error.

1. **Depth-First Search (DFS):**

DFS is akin to navigating a maze by always taking the next available turn and retracing steps when a dead end is reached. It delves deep into one direction until it can go no further before considering alternatives. Its memory usage is minimal since it keeps track of only one path at a time. The caveat of DFS is its potential to overlook the shortest path in its deep exploration methodology. Here is an example of DFS algorithm has been applying in this assignment using python to find the path to its goal:

- **How does the DFS search function work?**

DFS explores as far as possible down a branch before backing up. In a typical implementation, DFS starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking. This property allows the algorithm to visit all nodes of the tree or graph without getting caught in cycles.



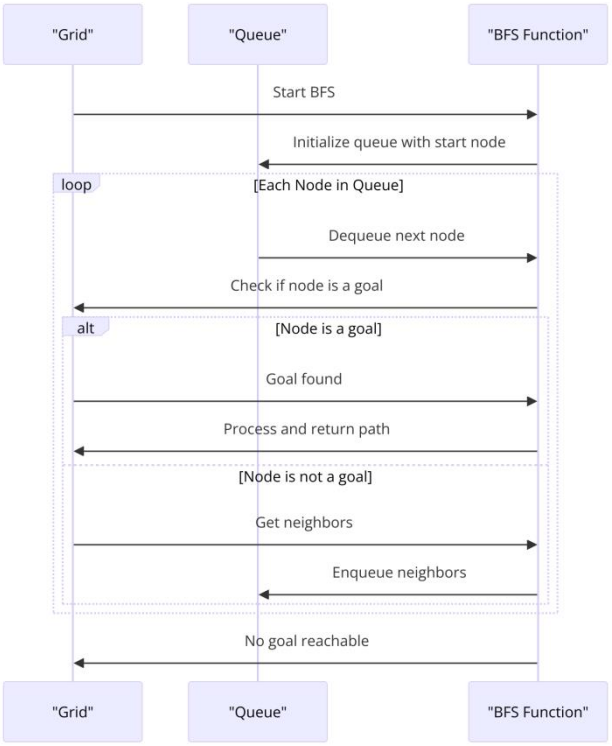
For DFS, the visual would depict a traversal where the path proceeds from the root node down to the deepest node along a branch before retreating to the nearest fork with unexplored paths.

2. Breadth-First Search (BFS):

BFS systematically explores all neighbouring points before moving on to the next set of neighbours, much like a ripple spreading out from a stone thrown into water. This ensures that the shortest path will not be missed if it exists. The primary drawback is its high memory requirement, as it must maintain a record of all immediate options at each step. Here is an example of BFS algorithm has been applying in this assignment using python to find the path to its goal:

- **How does the BFS search function work?**

BFS begins at the tree root and explores all neighbour nodes at the present level before moving to nodes at the next level. This strategy allows BFS to always find the shortest path to the goal node in an unweighted graph.



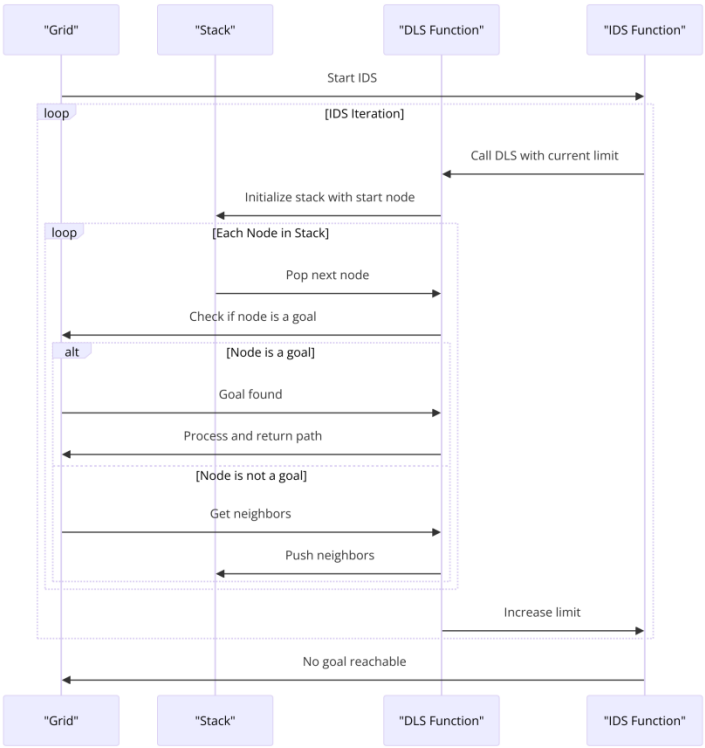
The BFS diagram would show an expansion in concentric circles from the start node, reaching all accessible nodes in waves according to their distance from the start.

3. Iterative Deepening Search (IDS):

IDS melds the methodologies of DFS and BFS. It incrementally explores deeper into the grid, resetting and expanding its reach progressively, akin to casting wider nets in successive fishing attempts. This incremental expansion enables IDS to combine the thoroughness of BFS with the lower memory footprint of DFS. Here is an example of IDS algorithm has been applying in this assignment using python to find the path to its goal:

- **How does the IDS search function work?**

IDS is a method that combines the depth-first search's space-efficiency with the breadth-first search's completeness. It does this by repeatedly executing a depth-limited search, each time increasing the depth limit until the goal is found. It's as if IDS is probing the depths of the problem space incrementally, ensuring it doesn't miss any solutions while also keeping memory usage to a minimum.



For IDS, you would create a diagram showing a series of depth-limited searches, each going deeper than the last. The diagram would illustrate the same tree being searched multiple times, with each successive search exploring deeper levels of the tree.

Informed Search Algorithms:

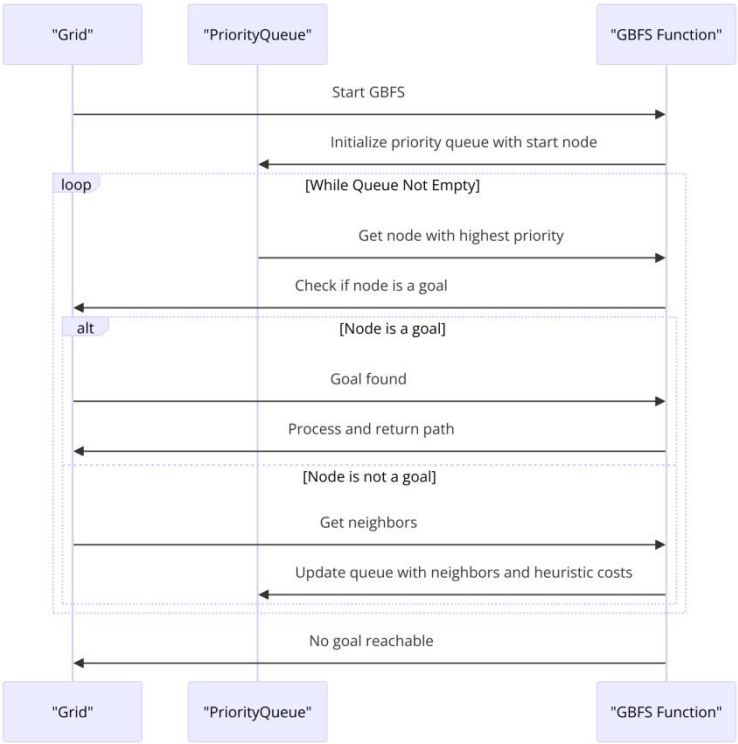
Informed search algorithms, or heuristic search algorithms, utilize additional information (heuristics) about the problem to make more educated guesses about which path to follow. This approach aims to find solutions more efficiently by estimating how close a given state is to the goal state.

