**Analysis of Graph Search Algorithms:**

**Depth-First Search, Breadth-First Search, Dijkstra and A\* Search in Maze Solving**

**Krishna Gopal Sharma**

**Student Number: B1055988**

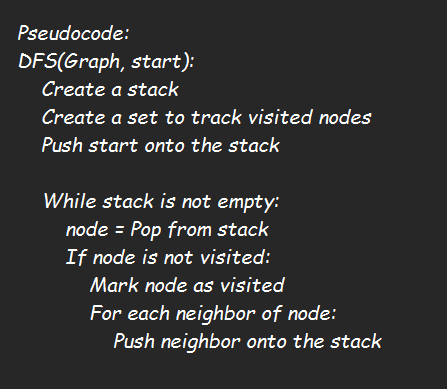
**Artificial Intelligence ICA S3454618**

**Total Words:**

**Introduction**

In this report I’m going to express, explore and compare graph search algorithms: 1. Depth-First Search (DFS), 2. Breadth-First Search (BFS), 3. Dijkstra, and 4. A\*. These algorithms are widely used in fields like Artificial Intelligence (AI) for tasks such as finding paths, exploring game states, and making decisions. By applying each algorithm to a maze-solving problem (same problem for each algorithm), I aim to showcase their strengths and differences, focusing on factors like path length, search efficiency, and execution time. This comparison will help in understanding when and why each algorithm is best suited for different types of problems.

Every Algorithm has its pros and cons. By applying these algorithms on same problem types I can be able to conclude that which algorithm should be used in which circumstances. SO, lets explore them one by one.

**1. DFS (Depth-First Search):** Is an algorithm used to explore all the nodes in a graph by starting at a chosen node and exploring as deep as possible along each branch before backtracking. It’s like following a path until you can’t go further, then going back and trying another path. DFS is good for problems like finding connected components or checking if a path exists in a graph.

A black background with white text

Description automatically generated

**2. FS (Breadth-First Search):** Explores a graph level by level, starting from the source node and visiting all its neighbours first, then moving to the neighbour’s neighbours, and so on. It uses a queue to track which nodes to visit next. BFS is especially useful for finding the shortest path in an unweighted graph, as it checks all possibilities in layers.

A black background with white text

Description automatically generated

**3. Dijkstra's Algorithm:** Finds the shortest path from one node to all others in a graph with non-negative edge weights. It works by picking the closest unvisited node, then updating the shortest known distances to its neighbours. Dijkstra’s is perfect for situations where you need the shortest path, like in routing or navigation, but only works with positive weights**.**

A black screen with white text

Description automatically generated

***4. A\* (A-Star) Algorithm***: It is a more efficient version of Dijkstra’s, which also uses a heuristic to estimate how close each node is to the destination. It combines the shortest path found so far with an estimate of how far a node is from the target, helping it to prioritize which paths to explore first. A\* is widely used in applications like game development and GPS systems because it speeds up the search process.

**Maze Setup 25x25.**

The maze grid I used here represents a 25x25 cell layout, where each cell has four directional values indicating the presence of walls (East, West, North, South). A value of 1 means a wall exists in that direction, and 0 means no wall. The goal is to navigate from a start point (at the bottom-right corner) to the endpoint, (at top-left corner). Each cell is described by its coordinates and its wall configuration, with East (E), West (W), North (N), and South (S) walls being specified for each. For example, cell (1,1) has walls to the North and West (1,0,0,1), while (2,1) has no walls to the North or South (0,0,1,1). By following these paths and avoiding blocked routes, the maze can be solved by delving into valid passages while respecting the wall constraints. The ultimate challenge is to find a continuous route from start to finish, navigating through open cells and avoiding dead-ends.

**Code Explanation for Graph Search Algorithms in my setup Maze**

This section I provide an explanation of how these popular graph search algorithms—**Breadth-First Search (BFS)**, **Depth-First Search (DFS)**, **Dijkstra** and **A\* (A-Star)**—are implemented and visualized in a maze-solving scenario. The code uses the pyamaze library to create and visualize mazes, along with different agents to show how each algorithm explores the maze and finds a path. Let’s go through each algorithm systematically.

