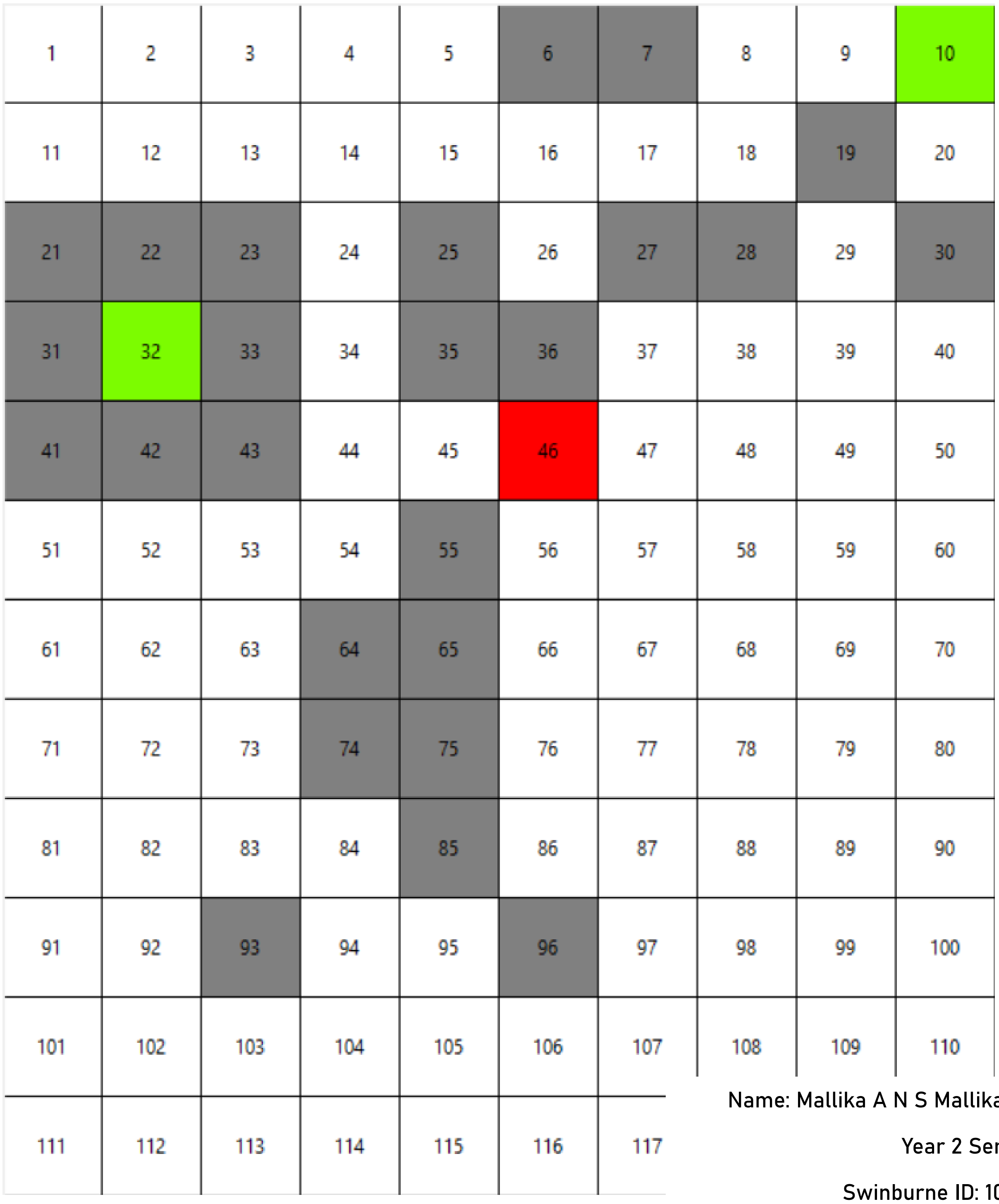
Tree Based Search – Assignment 1



Name: Mallika A N S Mallikarachchi

Year 2 Semester 1

Swinburne ID: 104756611

Email: 104756611@student.swin.edu.au

Contact Number: +61 401 876 063

Contents

Instructions

General Setup

Python Installation:

- Install Python 3.8 or higher from python.org
- Verify installation by running => *python –version in the command prompt or terminal*

Required Libraries:

- Tkinter for GUI Support (Will be installed by default with Python)
- Additional Packages might be required depending of the version. If any packages are missing install the packages using => *pip install tkinter*

Running the Program

Using the Command Line

- Navigate to the program dictionary.
- Open cmd (Command Prompt) for the directory
- Run the following code in terminal => *python run.py*
  - *Please Note that My Code doesn't work in the format of python run.py <input_file> <search_method>*
  - I was Not able to get this feature working. This is a missing feature in the code.
  - The test file should be manually updated in the *run.py* in order for the program to work properly
  - Update the *file_path* variable in run.py before starting the program

Using an IDE

- Update the run.py with correct file path
- Run the program

Interacting with the GUI

- The GUI Window will display the maze and control buttons for each algorithm.
- To Execute a search, simply click on the corresponding button

Important

- Do not run multiple algorithms at once. Make sure to wait until One Algorithm is done executing.
- If the Program Shows any Bugs (Visualizations) simply re run the Program.
- Make Sure that the File format adheres the standard notation given below

Text File Format

```
 [5,11] // The grid has 5 rows and 11 columns.
(0,1) // initial state of the agent – coordinates of the red cell
(7,0) | (10,3) // goal states for the agent – coordinates of the green cells
(2,0,2,2) // the square wall has the leftmost top corner occupies cell (2,0) and is 2 cells wide and 2 cell high
(8,0,1,2)
```

Introduction

Overview

This report is written on behalf of the Python program that I have created to demonstrate various Artificial Intelligence Search Algorithms which are basic Tree Search Algorithms. The Program includes a robot navigation problem with a simulated environment. My Primary goal is to Visualize and understand the Different search algorithms and to test their performances.