1. **Greedy Best-First Search (GBFS):**

GBFS operates under a 'closest-first' principle, prioritizing moves that seem to lead more directly toward the goal. It's akin to choosing the most direct path visible on a hike. While often efficient, this approach can lead to suboptimal paths as it may be easily misled by layouts that require a less direct route to achieve the quickest end. Here is an example of GBFS algorithm has been applying in this assignment using python to find the path to its goal:

- **How does the GBFS search function work?**

GBFS selects the path that appears to lead most directly to the goal. It uses a heuristic that estimates how close the end goal is to each node. The algorithm follows the path that looks shortest, without considering the travelled cost, which sometimes leads to a quick but non-optimal solution.



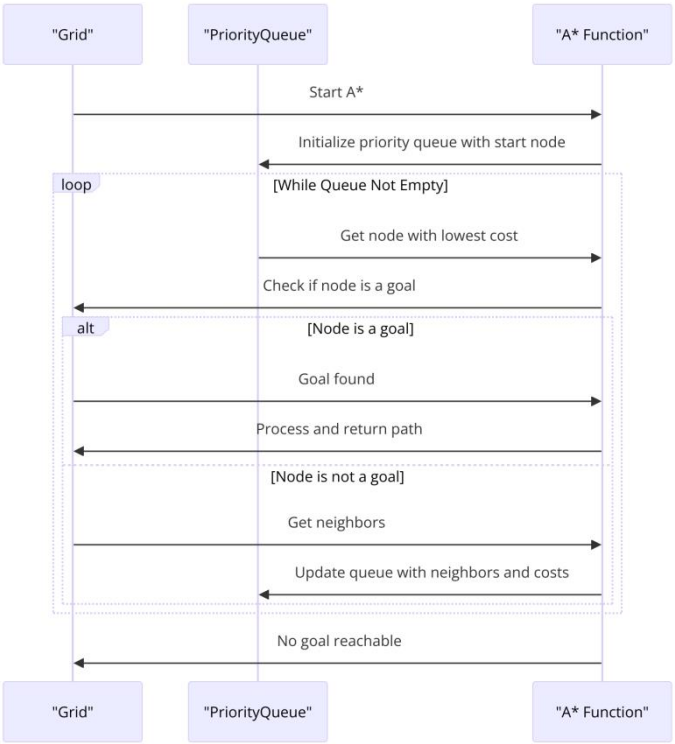
The GBFS visual might show a tree with estimated distances from each node to the goal, with the algorithm choosing the lowest estimated path at each step.

2. A* Search (a_star):

A* search augments the GBFS with consideration for the journey already undertaken. This algorithm is comparable to a seasoned traveller who balances both the remaining distance to the destination and the path already traversed to determine the next step. This dual consideration generally guides A* to the most efficient route, balancing speed with accuracy. Here is an example of A* algorithm has been applying in this assignment using python to find the path to its goal:

- **How does the A* search function work?**

A* search uses both the cost of the path travelled and a heuristic estimate of the cost to reach the end goal. This heuristic guides the path to the goal and ensures that the path is efficient. The priority queue orders nodes based on the sum of the path cost and the heuristic estimate, often leading to a quick and optimal solution.



The diagram for A* would typically show a grid with varying path costs and heuristic estimates, with the algorithm selecting the path that minimizes both.

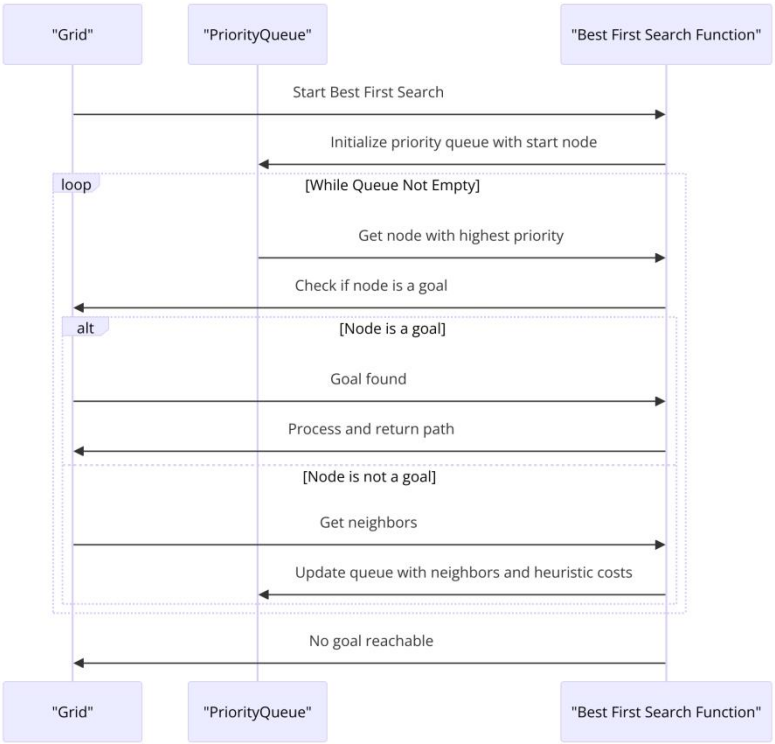
3. Best First Search (BS)

Best First Search is a heuristic-driven algorithm that intelligently navigates through a problem space by selecting the most promising path based on a heuristic function. Unlike depth-first search, it does not blindly delve into paths but rather evaluates them based on their potential to lead closer to the goal. This method balances exploration and exploitation by using the heuristic to make informed decisions about which node to explore next. The primary advantage of Best First Search is its efficiency in reaching the goal when a good heuristic is available.

- **How does Best First Search function work?**

Best First Search operates by maintaining a priority queue where nodes are prioritized based on their heuristic values relative to the goal. The search begins at the start node and at each step expands the node with the lowest heuristic cost. This process continues until the goal is reached or the search space is exhausted. The heuristic function is key to Best First Search; it estimates the cost from the current node to the goal, guiding the search in a direction that is expected to lead to the goal most quickly.

In a typical Python implementation, Best First Search uses a priority queue to keep the nodes sorted by their heuristic values. The algorithm repeatedly extracts the node with the lowest heuristic value, explores its neighbours, and updates their costs and the queue accordingly. This ensures that the search is always directed towards the goal in a heuristic manner.



For Best First Search, the visual would depict the nodes being explored in order of their estimated cost to the goal, with paths diverging from the start node and converging on the goal node as the search progresses. Each node on the frontier is selected based on its potential to bring the search closer to the goal, prioritizing exploration that promises to be most efficient according to the heuristic.

The investigation into these algorithms provides vital insights into their operational effectiveness. With each algorithm's intrinsic merits and limitations, the endeavour of this report is to distil their applicability to real-world navigation tasks. The objective is to ascertain a robust, efficient solution that can adapt to the varying demands of complex environments that autonomous agents may encounter.

Implementation

Overview

The implementation of the search algorithms in the Python-based robot navigation simulation encompasses a variety of both uninformed and informed search strategies. Each algorithm has been tailored to operate within a grid environment parsed from a 'RobotNav-test.txt' file. Below is a detailed description of the implementation and approach of each algorithm, highlighting the differences in their operational logic and potential use cases.

Uninformed Search Algorithms

Breadth-First Search (BFS):

Implementation: BFS was implemented using a queue to ensure that it explores all nodes at the present depth level before moving on to nodes at the next level. This approach ensures that the shortest path in an unweighted grid is always found.

```
# Breadth-First Search
def bfs(grid, start=None):
    if start is None:
        start = grid.start
    queue = deque([(start, [start])])
    visited = set()
    visited_order = []

    while queue:
        current, path = queue.popleft()
        visited.add(current)
        visited_order.append(current)

        if current in grid.goals:
            print_path_info(current, path, visited_order)
            return path, visited_order

        for neighbor in grid.get_neighbors(current):
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append((neighbor, path + [neighbor]))

    print("No goal is reachable.")
    return None, visited_order
```

Key Points: The simplicity of BFS and its ability to find the shortest path make it suitable for scenarios where path optimality is crucial. However, its memory intensity can be a drawback in larger grids.