**1. BFS (Breadth-First Search)**

1. **Necessary Libraries Imported**:
   * **pyamaze**: This library helps generate and visualize mazes.
   * **deque**: A special queue that allows adding and removing elements from both ends, making it efficient for BFS.
   * **COLOR, agent, textLabel**: Used to set up the visual appearance of the maze and display information.
2. **BFS Function**:
   * **Start Cell**: The search begins from the bottom-right corner of the maze.
   * **Frontier**: A deque keeps track of the cells that need to be explored next.
   * **Explored**: A list that stores cells that have already been visited to avoid revisiting them.
   * **BFS Path**: A dictionary that tracks the path taken from the start cell to each visited cell.
   * **Goal Detection**: The algorithm stops once it reaches the goal (usually the top-left corner of the maze).
   * **Path Reconstruction**: Once the goal is reached, the BFS path is reconstructed from the goal to the start using the BFS Path dictionary, creating the shortest path.
3. **Main Section**:
   * **Maze Creation**: The maze is generated using the pyamaze library and a file (e.g., mazetest.csv).
   * **BFS Execution**: The BFS function is called to find the order in which the cells are explored, the path taken, and the shortest path from the goal to the start.
   * **Agents**:
     + **Agent 1**: Visualizes the BFS search order (how BFS explores the maze).
     + **Agent 2**: Traces the full BFS path.
     + **Agent 3**: Shows the forward path, moving from the goal back to the start.
   * **Trace Path**: This function animates the movement of the agents along their respective paths.
   * **Labels**: Labels are displayed to show the lengths of the BFS search path and the forward path.
   * **m.run()**: Starts the visualization, showing how BFS explores the maze and traces the path.
4. **Key Concepts**:
   * **BFS Algorithm**: BFS is a graph traversal algorithm that explores all possible moves from a cell before moving on to the next. It guarantees finding the shortest path in an unweighted maze.
   * **Agents**: Visual representations that trace the BFS path and explore the maze.
   * **Maze Visualization**: pyamaze provides an interactive visualization where the agents' movements can be traced step-by-step.

**2. DFS (Depth-First Search)**

1. **Necessary Libraries Imported**:
   * **pyamaze**: Used to create and visualize mazes.
   * **deque**: Although not used directly in this DFS implementation (DFS uses a stack), the deque could be useful for more advanced implementations or other traversal algorithms.
   * **agent, textLabel, COLOR**: Used for visualizing the agents, updating their colors dynamically, and displaying helpful information (like path length).
2. **DFS Function**:
   * **Start Cell**: The search starts from a predefined point, typically the bottom-right corner of the maze unless specified otherwise.
   * **Stack**: DFS uses a stack (LIFO - Last In, First Out) to explore deeper paths first. The stack keeps track of the cells to be explored next, with the most recently discovered cell explored first.
   * **Visited Cells**: A dictionary (visited) tracks the path taken to each cell, helping to prevent revisiting the same cells and avoiding infinite loops.
   * **Exploration Order**: A list (exploration\_order) keeps track of the order in which cells are explored by DFS.
   * **Goal Detection**: DFS continues exploring until it reaches the goal. When the goal is reached, the search terminates.
   * **Path Reconstruction**: After finding the goal, the algorithm reconstructs the path from the start to the goal by backtracking using the visited dictionary, which maps each visited cell to the cell from which it was reached.
3. **Main Section**:
   * **Maze Creation**: The maze is generated using the pyamaze library and loaded from a CSV file (e.g., mazetest.csv).
   * **DFS Execution**: The DFS function is called to perform the search. It returns:
     + The order in which the cells were explored (exploration\_order).
     + A dictionary (visited) containing the parent-child relationship of the explored cells.
     + The reconstructed path (path\_to\_goal) from the goal back to the start.
   * **Agents**:
     + **Agent 1 (DFS Search Agent)**: Visualizes how DFS explores the maze, changing color as it moves along the exploration order.
     + **Agent 2 (DFS Full Path Agent)**: Traces the entire path from start to goal after DFS finishes searching.
     + **Agent 3 (Goal Agent)**: Moves along the final path from the goal to the start, following path\_to\_goal.
   * **Trace Path**: The tracePath() method animates the agents' movements along their respective paths, showing how the algorithm explores and ultimately reaches the goal.
   * **Labels**: The textLabel() method displays the lengths of the DFS path and search order for informative purposes.
   * **m.run()**: Starts the interactive maze visualization, displaying the agents and their movements across the maze, step-by-step.
4. **Key Concepts**:
   * **DFS Algorithm**: Depth-First Search explores as deeply as possible along a path before backtracking. It does not guarantee the shortest path in an unweighted maze, but it will find a path if one exists. DFS may explore longer paths first and may not find the most optimal path unless explicitly optimized.
   * **Stack**: DFS uses a stack to prioritize deeper exploration. It works by pushing unvisited neighboring cells onto the stack and exploring the most recent one. When a path is blocked or a dead end is reached, the algorithm backtracks by popping cells from the stack and exploring alternative paths.
   * **Agents**: Visual representations of the DFS process, tracking the exploration order, full path, and goal-reaching process.
   * **Maze Visualization**: pyamaze provides an interactive maze, where each agent's movement is animated, and step-by-step progress is shown.