The Robot Navigation Problem

Robot navigation involves finding a path from a start point (S) to a goal point (G) or multiple Goal points within an environment that may contain obstacles. The environment is represented as a grid where each cell can either be passable or blocked (Wall). The objective for the robot is to find the shortest possible route from the start to the goal without traversing any obstacles. (Russell & Norvig, 1995)

Basic Graph and Tree Concepts

Graphs:

- A Collection of nodes and edges connecting pairs of nodes. In the context of AI Search algorithms, nodes represent possible states or locations in the environment, and edges represent the possible actions or moves from one state to another (Russell & Norvig, 1995)

Trees:

- A Special form of graph where any two vertices are connected by exactly one path. In Tree Structures use for search algorithms, each node represent a state. And connections between nodes represent the transition from one state to another due to an action. (Russell & Norvig, 1995)

Paths

- A Path in graph theory is a sequence of edges which connects a sequence of vertices. For robot navigation, a path would be the sequence of moves leading from the start node to the goal node without any cycles

Nodes & Edges

- In the maze environment, each cell is treated as a node. An edge exists between two nodes if a robot can move directly from one cell to the other without encountering an obstacle.

Costs

- Movement from one node to another can have associated costs, which can vary depending on the algorithm used. For uniform cost search, all moves might have the same cost, whereas, for algorithms like A*, the costs might vary to reflect an estimated distance to the goal.

Heuristics

- A heuristic is a function that estimates the cost from a node to the goal. Used predominantly in algorithms like A* and Greedy Best First Search, heuristics help prioritize node exploration based on proximity to the goal.

Search Algorithms

Overview

Algorithm Descriptions & Performances

Search algorithms are at the heart of solving the robot navigation problem. These algorithms can be categorized based on their search mechanism and underlying principles such as uninformed search, which operates without any knowledge of the distance to the goal, and informed search, which uses heuristics to guess the closest path to the goal. (CS50, 2023)

Breadth First Search

BFS explores the graph level by level and is implemented using a queue. It expands all nodes at a given depth before moving on to nodes at the next depth level, ensuring the shortest path in an unweighted graph. It is particularly useful for our robot navigation as it guarantees finding the shortest path when all moves cost the same.

Depth Frist Search

DFS uses a stack to explore as far as possible along each branch before backtracking. This method can be more memory efficient than BFS but does not guarantee the shortest path, making it less ideal for pathfinding in mazes but useful for exploring all possible paths.

Heuristics

The Heuristic function that I am using in this python program is called the Manhattan Distance Heuristic. Which is basically the distance between 2 points. The Manhattan Distance is identified by the following equation where $x1, y1$ & $x2, y2$ are coordinates of a graph. (Russell & Norvig, 1995)

$$Manhattan\ Distance = (|x1 - x2| + |y2 - y1|)$$

A* Search

A* search algorithm uses both the actual cost from the start and a heuristic estimated cost to the goal to explore the most promising paths first. It is highly effective in pathfinding in grid-based games or maps, as it can significantly reduce the number of explored nodes. This algorithm uses a combination of the actual cost of reaching a node from the start node (known as $g(n)$) and a heuristic estimate of the cost from the node to the goal node (known as $h(n)$). The total estimated cost $f(n)$ for a node is defined as (GeeksforGeeks, 2024c):

$$f(n) = g(n) + h(n)$$

Greedy Best First Search

GBFS expands the nodes nearest to the goal, as estimated by a heuristic, potentially leading to faster find times but at the risk of not finding the shortest path. The equation used in Greedy Best-First Search can be described as follows:

$$f(n) = h(n)$$

Where f(n) is the evaluation function for a node n & h(n) is the heuristic function that estimates the cost from node n to the goal

Iterative Deepening Depth First Search (IDDFS)

IDDFS (Iterative Deepening Depth-First Search) is a graph/tree search algorithm that combines the benefits of depth-first search (DFS) and breadth-first search (BFS) strategies. It aims to find a solution in a search space where the depth of the solution is unknown, or the space is too large to store in memory entirely. IDDFS is particularly useful when memory constraints are a concern, and it ensures that the solution found is at minimal depth. (GfG, 2023)

Iterative Deepening A* (IDA*)

IDA* (Iterative Deepening A*) is a search algorithm that combines elements of both depth-first search (DFS) and the A* algorithm. It is designed to find the shortest path to a goal node in a graph or tree, while also addressing the memory limitations associated with traditional A* search by using an iterative deepening approach similar to Iterative Deepening Depth-First Search (IDDFS). IDA* is particularly useful for solving problems in which the search space is large and memory usage needs to be optimized. (GfG, 2023b)

Comparisons

Algorithm	Strategy	Completeness	Optimality	Space Complexity	Time Complexity (Worst Case)
BFS	Exploration level by level	Yes	Yes (for unweighted graphs)	$O(b^d)$, where b is the branching factor and d is the depth of the shallowest goal	$O(b^d)$
DFS	Deep exploration of one branch	No (can get stuck in infinite loops or cycles)	No (may not find the optimal solution)	$O(b×m)$, where b is the branching factor and m is the maximum depth of the tree	$O(b^m)$, where b is the branching factor and m is the maximum depth
IDDFS	Iterative deepening of DFS	Yes (if solution exists within finite depth)	Yes (finds optimal solution for uniform edge costs)	$O(b×d)$, where b is the branching factor and d is the maximum depth limit.	$O(b^d)$, where b is the branching factor and d is the maximum depth limit.
GBFS	Priority queue based on heuristic.	No (can get stuck in local optima)	No (may not find the optimal solution)	$O(b^m)$, where b is the branching factor and m is the maximum depth.	Depends on the quality of the heuristic.
A*	Priority queue based on $f(n)=g(n)+h(n)$	Yes (if $h(n)$ is admissible)	Yes (finds optimal solution for uniform edge costs)	$O(b^d)$ in worst case, where b is the branching factor and d is the depth of the shallowest goal	Exponential in worst case, but generally much faster with good heuristic
IDA*	Iterative deepening of A*	Yes (if solution exists within finite depth)	Yes (finds optimal solution for uniform edge costs)	$O(b×d)$ in worst case, where b is the branching factor and d is the depth of the shallowest goal	Exponential in worst case, but generally more efficient than A* due to iterative deepening

Concluding Thoughts

Each search algorithm has its strengths and weaknesses, and their effectiveness can vary greatly depending on the specific requirements and constraints of the problem being addressed. For example:

- **BFS** and **IDDFS** are preferred when path accuracy is critical.
- **DFS and IDDFS & IDA*** might be suitable for exhaustive searches where memory is an issue.
- **A*** is ideal for complex searches where a balance between efficiency and accuracy is needed.
- **GBFS** offers quick solutions where exact precision is less crucial.