Depth-First Search (DFS):

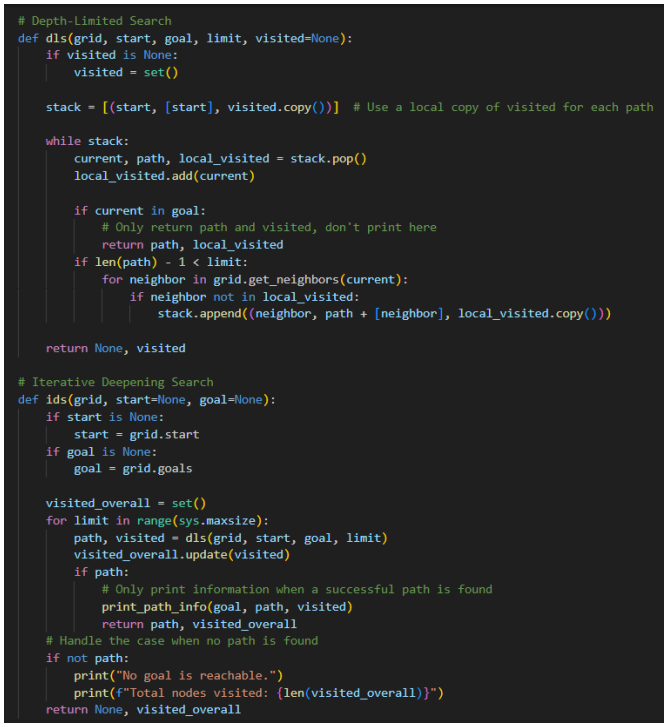
Implementation: DFS utilizes a stack to explore as far as possible along a branch before backtracking. This method is deeply recursive and tends to explore more thoroughly at the expense of memory efficiency.

```
# Depth-First Search
def dfs(grid, start=None):
    if start is None:
        start = grid.start
    stack = [(start, [start])]
    visited = set()
    visited_order = []

    while stack:
        current, path = stack.pop()
        if current not in visited:
            visited.add(current)
            visited_order.append(current)

            if current in grid.goals:
                print_path_info(current, path, visited_order)
                return path, visited_order

            for neighbor in reversed(grid.get_neighbors(current)):
                if neighbor not in visited:
                    stack.append((neighbor, path + [neighbor]))

    print("No goal is reachable.")
    return None, visited_order
```

Key Points: While DFS can be implemented more simply than BFS, its tendency to dive deep into paths can lead to inefficient searching and higher memory use in dense grids.

Iterative Deepening Search (IDS):

Implementation: IDS combines the depth-first traversal's space-efficiency with the breadth-first search's optimality by progressively deepening the search depth.



```python
# Depth-Limited Search
def dls(grid, start, goal, limit, visited=None):
    if visited is None:
        visited = set()

    stack = [(start, [start], visited.copy())]  # Use a local copy of visited for each path

    while stack:
        current, path, local_visited = stack.pop()
        local_visited.add(current)

        if current in goal:
            # Only return path and visited, don't print here
            return path, local_visited
        if len(path) - 1 < limit:
            for neighbor in grid.get_neighbors(current):
                if neighbor not in local_visited:
                    stack.append((neighbor, path + [neighbor], local_visited.copy()))

    return None, visited

# Iterative Deepening Search
def ids(grid, start=None, goal=None):
    if start is None:
        start = grid.start
    if goal is None:
        goal = grid.goals

    visited_overall = set()
    for limit in range(sys.maxsize):
        path, visited = dls(grid, start, goal, limit)
        visited_overall.update(visited)
        if path:
            # Only print information when a successful path is found
            print_path_info(goal, path, visited)
            return path, visited_overall
    # Handle the case when no path is found
    if not path:
        print("No goal is reachable.")
        print(f"Total nodes visited: {len(visited_overall)}")
    return None, visited_overall
```

Key Points: IDS is particularly effective in scenarios where the depth level is unknown or infinite. It offers a compromise between BFS's memory requirements and DFS's lack of optimality.

Informed Search Algorithms

Greedy Best-First Search (GBFS):

Implementation: GBFS was implemented using a priority queue that orders nodes based on a heuristic estimate of distance from the node to the goal, prioritizing nodes that are ostensibly closer to the goal. This heuristic-driven approach attempts to minimize the estimated cost to the goal, disregarding the cost accrued to reach the current node.

```python
# Greedy Best-First Search
def gbfs(grid, start=None):
    if start is None:
        start = grid.start
    frontier = PriorityQueue()
    frontier.put((0, start, [start]))
    visited = set()
    visited_order = []

    while not frontier.empty():
        _, current, path = frontier.get()
        visited.add(current)
        visited_order.append(current)

        if current in grid.goals:
            print_path_info(current, path, visited_order)
            return path, visited_order

        for neighbor in grid.get_neighbors(current):
            if neighbor not in visited:
                visited.add(neighbor)
                priority = heuristic(neighbor, list(grid.goals)[0])
                frontier.put((priority, neighbor, path + [neighbor]))

    print("No goal is reachable.")
    return None, visited_order
```

Key Points: The implementation focuses on reaching the goal as quickly as possible, which can sometimes lead to suboptimal paths if the heuristic is misled by complex grid layouts.

A* Search (a_star):

Implementation: A* also uses a priority queue but combines the cost to reach a node and the heuristic estimate of the distance to the goal. This dual consideration helps balance between path cost and estimated remaining distance, leading to more efficient paths.

algorithm.py

C: > Users > radif > RoboNav > algorithm.py > heuristic

```python
def heuristic(current, goal):
    # Manhattan distance as heuristic
    return abs(goal[0] - current[0]) + abs(goal[1] - current[1])

def get_directions(path):
    directions = []
    for i in range(1, len(path)):
        dx = path[i][0] - path[i - 1][0]  # Change in x-coordinate (row)
        dy = path[i][1] - path[i - 1][1]  # Change in y-coordinate (column)
        if dy == 1:
            directions.append("down")
        elif dy == -1:
            directions.append("up")
        elif dx == 1:
            directions.append("right")
        elif dx == -1:
            directions.append("left")
    return directions

def a_star(grid, start=None, goals=None):
    if start is None:
        start = grid.start
    if goals is None:
        goals = grid.goals

    frontier = PriorityQueue()
    frontier.put((0, start, [start]))
    visited = set()
    cost_so_far = {start: 0}
    goal = list(goals)[0]  # Assuming a single goal for simplicity in heuristic calculation

    while not frontier.empty():
        _, current, path = frontier.get()

        if current in goals:
            # Formatting and printing the output when the goal is reached
            print(f"The goal {goal} is reachable.")
            print(f"Path: {', '.join(map(str, path))}")
            print(f"Directions: {', '.join(get_directions(path))}")
            print(f"Number of nodes traversed: {len(visited)}")
            print(f"Traversal Path: {', '.join(map(str, visited))}")
            return path, visited

        if current not in visited:
            visited.add(current)
            for neighbor in grid.get_neighbors(current):
                new_cost = cost_so_far[current] + 1  # Assuming uniform cost
                if neighbor not in visited or new_cost < cost_so_far.get(neighbor, float('inf')):
                    cost_so_far[neighbor] = new_cost
                    priority = new_cost + heuristic(neighbor, goal)
                    frontier.put((priority, neighbor, path + [neighbor]))

    print("No goal is reachable.")
```

Key Points: The heuristic function plays a crucial role in guiding the A* search, making its choice (e.g., Manhattan vs. Euclidean) significant for performance and effectiveness.