**3. A\* Algorithm (A-Star)**

1. **Necessary Libraries Imported**:
   * **pyamaze**: Used to generate and visualize mazes, creating agents, and tracing paths.
   * **heapq**: Implements a priority queue (min-heap) to manage cells to be explored based on their f score (cost).
   * **agent, textLabel, COLOR**: Used to create and visualize agents in the maze and display relevant information such as path length or number of cells explored.
2. *A Algorithm Function*\*:
   * **Start Cell**: The A\* search begins at the bottom-right corner of the maze by default.
   * **Priority Queue (Open List)**: A priority queue stores cells to be explored. Each cell has an associated f score, calculated as the sum of the actual cost to reach the cell (g score) and a heuristic estimate of the cost to reach the goal (h score). This ensures the algorithm always explores the most promising cell next.
     + The formula used is:  
       f(n) = g(n) + h(n)
     + g(n): The cost to reach node n from the start.
     + h(n): The heuristic estimate of the cost from node n to the goal (in this case, Manhattan distance).
   * **Closed List**: Once a cell is fully explored (i.e., all its neighbors have been processed), it is added to the closed list to prevent revisiting it.
   * **Path Reconstruction**: After reaching the goal, the algorithm backtracks from the goal to the start using the came\_from dictionary to reconstruct the shortest path.
   * **Exploration Order**: A list exploration\_order keeps track of the order in which cells are explored during the A\* search. This is used to visualize the agent’s movement.
3. **Main Section**:
   * **Maze Creation**: A maze is generated using the pyamaze library and loaded from a custom CSV file (e.g., mazetest.csv).
   * *A Execution*\*: The astar() function is called to perform the search. It returns:
     + The order in which cells were explored (exploration\_order).
     + A dictionary (came\_from) that maps each visited cell to its parent.
     + The reconstructed path (path\_to\_goal) from the start to the goal.
   * **Agents**:
     + *Agent 1 (A Search Order)\*\*: Visualizes how the A* algorithm explores the maze, following the exploration order.
     + *Agent 2 (A Path)\*\*: Traces the shortest path found by A*, moving from the start to the goal.
   * **Trace Path**: The tracePath() function animates the agents' movements through the maze, showing the exploration order and final path.
   * **Labels**: Displays the length of the A\* path and the number of cells explored.
   * **m.run()**: Starts the maze visualization, showing how A\* explores the maze and finds the shortest path.
4. **Key Concepts**:
   * *A Algorithm*\*: A\* is a popular pathfinding algorithm that uses both actual cost (g score) and heuristic estimate (h score) to find the shortest path. It guarantees finding the shortest path if the heuristic is admissible (never overestimates the true cost).
   * **Priority Queue (Min-Heap)**: The priority queue ensures that the algorithm always explores the most promising node (the one with the lowest f score) first, improving efficiency over BFS and DFS.
   * **Manhattan Distance**: A common heuristic used in grid-based pathfinding, where the distance between two points is calculated by summing the absolute differences of their x and y coordinates.