Understanding these characteristics allows developers to select the most appropriate algorithm that best meets the needs of their application, optimizing both performance and resource utilization. This nuanced approach to algorithm selection is vital for developing sophisticated AI systems that are both efficient and effective.

Implementation

The implementation of the search algorithms for the robot navigation problem involves several key Python modules, which are organized as follows:

- **run.py:** This is the main entry script that orchestrates the flow of the entire program. It initializes the execution by calling functions from other modules to set up the environment and start the search process.
- **AI_algorithms.py:** Contains the core logic for the various search algorithms (BFS, DFS, A*, GBFS, UCS, IDDFS, IDA*). Each algorithm is implemented as a separate function within this module.
- **Read_file.py:** Responsible for setting up the maze environment from an input file. It reads and processes the file to extract maze dimensions, wall positions, start and goal locations.
- **initialize_maze.py:** This module sets up the graphical display for the maze and updates the grid as the search algorithms run.

Detailed Implementation

Breadth-First Search (BFS)

Initialization:

- A queue is used to manage the nodes that need to be explored next.
- BFS starts from the given starting node and explores all neighboring nodes at the present depth prior to moving on to nodes at the next depth level. (GeeksforGeeks, 2024)

Process:

- Nodes are removed from the queue and their adjacent nodes are added to the queue.
- All nodes at a given depth are explored before any nodes at the next depth level.
- This ensures that the shortest path in an unweighted graph is found first.
- The path can be reconstructed using parent pointers which track how each node is reached.

Depth-First Search (DFS)

Initialization:

- A stack is used to keep track of which nodes to explore next.

- DFS explores as far down a branch as possible before backtracking to explore other branches. (GeeksforGeeks, 2024a)

Process:

- Nodes are popped from the stack, and their neighbors are pushed onto the stack if they haven't been visited.
- This allows the algorithm to dive deep into a branch before backtracking to earlier nodes to explore other branches.
- This method does not necessarily find the shortest path, but it will explore all possible paths.
- Similar to BFS, path reconstruction can utilize parent pointers.

Greedy Best-First Search (GBFS)

Initialization:

- A priority queue is used, where nodes are prioritized based on a heuristic that estimates the cost from the node to the goal.
- This algorithm is greedy and chooses whichever path seems best at the moment based on the given heuristic without considering the overall path cost. (GeeksforGeeks, 2024a)

Process:

- Nodes are expanded based on their heuristic cost to the goal, not including the cost come from the start.
- This can lead to faster solutions but does not guarantee the shortest path.
- Path reconstruction is done by backtracking from the goal node using a parent map.

Uniform Cost Search (UCS)

Initialization:

- Similar to BFS but uses a priority queue instead of a regular queue.
- Each node in the queue has a cost associated with it, which represents the exact cost of reaching that node from the start.

Process:

- Nodes are expanded in order of their path cost from the start node.
- Ensures that the first time a node is reached, it is via the cheapest path available.
- Guarantees finding the shortest path in a weighted graph.
- Path reconstruction is done using a parent map.

Iterative Deepening Depth-First Search (IDDFS)

Initialization:

- Combines the space-efficiency of DFS and the level-by-level exploration of BFS.
- Begins at the root and explores as deep as a specified depth limit.

Process:

- The depth limit starts at zero and increases with each iteration until the goal is found or all nodes are explored.

- Nodes are explored via DFS up to the depth limit, and any nodes beyond this limit are ignored during that iteration.
- This repeated deepening allows the algorithm to find the shortest path, unlike standard DFS.
- Path reconstruction utilizes parent pointers.

Iterative Deepening A (IDA*)

Initialization:

- Combines ideas from A* and IDDFS, using a heuristic to evaluate nodes and a depth limit that increases each iteration.
- Starts with a depth limit based on the heuristic estimate from the start node to the goal.

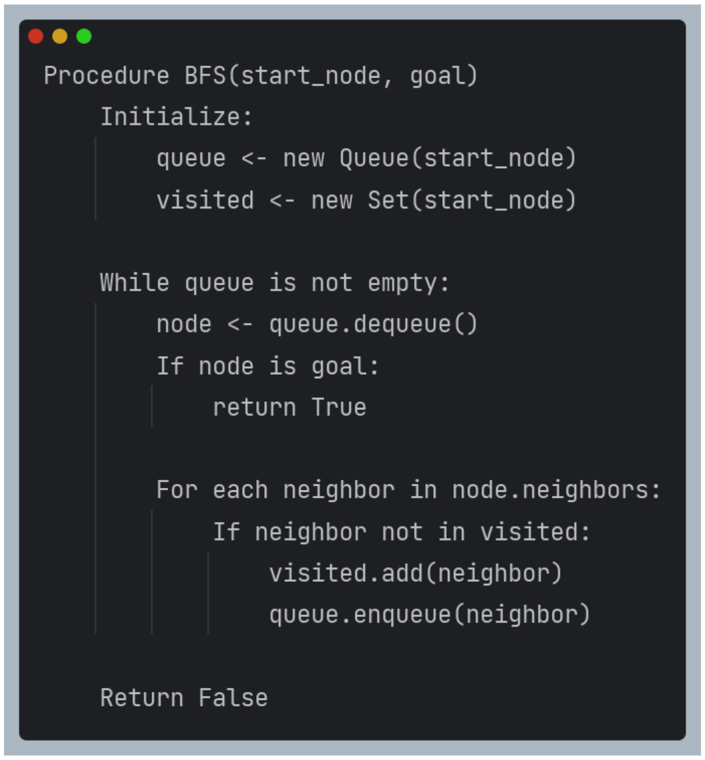
Process

- Nodes are explored using a depth-first manner, but expansion is limited to nodes where the estimated cost (f = g + h) is less than the current threshold.
- The threshold is increased based on the minimum cost of all nodes examined but not expanded in the previous iteration.
- Ensures that the algorithm does not miss the shortest path due to an overly aggressive heuristic.
- Path reconstruction is achieved through parent pointers, similar to A*.