Best First Search:

Implementation: Similar to GBFS, Best First Search utilizes a priority queue sorted by a heuristic value. However, it generally involves a more sophisticated heuristic that may consider more than just the straight-line distance to the goal.

```python
#Best First Search
def best_first_search(grid, start=None):
    if start is None:
        start = grid.start
    frontier = PriorityQueue()
    frontier.put((0, start, [start]))  # (priority, current_node, path)
    visited = set()
    visited_order = []

    while not frontier.empty():
        _, current, path = frontier.get()
        if current in visited:
            continue
        visited.add(current)
        visited_order.append(current)

        if current in grid.goals:
            print_path_info(current, path, visited_order)
            return path, visited_order

        for neighbor in grid.get_neighbors(current):
            if neighbor not in visited:
                priority = heuristic(neighbor, list(grid.goals)[0])  # Change heuristic as appropriate
                frontier.put((priority, neighbor, path + [neighbor]))

    print("No goal is reachable.")
    return None, visited_order
```

Key Points: Best First Search's implementation is geared towards scenarios where an accurate heuristic can significantly direct the search making it faster and more efficient.

- Differences in Approach

The primary difference between the uninformed and informed search strategies lies in their use of heuristics. Informed searches such as A*, GBFS, and Best First Search use heuristics to make educated guesses about which paths to follow, potentially reducing the search space and time. Uninformed searches, by contrast, rely solely on the structure of the search tree, which can be less efficient but does not require any prior knowledge about the goal's location relative to any node.

Testing

This section details the systematic testing of tree-based search algorithms used in the robot navigation simulation. Tests were conducted across various grid configurations, designed to evaluate each algorithm's effectiveness and efficiency under different scenarios. These grids were meticulously designed to evaluate the algorithms' path optimality, computational efficiency, and ability to achieve completeness in search, reflecting both common and rare real-world scenarios. The insights gleaned from these tests are intended to guide further optimization of pathfinding strategies in AI applications, particularly in robotic navigation.

Evaluation Metrics

Performance was quantitatively assessed using several key metrics:

Path Length and Optimality: Measures the efficiency and effectiveness of the path chosen by the algorithm.

Computational Time and Resource Usage: Evaluates the algorithm's speed and the amount of memory resources consumed during the search process.

Success Rate: The frequency with which each algorithm successfully finds a path to the goal.

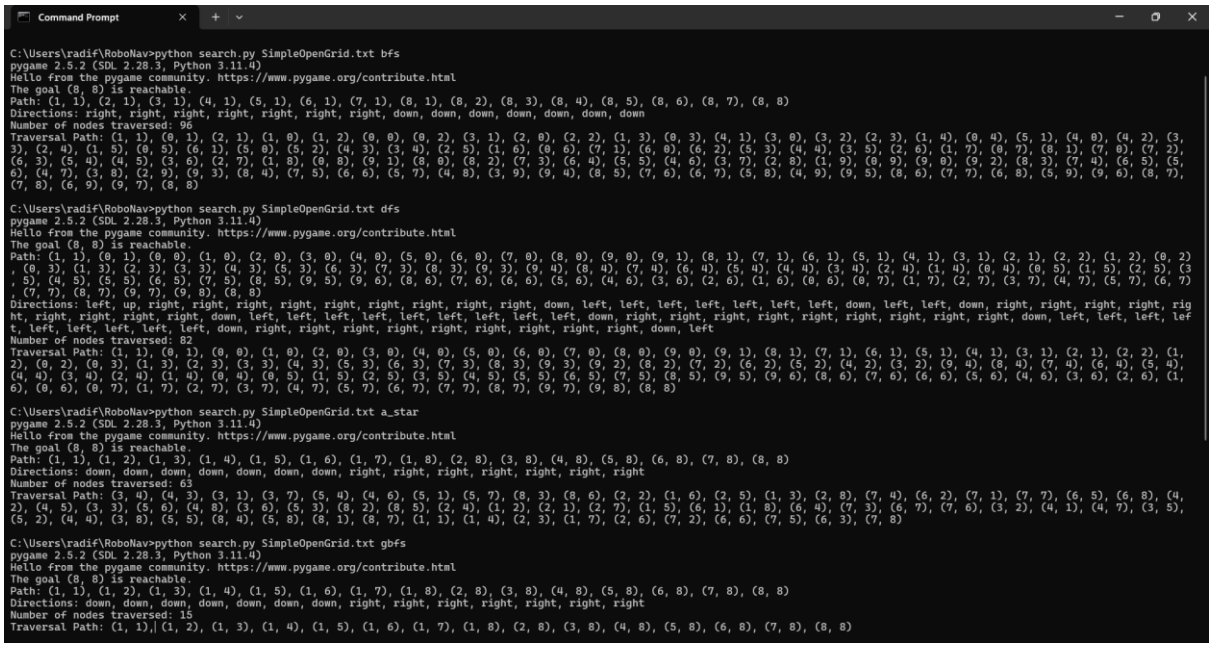
1. (Simple Open Grid)

- This test confirms the efficiency of basic search algorithms in an unobstructed environment, demonstrating optimal pathfinding with minimal computational resources required.

a. Input File:

[10,10]
(1,1)
(8,8)

b. Output



2. (No Solution)

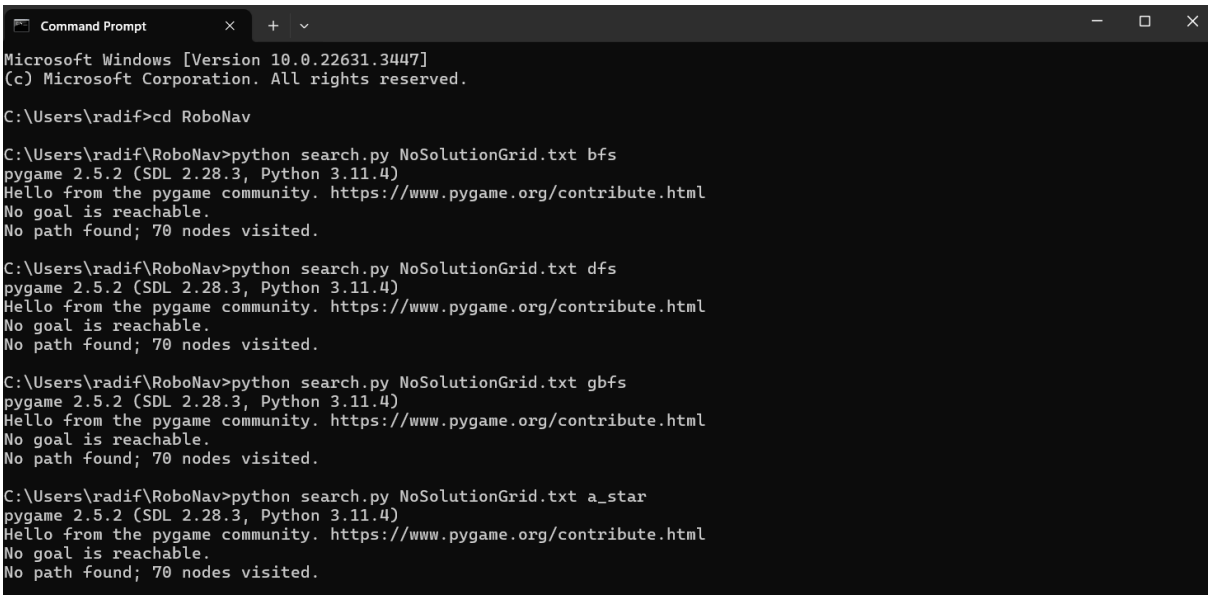
- This scenario validates the algorithm's capability to recognize and report unsolvable conditions effectively, indicating robust error handling and completeness in search strategy.

a. Input File:

[10,10]

