Josep Cubedo, Justin Jelinek, Juli Delgado and Martí Alsina

DATA STRUCTURES AND ALGORITHM DESIGN
ASSIGNMENT 2: Tree Search

1. Code Description

1.1. Maze I

I. is_within_bounds(maze, x, y)

The function is_within_bounds checks whether the coordinates (x, y) are within the valid boundaries of a 2D maze. It ensures that the given x and y values are within the row and column limits of the maze, respectively.

This function prevents out-of-bounds access to the maze. When navigating through the maze (for example, in search algorithms like DFS or BFS), this function ensures that movements stay within the valid grid area and don't attempt to access invalid positions outside the maze.

This function is particularly useful in maze-solving algorithms or grid-based traversal where boundary checks are necessary to avoid errors or unwanted behavior.

II. is_walkable(maze, x, y)

The function is_walkable checks if the cell at coordinates (x, y) in a given maze is a valid or walkable location. It returns *True* if:

- The cell contains a *1*, indicating a walkable path.
- The cell contains *'B',* which represents the exit of the maze.

Otherwise, it returns *False*, meaning the cell is not walkable (e.g., it's a wall or invalid).

The purpose of this function is to determine whether a specific position in the maze is accessible or not. It helps guide maze traversal algorithms, such as DFS or BFS, by identifying which cells can be visited and explored. This check ensures that movements are restricted to valid paths and the exit point, avoiding walls or other non-walkable areas in the maze.

III. tree_creation(maze)

The tree_creation function converts a 2D maze into a tree structure using **Depth-First Search (DFS)**. Starting from the entrance of the maze (represented by 'A' at position (0, 0)), the function recursively explores all possible walkable paths in the maze and constructs a tree where each node represents a valid position in the maze.

The purpose of this function is to transform the maze into a tree representation, where each node corresponds to a walkable position and its children represent the neighboring walkable cells. The tree structure can then be used for search algorithms (like DFS, BFS) to solve the maze by finding paths from the entrance ('A') to the exit ('B'). It provides a foundation for solving maze-related problems by organizing the maze's structure into a traversable format.

This function is especially useful for exploring all possible paths from the start to the end of the maze, or for visualizing the maze as a tree of connected nodes.

1.2. Maze II

I. DFS(Tree, target)

The DFS function performs a **Depth-First Search (DFS)** on a tree using **preorder traversal**, which means it explores each node's children in sequence before moving to the next sibling. The function checks each child node for a specific target value, and if it finds a match, it returns that node. If the target isn't found at the current level, the function recursively searches the children of each node.

This function searches for a node with a specific value (target) in a tree using DFS. It navigates the tree structure by exploring each child node before moving deeper into each subtree. When the target node is found, the function returns it.

II. BFS(Tree, target)

The BFS function performs a **Breadth-First Search (BFS)** on a tree to find a node with a specific *target* value. BFS explores the tree level by level, starting from the root node (Tree) and moving through all of its children before descending to the next level of nodes. It uses a queue to keep track of nodes that need to be explored. The function dequeues the next node in line, checks if it matches the target, and if not, adds its children to the queue for further exploration.

This function finds a node with a specific *target* value in a tree using BFS. Unlike DFS, which explores one path fully before backtracking, BFS explores nodes at the same depth before moving to deeper levels. BFS is ideal when the target node is expected to be closer to the root, as it ensures that shallower nodes are checked first.

III. find_path(Tree)

The find_path function traces the path from a given node back to the root of a tree or graph by following each node's parent pointer. It builds a list of the coordinates (x, y) of each node along the path and then returns the path in reverse order, so it starts from the root and ends at the given node.

This function reconstructs and returns the path from the root (e.g., the entrance of a maze) to a specific target node (e.g., the exit) after a search algorithm (DFS or BFS) has found the target. The function effectively traces the route taken by the search algorithm to reach the target.

IV. measure_execution_time(algorithm, tree, target):

The measure_execution_time function measures how long it takes for a given algorithm to execute. The function accepts an algorithm (such as BFS or DFS), a tree structure (on which the algorithm will operate), and a target value (which the algorithm will search for). It records the time before and after the algorithm runs, then calculates and returns the elapsed time.