Pseudocodes For All Algorithms

- Please Note that These Images were made by me for the pseudocodes that I implemented myself

BFS



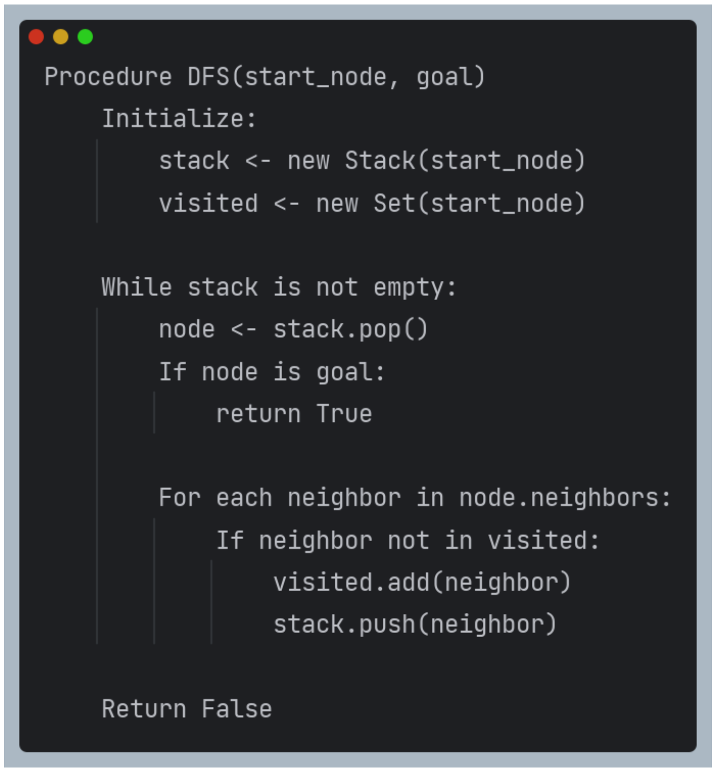
```
Procedure BFS(start_node, goal)
    Initialize:
        queue <- new Queue(start_node)
        visited <- new Set(start_node)

    While queue is not empty:
        node <- queue.dequeue()
        If node is goal:
            return True

        For each neighbor in node.neighbors:
            If neighbor not in visited:
                visited.add(neighbor)
                queue.enqueue(neighbor)

    Return False
```

DFS



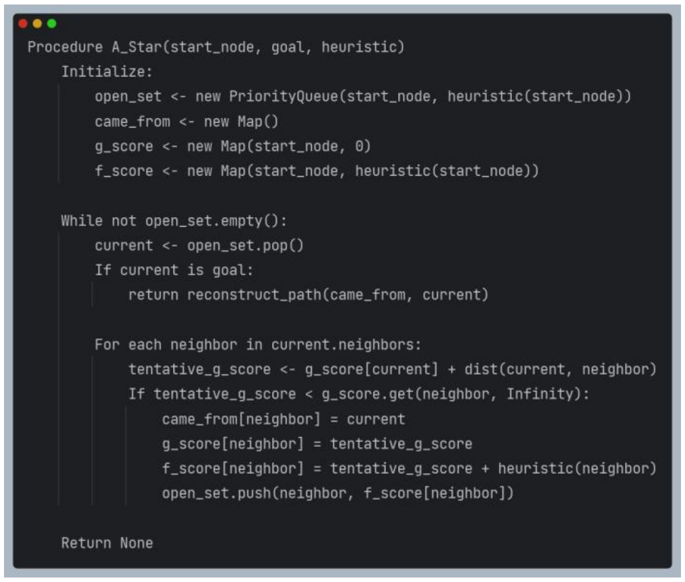
```
Procedure DFS(start_node, goal)
    Initialize:
        stack <- new Stack(start_node)
        visited <- new Set(start_node)

    While stack is not empty:
        node <- stack.pop()
        If node is goal:
            return True

        For each neighbor in node.neighbors:
            If neighbor not in visited:
                visited.add(neighbor)
                stack.push(neighbor)

    Return False
```

A*



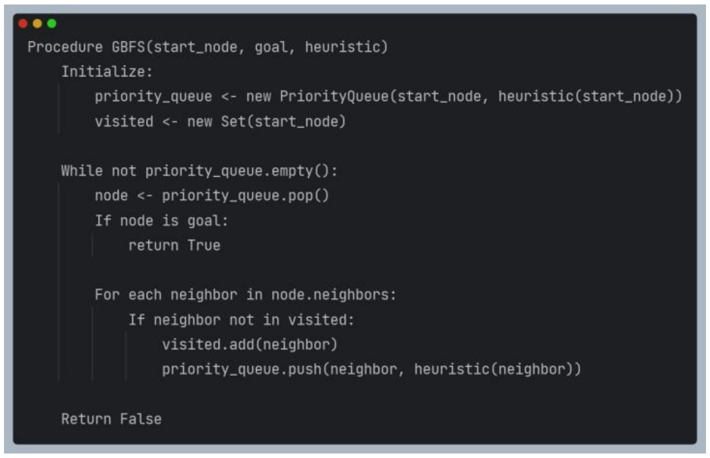
```
Procedure A_Star(start_node, goal, heuristic)
    Initialize:
        open_set <- new PriorityQueue(start_node, heuristic(start_node))
        came_from <- new Map()
        g_score <- new Map(start_node, 0)
        f_score <- new Map(start_node, heuristic(start_node))

    While not open_set.empty():
        current <- open_set.pop()
        If current is goal:
            return reconstruct_path(came_from, current)

        For each neighbor in current.neighbors:
            tentative_g_score <- g_score[current] + dist(current, neighbor)
            If tentative_g_score < g_score.get(neighbor, Infinity):
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g_score
                f_score[neighbor] = tentative_g_score + heuristic(neighbor)
                open_set.push(neighbor, f_score[neighbor])

    Return None
```