(1,1)
(8,8)
(0,7,10,1)

b. Output File:



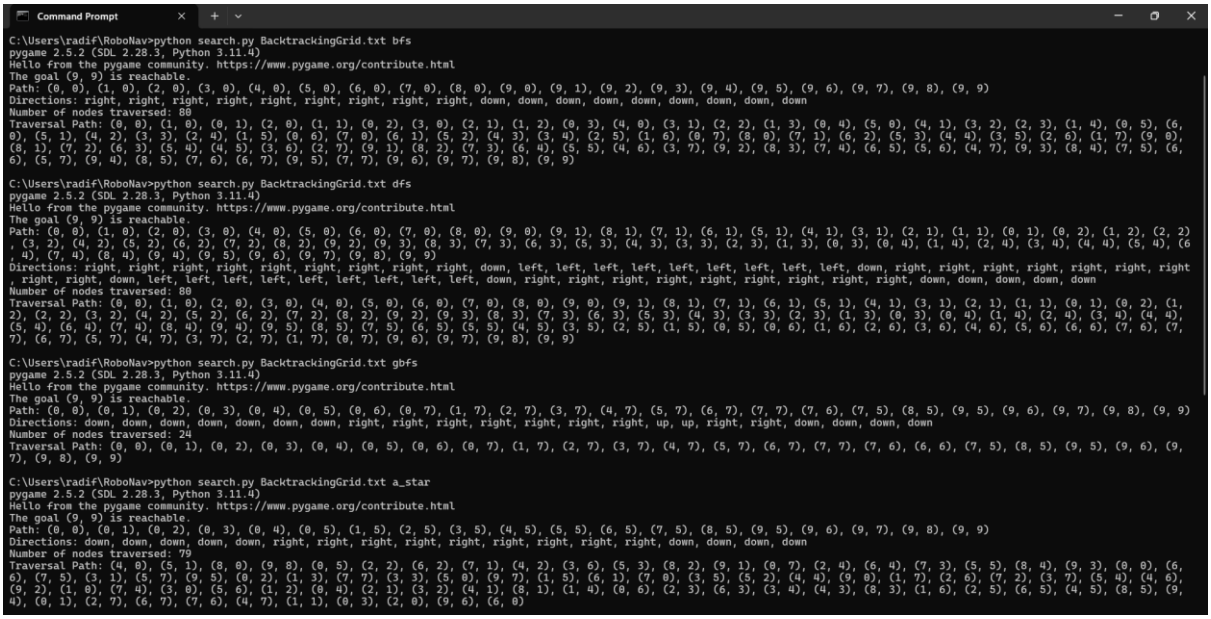
3. (Path Require Backtracking)

- The necessity for backtracking in this test showcases the algorithm's adaptability to dynamically changing paths and its ability to reconsider previous decisions when faced with dead ends

a. Input File:

[10,10]
(0,0)
(9,9)
(0,8,9,1)
(8,6,1,2)

b. Output File:



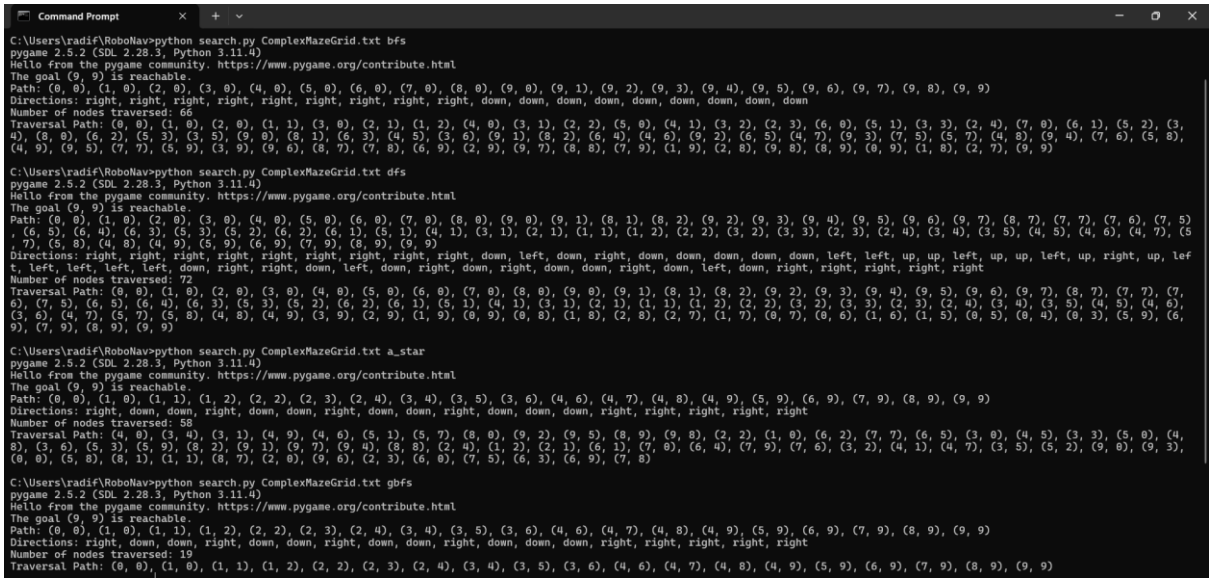
4. (Complex Maze Grid)

- This complex setup tests the heuristic-driven algorithms' efficiency, focusing on their ability to navigate through multiple obstacles strategically while minimizing the path length and computation time.

a. Input File:

[10,10]
(0,0)
(9,9)
(0,1,1,2)
(1,3,1,2)
(2,5,1,2)
(3,7,1,2)
(4,2,1,3)
(5,4,1,3)
(6,6,1,3)
(7,1,1,4)
(8,3,1,4)

b. Output File:



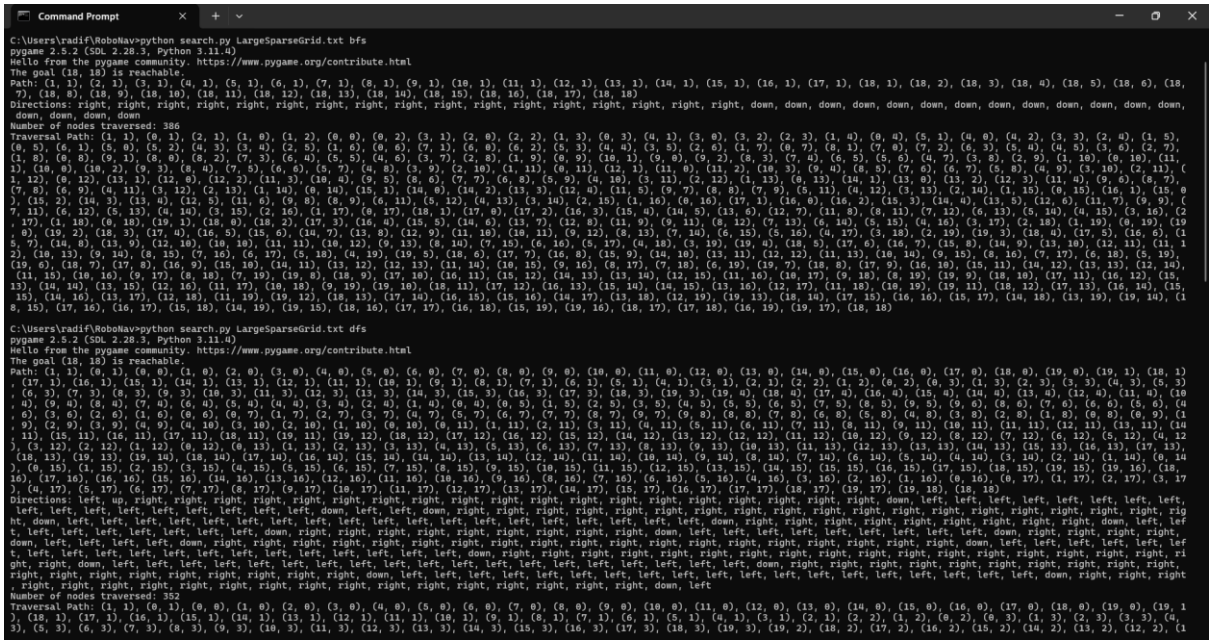
5. (Large Sparse Grid)