This function measures the performance (execution time) of a given algorithm. It is particularly useful for comparing the efficiency of different algorithms (like BFS vs. DFS) when applied to the same problem or dataset, such as searching for a node in a tree. Returning the total time taken helps evaluate the relative speed and efficiency of the algorithms.

V. run_experiment(iterations = 50)

The run_experiments function executes **BFS** and **DFS** multiple times on three different mazes and measures their execution times. It runs each algorithm a specified number of times (we chose 50) and records the time for each run. This repetition is done to get more reliable results, as single execution measurements may sometimes vary in results due to the algorithms completing too quickly for precise timing. The function then calculates and returns the average execution times for both BFS and DFS across the mazes.

This function's purpose is to obtain accurate and averaged execution times for BFS and DFS by running each algorithm multiple times (to avoid unreliable single-time results) and comparing their performance across different maze configurations. This approach ensures more stable and meaningful performance data by mitigating timing precision issues, which occur when the algorithms run too quickly to be measured reliably in a single execution.

VI. plot_results(avg_bfs_times, avg_dfs_times)

The plot_results function visualizes the average execution times for BFS and DFS algorithms across three different maze sizes using a bar chart. It takes in the average times for BFS (avg_bfs_times) and DFS (avg_dfs_times) from 50 trials and plots them side by side for easy comparison. The chart uses a logarithmic scale on the Y-axis to better highlight differences in execution time, especially if there are large variations between the algorithms.

This function provides a clear, visual comparison of BFS and DFS's performance across different maze sizes. Plotting the average execution times allows for an easy interpretation of how the two algorithms scale with increasing maze complexity.

1.3. Maze III

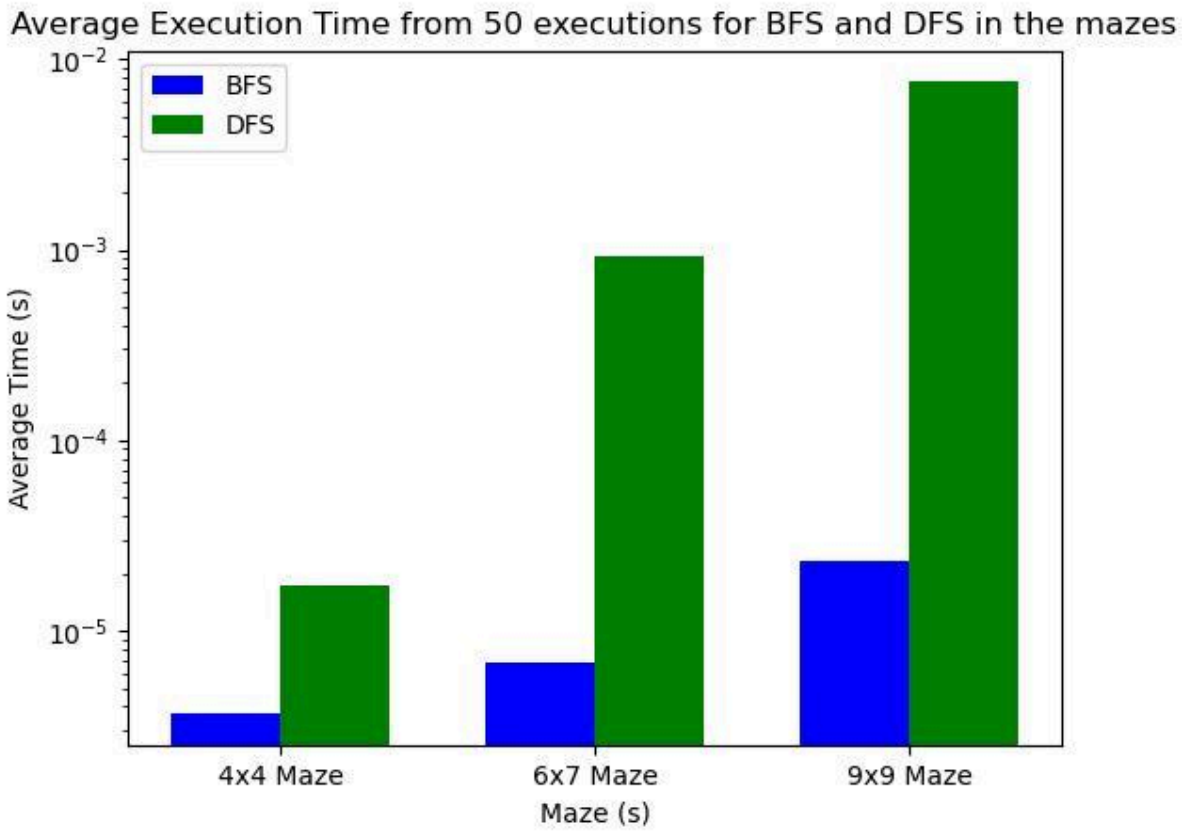
I. tree_creation_with_loops(maze,node)