GBFS



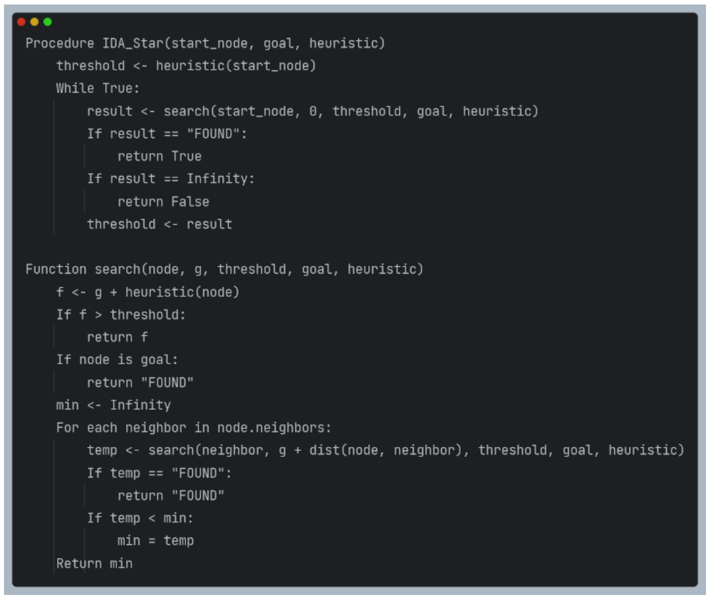
```
Procedure GBFS(start_node, goal, heuristic)
    Initialize:
        priority_queue <- new PriorityQueue(start_node, heuristic(start_node))
        visited <- new Set(start_node)

    While not priority_queue.empty():
        node <- priority_queue.pop()
        If node is goal:
            return True

        For each neighbor in node.neighbors:
            If neighbor not in visited:
                visited.add(neighbor)
                priority_queue.push(neighbor, heuristic(neighbor))

    Return False
```

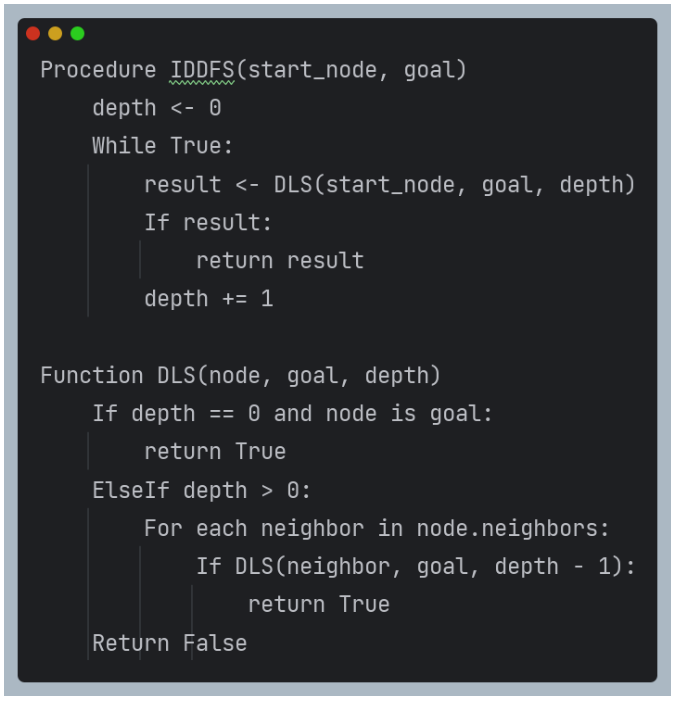
IDA*



```
Procedure IDA_Star(start_node, goal, heuristic)
    threshold <- heuristic(start_node)
    While True:
        result <- search(start_node, 0, threshold, goal, heuristic)
        If result == "FOUND":
            return True
        If result == Infinity:
            return False
        threshold <- result

Function search(node, g, threshold, goal, heuristic)
    f <- g + heuristic(node)
    If f > threshold:
        return f
    If node is goal:
        return "FOUND"
    min <- Infinity
    For each neighbor in node.neighbors:
        temp <- search(neighbor, g + dist(node, neighbor), threshold, goal, heuristic)
        If temp == "FOUND":
            return "FOUND"
        If temp < min:
            min = temp
    Return min
```

IDDFS



```
Procedure IDDFS(start_node, goal)
    depth <- 0
    While True:
        result <- DLS(start_node, goal, depth)
        If result:
            return result
        depth += 1

Function DLS(node, goal, depth)
    If depth == 0 and node is goal:
        return True
    ElseIf depth > 0:
        For each neighbor in node.neighbors:
            If DLS(neighbor, goal, depth - 1):
                return True
    Return False
```

Features/ Bugs/ Missing Features

Features

The program includes a robust set of features designed to showcase various search algorithms and enhance user experience:

- **Multiple Search Algorithms:** The application supports multiple algorithms including BFS, DFS, A*, Greedy BFS, Uniform Cost Search, Iterative Deepening DFS, and Iterative Deepening A*.

- **Graphical User Interface (GUI):** A user-friendly GUI implemented with Tkinter displays the maze and visualizes the algorithm's progress in real time.
- **Dynamic Visualization:** Nodes change color during execution to indicate their status (e.g., current frontier, visited, path), allowing users to visually track the algorithm's progress.

Known Bugs

While extensive efforts have been made to ensure the program runs smoothly, a few issues have been identified:

- **GUI Responsiveness:** Under certain conditions, the GUI may become unresponsive, particularly when rapidly switching between algorithms. This is suspected to be due to threading issues in the visualization code.
- **Memory Usage:** DFS can consume a significant amount of memory for large mazes, which may lead to performance degradation on systems with limited RAM.
- **Path Display Error:** The final path displayed by the BFS algorithm may not clear previous paths (Nodes already in the queue) completely, causing misleading visual overlaps. The Nodes in the Queue will also show animation, but the program has already been executed.

Missing Elements

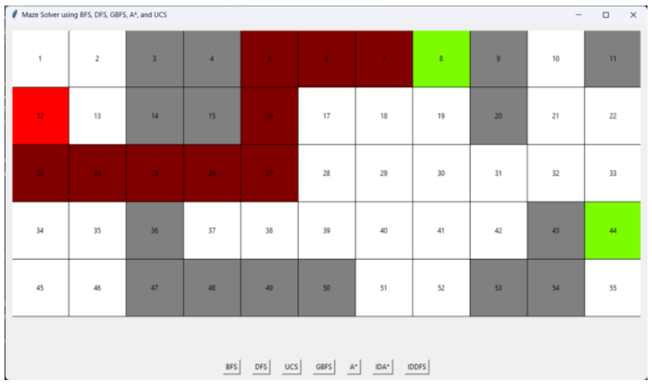
The following are features and functionalities that were not implemented in the current version of the program:

- **Command Line Interface for Batch Testing:** Originally planned, this feature was not implemented. The program was intended to support batch testing through a DOS command-line interface to facilitate automated testing scenarios, but due to time constraints, it was not completed.
- **GUI Doesn't have Interactive Elements:** The GUI Only contain the 6 Algorithms. I was not able to implement loading text files, generating maze text files, or other GUI Features.