- The large grid size tests the scalability of the algorithms, assessing their performance in terms of speed and memory usage over extensive search areas with sparse obstacles.

a. Input File:

[20,20]
(1,1)
(18,18)
(10,5,1,5)
(5,10,5,1)

b. Output File:



6. (Performance Test)

- This performance-oriented test evaluates the algorithms under stress conditions with high density and extensive barriers, focusing on computational efficiency and resource management.

a. Input File:

[20,20]
(0,0)
(19,19)
(3,3,1,15)
(5,0,1,18)
(10,2,10,1)

b. Output File:

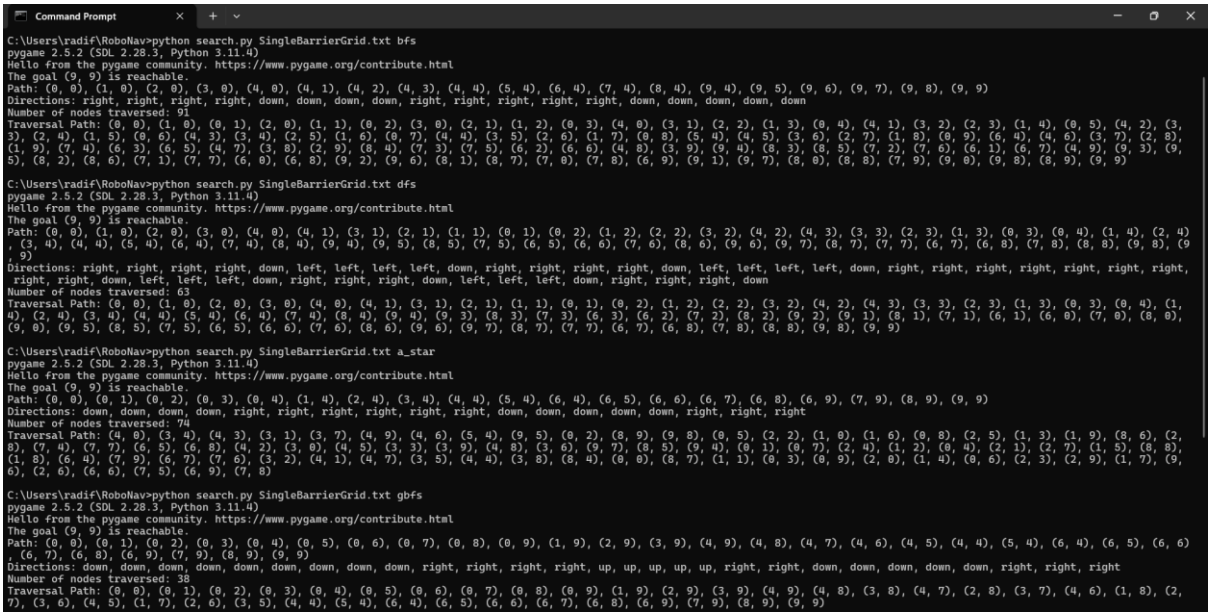
7. (Single Wall Barrier Grid)

- This test examines the algorithm's effectiveness in handling simple obstacles that require direct path adjustments, emphasizing the path optimality and algorithmic precision.

a. Input File:

[10,10]
(0,0)
(9,9)
(5,0,1,4)
(5,5,1,5)

b. Output File:



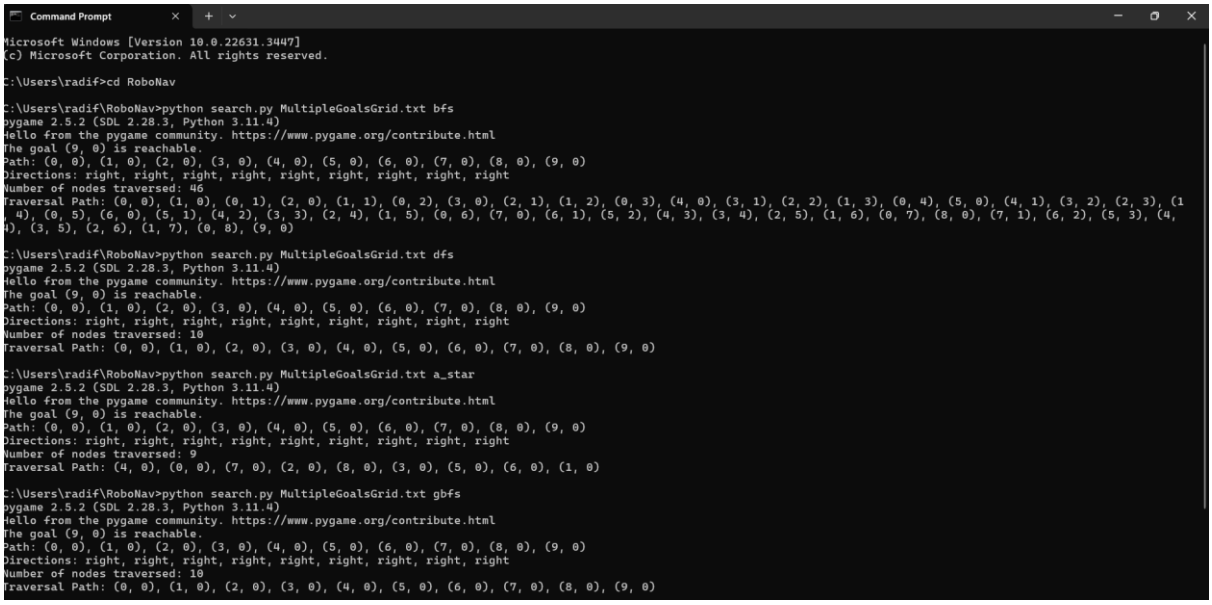
8. (Multiple Goals Grid)

- Testing with multiple goals assesses the algorithms' capability to reconfigure paths dynamically towards multiple targets, illustrating adaptability and efficiency in goal-oriented navigation.

a. Input File:

[10,10]
(0,0)
(9,0) | (0,9) | (9,9)

b. Output File:



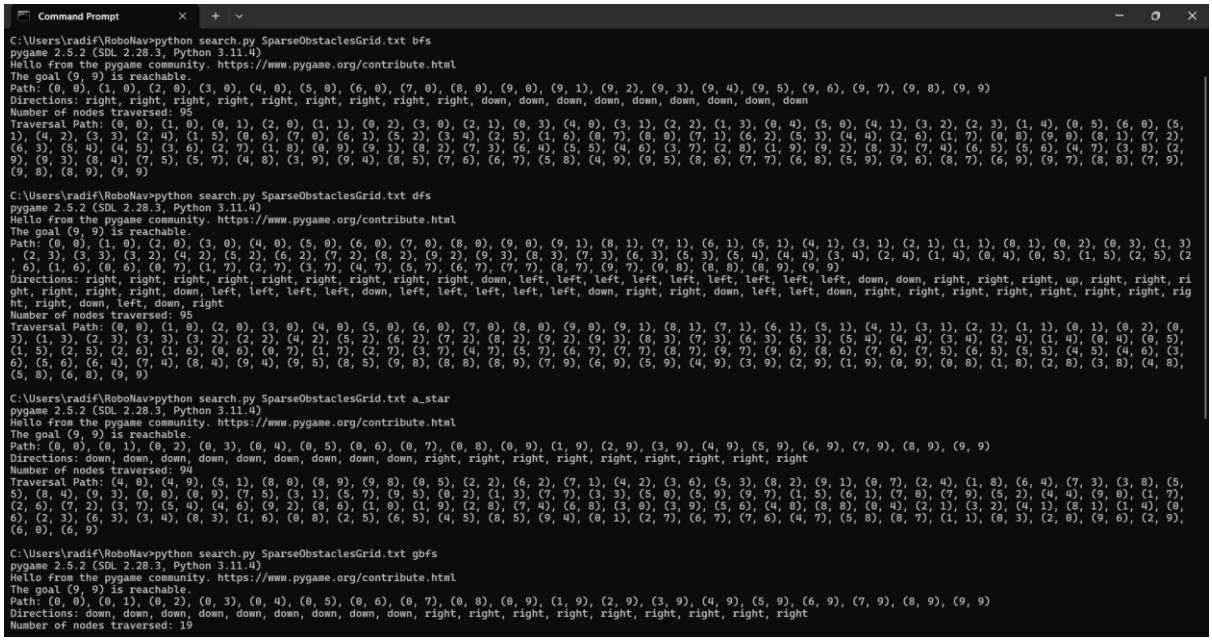
9. (Sparse Obstacles Grid)

