**Analysis of Graph Search Algorithms:**

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

**Krishna Gopal Sharma**

**Student Number: B1055988**

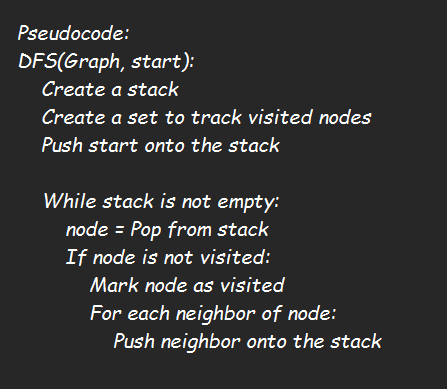
**Artificial Intelligence ICA S3454618**

**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 screen with white text

Description automatically generated

**3. Greedy BFS**: Tries to find the shortest path by always choosing the node that looks closest to the goal, based on a given heuristic. It explores the nodes in the order that seems best, but it doesn’t always find the shortest path because it only looks at the "closest" node, not the full path cost. It's quick and works well when you have a good guess about where the goal is, but it doesn’t guarantee the best result.

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 30x50.**

The maze grid I used here represents a 30x50 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)**, **Greedy\_BSF** 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. Greedy Best-First Search (Greedy BFS) Algorithm**

The **Greedy Best-First Search (Greedy BFS)** algorithm is a pathfinding algorithm that explores nodes based on a heuristic that estimates how close a node is to the goal. Unlike BFS, which explores nodes level by level, Greedy BFS chooses the node that appears closest to the goal based on the heuristic. However, it does not account for the path cost to reach the node, so it is faster but may not always find the optimal path.

**I. Necessary Libraries**

* + **pyamaze: Used for maze generation and visualization.**
  + **heapq**: A Python library implementing a priority queue (min-heap) to efficiently select the next node to explore.
  + **COLOR, agent, textLabel**: For visualizing the maze and displaying relevant information such as path lengths.

**II. Greedy BFS Function:** The function greedy\_bfs() follows these key steps:

* + **Start Cell**: The algorithm begins at the bottom-right corner of the maze (or a specified start point).
  + **Priority Queue:** A priority queue (min-heap) is used to store the cells to be explored, prioritized by their heuristic value, which estimates their distance to the goal.
  + **Heuristic:** A Manhattan distance heuristic is used to calculate the distance from the current cell to the goal. This guides the search toward the goal.
  + **Exploration:** The algorithm explores neighboring cells (up, down, left, right) from the current cell. If the cell has not been visited yet, it is added to the priority queue with its heuristic value.
  + **Goal Detection:** The algorithm stops once it reaches the goal cell.
  + **Path Reconstruction:** Once the goal is reached, the path is reconstructed by tracing back from the goal to the start using a came\_from dictionary, forming the path from start to goal.

**III. Main Section**

* + **Maze Creation:** The maze is created using pyamaze and can load custom mazes from a CSV file.
  + **Greedy BFS Execution**: The greedy\_bfs() function is called, returning:
  + **Exploration Order**: The sequence in which the cells are explored.
  + **Came From:** A dictionary that stores the previous cell for each visited cell.
  + **Path to Goal:** The reconstructed path from the start to the goal.
  + **Agents:**
    - Agent 1: Visualizes the order of exploration (Greedy BFS).
    - Agent 2: Visualizes the full path from start to goal.
    - Agent 3: Visualizes the forward path from start to goal.
    - Trace Path: Animates the agents’ movement through the maze, showing the exploration and the final path.
  + **Labels**: Displays the lengths of the Greedy BFS search path and the final path.
  + **Visualization:** m.run() starts the simulation and shows the algorithm in action.

**IV. Key Concepts**

* + **Greedy BFS Algorithm:** Greedy BFS is a heuristic-driven search algorithm that explores nodes based solely on their estimated distance to the goal. It does not consider the path cost, making it faster but less optimal.
  + **Heuristic:** The heuristic used in Greedy BFS is the Manhattan distance, which estimates the number of steps required to reach the goal from a given cell.
  + **Priority Queue:** The min-heap ensures that the cell with the smallest heuristic value is explored first, optimizing the search toward the goal.
  + **Agents:** Visual representations help track the search process, showing how the algorithm explores the maze and traces the final path.

**Comparison of Results:**

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 |

In comparison with each Algorithm, I have created 3 scenarios for it. The goal position in each scenario is different, and over this position, we compare which algorithm perform.