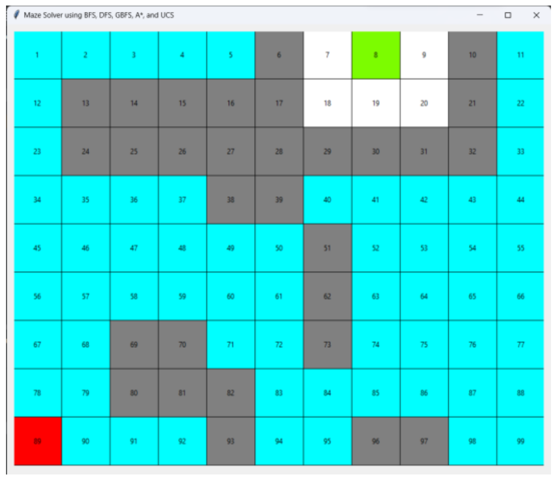
Testing (10 Sample Files)

- All the test files that I am using to test and verity the Maze algorithms are included in the ZIP file that I am uploading along with this report. Due to the page limitation I am not able to include all screenshots of the mazes visualizing all Algorithms. Below are some screenshots of the 10 Text Files that I generated.

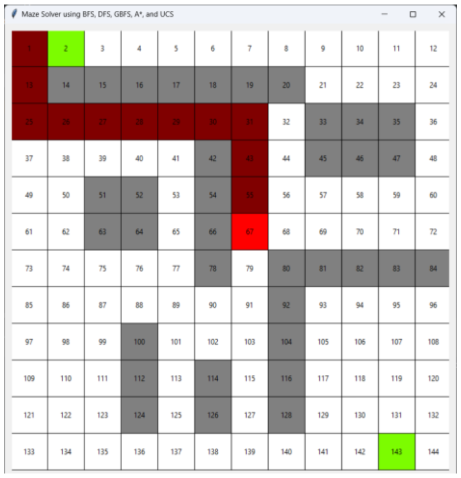
1.txt – IDA*



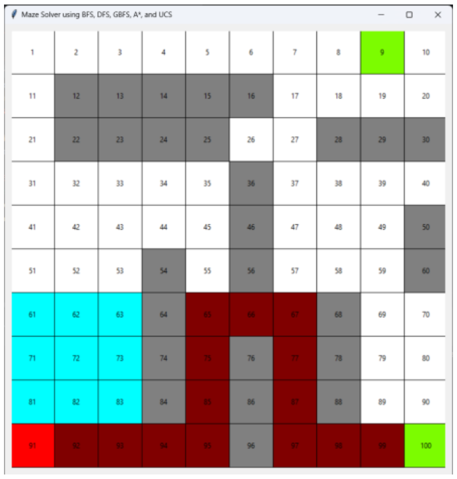
2.txt - BFS



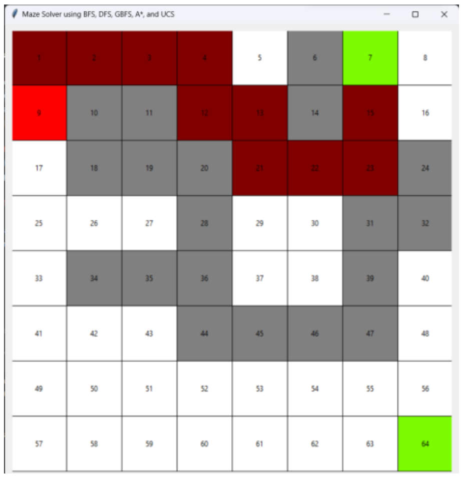
3.txt – GBFS



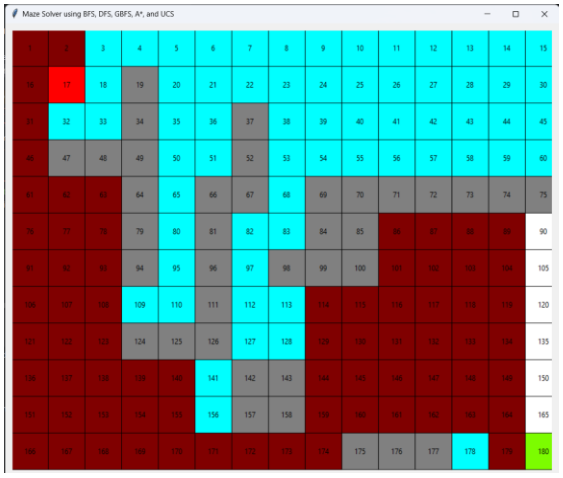
4.txt – A*



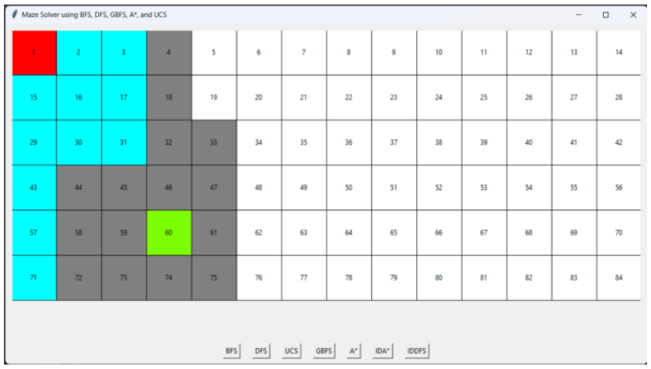
5.txt – IDDFS



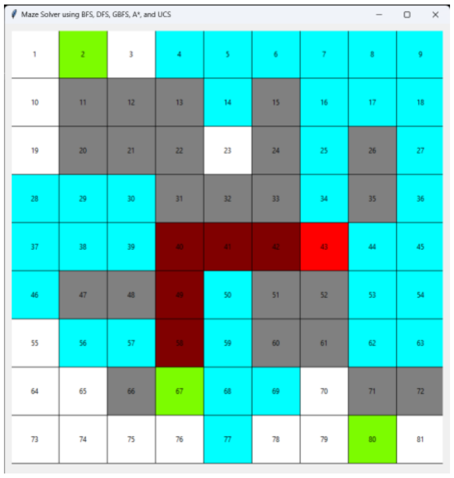
6.txt - DFS



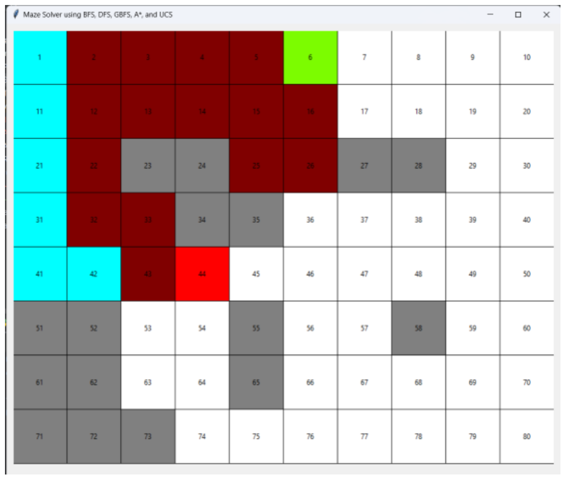
7.txt – A*



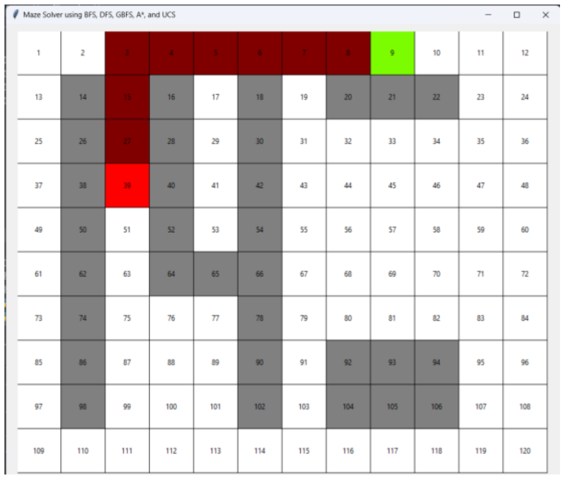
8.txt – BFS