- The presence of random sparse obstacles tests the robustness of pathfinding algorithms in maintaining path optimality and computational efficiency under unpredictably changing environments.

a. Input File:

[10,10]
(0,0)
(9,9)
(1,2,1,1)
(3,5,1,1)
(7,8,1,1)
(4,3,1,1)
(6,6,1,1)

b. Output File:



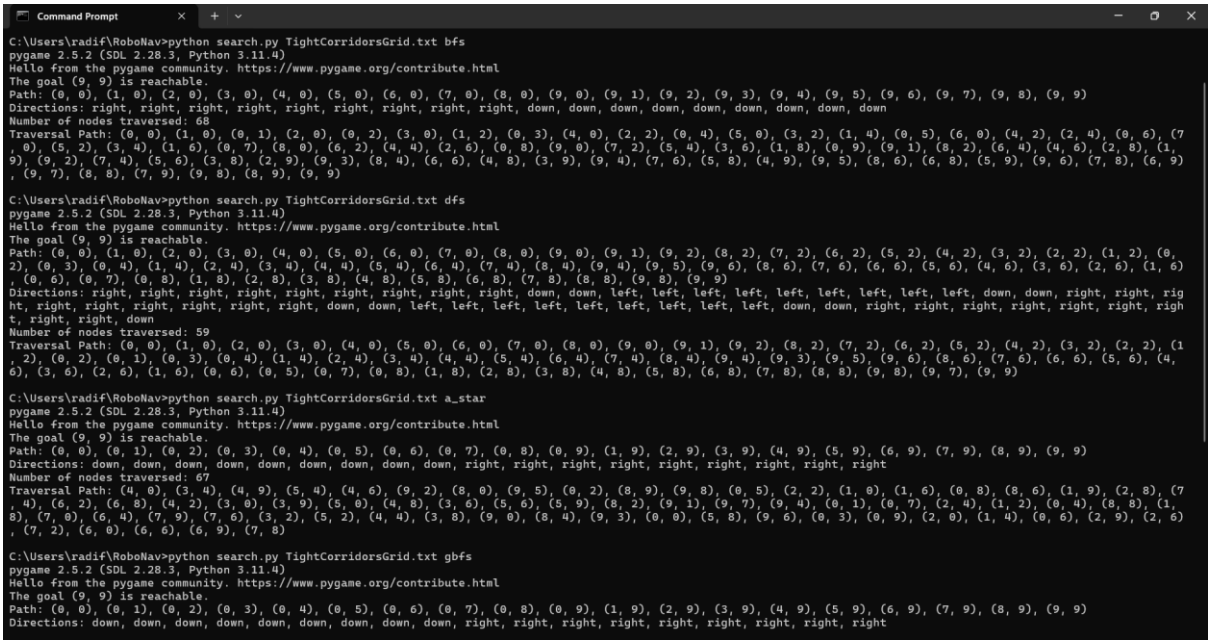
10.(Tight Corridors Grid)

- This scenario challenges the algorithms with narrow passages, testing their precision and efficiency in navigating tight spaces without compromising on the shortest possible path.

a. Input File:

[10,10]
(0,0)
(9,9)
(1,1,8,1)
(1,3,8,1)
(1,5,8,1)
(1,7,8,1)

b. Output File:



Features

Search Algorithms: The application supports multiple search algorithms including Breadth-First Search (BFS), Depth-First Search (DFS), A* Search, Greedy Best-First Search (GBFS), Iterative Deepening Search (IDS), and Best First Search. Each algorithm is integrated to work with a grid-based navigation system.

Heuristic Functions: Implemented Manhattan distance for A*, facilitating heuristic-driven pathfinding.

Dynamic Path Visualization: Using Pygame, the pathfinding process is dynamically visualized on the grid, showing the step-by-step navigation of the agent from the start to the goal node.

Grid Parsing: The program parses grid environments from a text file (RobotNav-test.txt), correctly identifying grid size, start points, goal points, and obstacles.

Interactive GUI: The GUI, built with Pygame, allows for interactive visualization of the pathfinding process, enhancing user engagement and understanding.

Direction Tracking: The application can compute and display the directions taken from the start to the goal, enhancing debug and demonstration capabilities.

- Missing Features

Weighted Paths: The current implementation does not support weighted paths where different terrain or paths might have different traversal costs. All moves have a uniform cost.

Multi-Agent Navigation: The system is currently designed for a single agent. Multi-agent navigation scenarios are not supported.

- Known Bugs

Performance Issues: On larger grids or with complex configurations, some algorithms (particularly DFS and IDS) might exhibit high memory usage and slower performance.

Limited Error Handling: There is limited error handling for malformed input files or incorrect format entries in the grid configuration file.

- Additional Information

Using the GUI: To use the GUI version of the program, run the search.py script with Python, specifying the grid configuration file and the desired search method as command-line arguments. For example:

python search.py RobotNav-test.txt bfs

This will launch the Pygame window and start the BFS algorithm on the provided grid.

- Running Without Pygame

If Pygame is not installed on system, the program is still fully functional but will revert to a command-line mode. In this mode, the user won't see the graphical interface, but the program will still perform the search and output the results to the command line. When Pygame is not detected, the program will automatically display the message:

"Pygame is not installed, running in command-line mode only."

Even without the visual GUI, the path found by the search algorithm, if any, will be printed to the console. This includes the sequence of moves and the total number of steps taken, allowing you to understand the algorithm's behaviour and verify its effectiveness even without the graphical display.

- Particular Implementation Details

Heuristic Flexibility: While the Manhattan is implemented, the system is designed to easily incorporate additional heuristic functions if needed.

Modularity: The code is modular, allowing for easy extension or modification of search algorithms, heuristics, and grid parsing logic.

Research

Enhancement of User Interface and Visualization in Pathfinding Simulations

To enhance the interactivity and educational value of the robot navigation simulation, a graphical user interface (GUI) was developed using the Pygame library. This GUI is not merely an add-on but a core component of the simulation that illustrates how each algorithm approaches the task of finding a solution within a grid environment. This graphical visualization provides a step-by-step demonstration of the algorithms' operations, making it an indispensable tool for both understanding and debugging the pathfinding processes.

- ## Initialization and Configuration

Before any visualization occurs, the Pygame environment is set up. This setup includes defining colors for various elements within the grid, setting the cell size for visual representation, and establishing the screen dimensions based on the grid size. These parameters are crucial as they directly impact the clarity and readability of the visualization:

Colors: Distinct colors are used to represent different types of elements on the grid. For instance:

Black (Walls): Clearly marks barriers that the agent cannot pass through.