**4. Dijkstra's Algorithm**

1. **Necessary Libraries Imported**:
   * **pyamaze**: Used for maze generation and visualization.
   * **heapq**: Implements a priority queue (min-heap) to manage cells based on their current shortest distance.
   * **agent, textLabel, COLOR**: Used for visualizing the maze, agents, and displaying path-related information.
2. **Dijkstra Algorithm Function**:
   * **Start Cell**: Dijkstra begins from the bottom-right corner of the maze by default.
   * **Priority Queue**: A priority queue is used to explore cells based on their current shortest distance from the start. The priority queue ensures that the algorithm always explores the closest unvisited cell next.
   * **Explored List**: A set that stores cells that have already been visited to avoid revisiting them.
   * **Distance Dictionary**: A dictionary (dist) stores the shortest known distance to each cell.
   * **Path Reconstruction**: After the goal is reached, the algorithm backtracks from the goal to the start using a came\_from dictionary that tracks the parent of each cell to reconstruct the shortest path.
3. **Main Section**:
   * **Maze Creation**: A maze is created using pyamaze and loaded from a custom CSV file (e.g., mazetest.csv).
   * **Dijkstra Execution**: The dijkstra() function is called to perform the search. It returns:
     + The order in which the cells are explored.
     + The dictionary (came\_from) used to reconstruct the shortest path.
   * **Agents**:
     + **Agent 1 (Dijkstra Search Agent)**: Follows the exploration order while visualizing the shortest path discovery.
     + **Agent 2 (Dijkstra Path)**: Traces the shortest path from the start to the goal.
   * **Trace Path**: The tracePath() method animates the agents' movements, showing the exploration and shortest path.
   * **Labels**: Displays the total number of cells explored and the length of the shortest path.
   * **m.run()**: Starts the visualization to show how Dijkstra explores the maze and finds the shortest path.
4. **Key Concepts**:
   * **Dijkstra's Algorithm**: A graph search algorithm that finds the shortest path by exploring the closest unvisited cell. It guarantees the shortest path in a weighted graph where all edge weights are non-negative.
   * **Priority Queue (Min-Heap)**: Ensures the algorithm explores the cell with the smallest distance first.
   * **Shortest Path**: Dijkstra guarantees the shortest path because it explores cells based on their actual distance from the start.

**Comparison of Results:**

Dijkstra, and A\* all find the shortest path (length = 49), while DFS produces a longer, suboptimal path (length = 59). BFS and Dijkstra explore a similar number of nodes (623 and 621), whereas DFS explores far fewer (94) but at the cost of optimality. A\* is the most efficient, leveraging a heuristic to minimize exploration while still finding the shortest path. Although BFS and Dijkstra guarantee optimal paths, they are less efficient compared to A\*.

This doesn’t mean that A\* will work better in all situations or in all mazes. There are many possible cases that A\* will not perform better. Like what if the goal is direct near to the starting state. The agent chooses to go left and find the goal directly, which lead to BSF as best algorithm and if the same goal is in left but at far depth then the DFS is best algorithm.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Algorithm** | **Path Optimality** | **Efficiency** | **Notes** | **Path Length** | **Search Length** |
| **BFS** | Optimal | Medium | Guarantees shortest path but explores all nodes at the current level. | 49 | 623 |
| **DFS** | Suboptimal | Low | Explores exhaustively without optimization. | 59 | 94 |
| **Dijkstra** | Optimal | Medium | Guarantees optimality but explores more than A\*. | 49 | 621 |
| **A**\* | Optimal | High | Uses a heuristic to minimize exploration. | 49 | 195 |