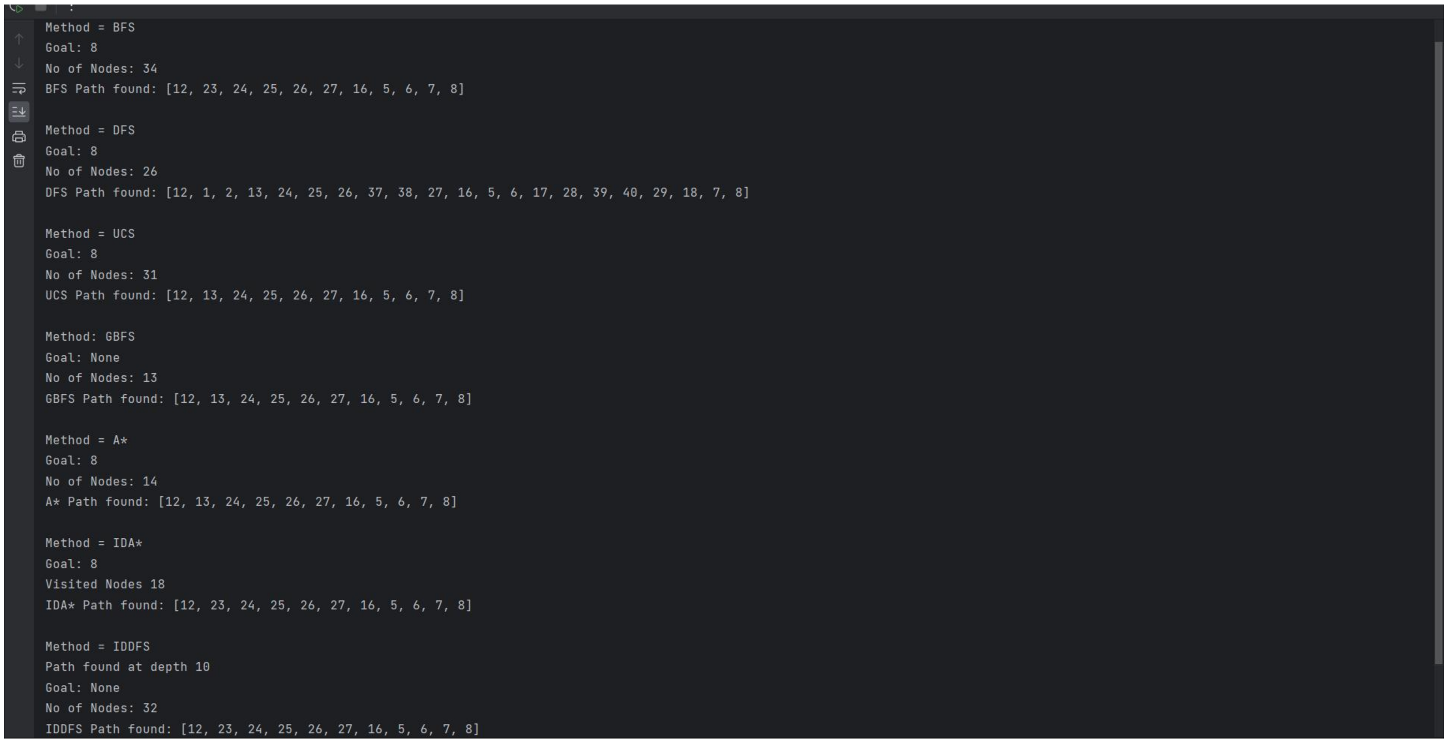
9.txt - DFS



10.txt – IDA*



Sample Output for all 6 Algorithms for 1.txt



```
Method = BFS
Goal: 8
No of Nodes: 34
BFS Path found: [12, 23, 24, 25, 26, 27, 16, 5, 6, 7, 8]

Method = DFS
Goal: 8
No of Nodes: 26
DFS Path found: [12, 1, 2, 13, 24, 25, 26, 37, 38, 27, 16, 5, 6, 17, 28, 39, 40, 29, 18, 7, 8]

Method = UCS
Goal: 8
No of Nodes: 31
UCS Path found: [12, 13, 24, 25, 26, 27, 16, 5, 6, 7, 8]

Method: GBFS
Goal: None
No of Nodes: 13
GBFS Path found: [12, 13, 24, 25, 26, 27, 16, 5, 6, 7, 8]

Method = A*
Goal: 8
No of Nodes: 14
A* Path found: [12, 13, 24, 25, 26, 27, 16, 5, 6, 7, 8]

Method = IDA*
Goal: 8
Visited Nodes 18
IDA* Path found: [12, 23, 24, 25, 26, 27, 16, 5, 6, 7, 8]

Method = IDDFS
Path found at depth 10
Goal: None
No of Nodes: 32
IDDFS Path found: [12, 23, 24, 25, 26, 27, 16, 5, 6, 7, 8]
```