The tree_creation_with_loops function creates a tree structure from a maze, handling **possible loops** in the maze. It uses a **Breadth-First Search (BFS)** approach, with a queue to explore the maze iteratively and a *visited* set to avoid revisiting nodes, preventing infinite loops. The tree starts at the root node, representing the entrance to the maze, and adds child nodes for each walkable path.

The goal is to **transform a maze into a tree** structure while handling loops in the maze. By tracking visited nodes, the function ensures that no path is traversed more than once, allowing exploration of all valid routes while avoiding cyclic paths. This is essential when dealing with mazes that contain loops, ensuring the search algorithm doesn't get stuck in infinite cycles.

The tree generated by this function can be used for a variety of purposes, such as searching for a target node, visualizing the maze's structure, or solving pathfinding problems.

2. Graph comparing DFS vs BFS in Maze II



The graph shows the **average execution times** of **BFS (blue)** and **DFS (green)** across three different maze sizes: **4x4**, **6x7**, and **9x9**, after 50 executions. The y-axis represents the average execution time in seconds, and the x-axis indicates the maze sizes. Importantly, the y-axis is set to a **logarithmic scale** to better visualize the differences in execution times.

Initially, the execution times for both BFS and DFS, especially for smaller mazes, were so small that they didn't show up clearly on a regular scale. Following the teacher's advice, we switched to a logarithmic scale on the y-axis. This allowed us to visualize the execution time differences more clearly, especially as the maze size increases and the performance gap between BFS and DFS becomes more significant.

BFS (iterative) is faster in this context because it uses a **queue**, which avoids the overhead of recursive function calls. The BFS code iterates through nodes, as shown here:

```
    while queue:
        # rest of BFS code
```

DFS (recursive) involves multiple function calls and deeper recursion, which introduces additional overhead, especially in larger mazes. This slows down DFS, particularly as maze complexity grows. The DFS code recursively traverses the tree, as shown here:

```
for child in tree.children:
   node = DFS(child, target)
   # rest of DFS code
```

As the maze size increases, the time difference between BFS and DFS becomes more noticeable, with **BFS** showing significantly better performance for larger mazes.

DFS suffers more from deeper recursion, especially in the larger 6x7 and 9x9 mazes, whereas **BFS** maintains its efficiency across maze sizes.

Conclusion

The graph suggests that BFS consistently performs faster than DFS across all maze sizes, with the performance gap increasing as the maze complexity grows. This aligns with the general behavior of iterative algorithms (BFS) being more efficient than recursive ones (DFS) when recursion depth and problem size increase.

3. Self Assessment

Issues with Group Management

We had no real issues communicating or anything, and work was extremely smooth. The only big challenge we encountered was plotting the times, which took us more than a week to solve, but with the help of the professor, it was solved quickly. Details are explained in the graph section with the plotting times. Furthermore, We created a shared coding document with pycharm to work. But overall, it was always pleasant to work with each other.

We did it all together in the class (and finished Maze 1 in class), and after that, each of us did the following:

Josep Cubedo: Finished the code of Maze II and part of its comments, completed part of the document, and helped manage and code the graph.

Justin Jelinek: Finished the code of Maze III, added comments to the code, did part of the coding of the graph, and helped with the document.

Juli Delgado: Added comments to the code, helped with coding of Maze III, completed part of the document, and helped with the coding of the graph.

Martí Alsina: Added comments to the code, helped with the code of Maze II, completed part of the document, helped managing the graph.