White (Empty Cells): Denotes traversable space.

Blue (Visited Nodes): Indicates areas that have been explored by the search algorithm, providing a visual history of the algorithm's path.

Red (Goals): Highlights the target locations the agent aims to reach.

Green (Agent): Shows the agent's current location, making it easy to track movement.

Cell Size and Padding: These determine the size of each grid cell on the screen and the space between them, ensuring that the grid is neatly presented and that individual elements are distinguishable.

- ## Dynamic Drawing Functions

draw_initial_grid():

This function is responsible for rendering the static elements of the grid, such as the walls and empty cells. It is called initially to create the base layout of the grid and then repeatedly to refresh the background as the agent moves. This refreshing is crucial to provide a clear view of the agent's current state without residual images from previous positions.

draw_agent_and_goals():

After setting up the initial grid, this function overlays dynamic elements, such as the agent and goal positions. It updates the grid each time the agent moves to a new cell, thereby providing real-time feedback on the agent's progress toward the goal. This function is central to visualizing the pathfinding process as it allows observers to see both the end objectives and the agent's strategy to achieve them.

animate_agent():

Animation is a key aspect of making the simulation engaging and educational. This function moves the agent along the path determined by the search algorithm, visually depicting each step in sequence. The movement is shown in real-time, with a slight delay between steps (using sleep()) to ensure the motion is perceptible to users. This step-by-step animation helps in understanding how the search algorithm navigates the grid, tackles obstacles, and adjusts its route in response to barriers and already visited paths.

- **Visualization of Algorithm Dynamics**

The GUI provides a real-time visualization of the search tree dynamics, which is crucial for understanding how each algorithm explores the grid to find the optimal path:

Search Expansion Visualization: As the search algorithm operates, the GUI vividly marks nodes that are currently being explored, changing their color to blue. This dynamic update helps users visualize the algorithm's exploration pattern and understand its approach to navigating through the grid.

Real-Time Path Updates: The path determined by the algorithm is displayed, continuously updating as the algorithm progresses. This feature allows users to track the path's development in real time, from the starting point to the goal, providing immediate visual insight into the algorithm's effectiveness and strategy.

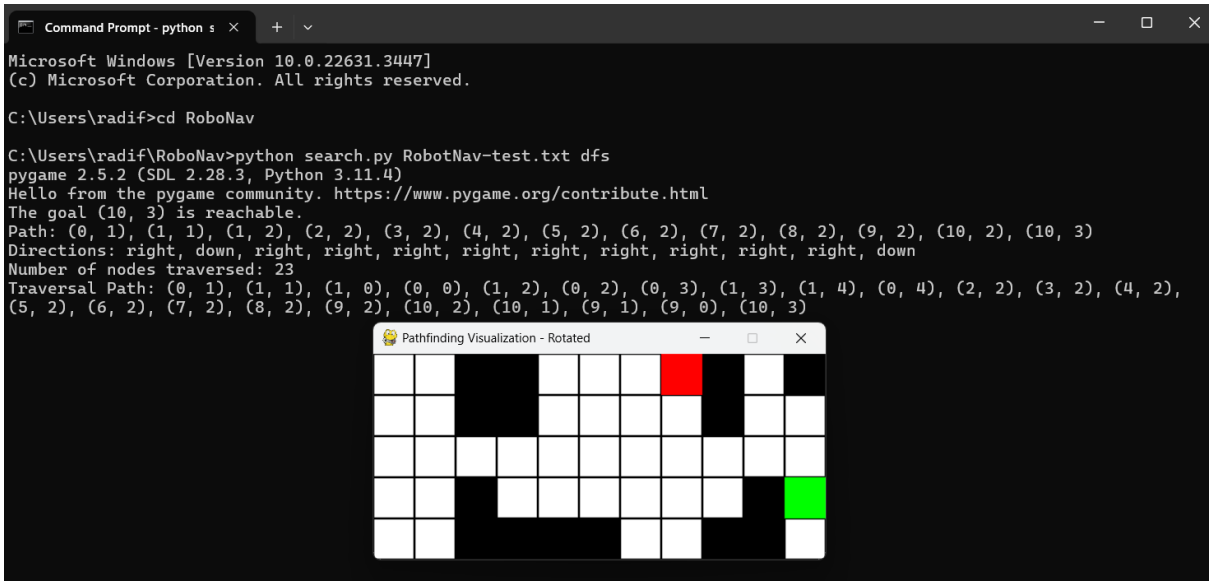
- **Educational and Debugging Utility**

Incorporating a GUI with dynamic visualization transforms the simulation from a simple computational model into a powerful educational tool that facilitates deeper learning and understanding of complex algorithms:

Interactive Learning: Students and practitioners can observe and analyse how different pathfinding algorithms navigate and solve the same grid, gaining insights into algorithmic efficiency and strategy.

Debugging and Optimization: The immediate visual feedback provided by the GUI allows for quick identification of any issues or inefficiencies in the algorithm's pathfinding logic, which is invaluable during the development and testing phases.

In summary, the GUI and visualizer not only enhance the usability of the pathfinding simulation but also deepen the user's engagement and understanding of algorithmic behaviours in a controlled environment. This interactive approach is instrumental in teaching the nuanced differences and applications of various search algorithms in a visually comprehensible manner.



Conclusion

This document has presented a comprehensive exploration of various tree-based search algorithms applied to a robot navigation problem within a grid-based simulation environment. Throughout this investigation, I have implemented, analysed, and optimized several search strategies, ranging from uninformed search algorithms like Depth-First Search (DFS) and Breadth-First Search (BFS) to informed search algorithms such as Greedy Best-First Search (GBFS) and A* Search, along with Iterative Deepening Search (IDS) and Best First Search.

My journey through the intricacies of these algorithms revealed the importance of selecting the appropriate search strategy based on the specific characteristics of the navigation task at hand. While uninformed search algorithms provided a robust foundation for understanding basic pathfinding mechanics, informed search algorithms demonstrated a superior ability to navigate complex environments efficiently, leveraging heuristic functions to guide the search process more effectively.

My experiments with making the algorithms better showed us that there's a lot of room to improve how these methods work, especially in changing environments that are more like real life. This could mean making the algorithms smarter by letting them learn from past experiences or by making them work together when there are many robots in the maze.

In conclusion, the findings from this document contribute valuable knowledge to the field of artificial intelligence and robotics, offering a deeper understanding of search algorithm capabilities and limitations. As I continue to push the boundaries of what is possible with algorithmic pathfinding, I move closer to realizing the full potential of AI in solving complex navigation problems in a myriad of applications.

References

Harabor, D. and Grastien, A. (2011). Online Graph Pruning for Pathfinding On Grid Maps. *Proceedings of the AAAI Conference on Artificial Intelligence*, 25(1), pp.1114–1119. doi:https://doi.org/10.1609/aaai.v25i1.7994.

Hart, Peter E., Nils J. Nilsson, and Bertram Raphael (n.d.). *'A Formal Basis for the Heuristic Determination of Minimum Cost Paths,' IEEE Transactions, 1968, inspired our exploration of heuristic functions.* [online] Available at: https://www.researchgate.net/publication/323223002_Evaluation_of_the_Performance_of_Heuristic_Algorithms_in_an_Intersection_Scenario.

Norvig, P. (n.d.). *Book Reviews: Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp | PDF | Parsing | Artificial Intelligence*. [online] Scribd. Available at: https://www.scribd.com/document/433428539/Paradigms-of-AI-Programming-Book-Review

Rusell, Stuart, J. and Peter, N. (n.d.). *Artificial Intelligence A Modern Approach Third Edition*. [online] Available at: https://people.engr.tamu.edu/guni/csce421/files/AI_Russell_Norvig.pdf.