Research

The primary research component of this project involved implementing a Graphical User Interface (GUI) using the Tkinter library in Python. This enhancement was aimed at providing a visual representation of the algorithms' operations, making the software more interactive and user-friendly, and facilitating easier demonstrations of the algorithms' functionality.

GUI Implementation Overview:

- **Purpose:** The GUI was designed to visually demonstrate how different search algorithms navigate through a maze, showing real-time updates as nodes are explored, marking the start and end points, and visually distinguishing the path found.

- **Functionality:** Users can select different algorithms to run and see the maze populated in real time with colors indicating visited nodes, the current frontier, and the final path. This setup helps in comparing the efficiency and pathfinding capabilities of each algorithm visually.
- **Impact:** The implementation of the GUI has significantly enhanced the educational value of the project by making the differences between algorithms clear and tangible, even to those without a technical background. It allows users to visually grasp complex concepts like pathfinding and algorithm efficiency.

Implementation

- While making the GUI I made 3 python dictionaries to keep track of the maze properties. Coordinates of the maze, rectangle ids, and rectangle objects were stored in these dictionaries
- An animate_path() & update_rectangle_color(rect_id, color) was made in order to visualize each and every algorithm as it happens
- Added a time.sleep(0.1) for the monitor to stop stressing when given a complicated maze.

Improvements

- **Memory Efficiency:**
    - Optimizing the existing code to reduce the memory footprint, especially for depth-first search, which could improve performance on larger mazes.

Conclusion

Reflections on Search Algorithms

Throughout this project, we explored various AI search algorithms aimed at solving the robot navigation problem in a maze-like environment. Each algorithm, from BFS and DFS to more sophisticated ones like A*, Greedy BFS, and Uniform Cost Search, has demonstrated unique strengths and limitations in terms of pathfinding efficiency, computational overhead, and practical usability.

- **BFS & GBFS** proved to be extremely reliable for finding the shortest path in scenarios where path cost is a critical factor. Their straightforward implementation and optimal pathfinding capability make them excellent choices for many practical applications.
- **DFS**, while not useful for finding the shortest path, was invaluable for applications requiring exhaustive path exploration.
- **A*** and **IDA*** stood out for their efficiency in navigating complex mazes using heuristic evaluations, significantly reducing the search space and execution time compared to uninformed search strategies.

Challenges Encountered

Several challenges were encountered during the development process, including issues with GUI responsiveness and memory management during intensive search tasks. These challenges were pivotal learning opportunities, providing firsthand experience in debugging and optimizing Python code for better performance.

Final Thoughts

In conclusion, this project has not only reinforced the practical applications of classic AI search algorithms but has also provided a platform for deeper understanding and innovation in algorithmic problem-solving within AI. The experiences and knowledge gained from this project will undoubtedly serve as a valuable foundation for future explorations into AI and robotics.

Acknowledgements/ Resources

- The video script explains the Breadth First Search algorithm using an example graph to find the path from the root node to the goal node. It demonstrates how to expand the graph into a tree and choose the path at the second depth level to reach the goal node.

  https://www.youtube.com/watch?v=GfgdWpgBJ0A&list=PLrjkTql3jnm_yol-ZK1QqPSn5YSg0NF9r&index=13&ab_channel=Education4u

- The video script introduces key concepts in artificial intelligence, including transition models, search algorithms like A* and Minimax, and heuristics for optimal solutions in various scenarios. It emphasizes the importance of game components and search strategies like DFS, BFS, and Greedy Best-First Search for efficient pathfinding and decision-making in games.
  https://www.youtube.com/watch?v=WbzNRTTrX0g

- This video script is a beginner course on Tkinter for Python GUI development. It covers the basics of creating windows, setting up widgets like labels, text boxes, buttons, and menus, as well as adding functionality to the GUI elements. The script also demonstrates how to handle events like closing the window and keyboard shortcuts.
  https://www.youtube.com/watch?v=ibf5cx221hk&t=975s&ab_channel=NeuralNine

References

Russell, S., & Norvig, P. (1995). Artificial intelligence: a modern approach. *Choice/Choice Reviews*, *33*(03), 33–1577. https://doi.org/10.5860/choice.33-1577

GeeksforGeeks. (2024, March 8). *Breadth first search or BFS for a graph*. GeeksforGeeks. https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

GeeksforGeeks. (2024a, February 16). *Depth first search or DFS for a graph*. GeeksforGeeks. https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/

GeeksforGeeks. (2024a, January 18). *Greedy Best first search algorithm*. GeeksforGeeks. https://www.geeksforgeeks.org/greedy-best-first-search-algorithm/

GeeksforGeeks. (2024c, March 7). *A search algorithm*. GeeksforGeeks. https://www.geeksforgeeks.org/a-search-algorithm/

GfG. (2023, February 20). *Iterative Deepening Search(IDS) or Iterative Deepening Depth First Search(IDDFS)*. GeeksforGeeks. https://www.geeksforgeeks.org/iterative-deepening-searchids-iterative-deepening-depth-first-searchiddfs/

GfG. (2023b, March 28). *Iterative Deepening A algorithm (IDA) Artificial intelligence*. GeeksforGeeks. https://www.geeksforgeeks.org/iterative-deepening-a-algorithm-ida-artificial-intelligence/

CS50. (2023, July 24). *Search - Lecture 0 - CS50's Introduction to Artificial Intelligence with Python 2020* [Video]. YouTube. https://www.youtube.com/watch?v=WbzNRTTrX0g