AI GROUP PROJECT – 1

ANU REKULA

NIRALEE KOTHARI

NEERAJ SHANKAR UMMADISINGHU

OVERVIEW:

In this programming project, the objective is to implement and compare three search algorithms: Depth-First Search (DFS), Breadth-First Search (BFS), and A* algorithm. The goal is to solve a specific problem represented by a 2D grid maze. The maze comprises open cells ('.') denoting valid paths and obstacles ('X') representing blocked areas.

The project involves:

1. Maze Generation: Implementing a function to generate a random maze with a size of 10x10 and a variable density of obstacles using Python. Each maze should have a defined starting point (S) and a goal point (G). It's essential that each run of the algorithms produces a different maze configuration in terms of starting point, goal point, and obstacle location. However, within each run of the three algorithms (DFS, BFS, A*), the maze configuration remains the same.

2. Search Algorithms: Implementing the three search algorithms:

- Depth-First Search (DFS)
- Breadth-First Search (BFS)
- A* algorithm

3. Evaluation: Evaluating the performance of each algorithm in finding the optimal path from the starting point to the goal point in the maze. This evaluation might consider metrics such as time complexity, space complexity, and the length of the path found.

DFS

Depth-First Search (DFS) is a popular graph traversal algorithm used to explore nodes and edges of a graph systematically. The algorithm starts at a designated starting node and explores as far as possible along each branch before backtracking. This means that it traverses deeply into the graph structure before exploring neighboring nodes. DFS is often used in maze solving, pathfinding, topological sorting, and cycle detection.

In the context of the maze-solving problem described above, DFS can be implemented recursively or iteratively. In the recursive approach, the algorithm starts at the starting point (S) and explores one of the neighboring cells. It continues exploring deeper into the maze until it reaches either the goal point (G) or a dead end (obstacle 'X'). Upon encountering a dead end, the algorithm backtracks to the most recent branching point and explores another unvisited neighboring cell. This process repeats until the goal is found or all reachable cells are explored.

For the iterative implementation of DFS, a stack data structure is used to keep track of nodes to be explored. The algorithm starts by pushing the starting node onto the stack and then enters a loop where it pops a node from the stack, explores its neighbors, and pushes them onto the stack if they haven't been visited yet. This process continues until the stack is empty or the goal node is found. Like the recursive approach, iterative DFS also involves backtracking when a dead end is encountered.

The code implements Depth-First Search (DFS) to navigate through a maze.

1. DFS Function (`dfs()`):

- This function takes five parameters: `maze` (the maze grid), `start` (the starting cell coordinates), `goal` (the goal cell coordinates), `visited` (a set to keep track of visited cells), and `path` (a list to store the current path being explored).

- It uses a recursive approach to explore the maze depth-first.

- It marks the current cell as visited and appends it to the `path` list.

- If the current cell is the goal cell, the function returns `True` to indicate success.

- Otherwise, it iterates through the neighboring cells in four directions (up, down, left, right) and recursively calls itself on valid unvisited neighboring cells.

- If no path is found from the current cell, it backtracks by popping the last cell from the `path` list and returns `False`.

2. Visualization Function (`visualize_maze_with_path()`):

- This function visualizes the maze with the path found by DFS.

- It takes two parameters: `maze` (the maze grid) and `path` (the path found by DFS).

- It uses matplotlib to plot the maze grid and colors each cell based on its content (start, goal, obstacle, empty).

- If a path is provided, it plots the path cells in blue.

3. Performance Metrics:

- The code also includes a function `count_expanded_nodes()` to count the number of nodes expanded during the DFS exploration. It counts unique cells visited during the DFS traversal.

- It records performance metrics such as success status, solution path length, number of nodes expanded, and execution time for each run of DFS on the maze.

4. Loop to Run DFS Multiple Times:

- The code runs DFS 10 times on different randomly generated mazes.

- For each run, it generates a new maze, executes DFS on it, records performance metrics, and visualizes the maze with the path found by DFS.

- It prints the performance metrics for each run and displays a visualization of the maze with the path.- It also calculates and displays the average solution path length, number of nodes expanded, and execution time over the 10 runs

BFS

Breadth-First Search (BFS) is a fundamental graph traversal algorithm that systematically explores the nodes of a graph level by level. It starts at a given node, explores all of its neighbors at the present depth level, and then moves on to explore the neighbors at the next depth level before continuing further. This approach ensures that BFS visits all nodes at a given depth level before moving deeper into the graph. BFS is commonly used to find the shortest path between two nodes in an unweighted graph, as it guarantees that the shortest path will be found when applied to such graphs.

In the context of the maze-solving problem described above, implementing BFS involves starting from the designated starting point (S) in the maze and systematically exploring adjacent cells level by level until the goal point (G) is reached. The algorithm maintains a queue data structure to keep track of the nodes to be explored, ensuring that nodes are visited in the order they were discovered. As BFS explores nodes level by level, it guarantees that the shortest path from the starting point to the goal point is found, making it suitable for finding optimal solutions in maze navigation problems, particularly in scenarios where the maze is represented as an unweighted graph.

To implement BFS in the maze-solving problem, you would initialize a queue data structure with the starting point (S) as the initial node. Then, in each iteration of the algorithm, you would dequeue a node from the queue, explore its neighbors, enqueue the unvisited neighbors, and mark them as visited to avoid revisiting them. This process continues until the goal point (G) is reached or until all reachable nodes have been explored. Finally, the algorithm returns the shortest path from the starting point to the goal point, if one exists, providing an efficient and systematic approach to solving maze navigation problems.

The code implements the Breadth-First Search (BFS) algorithm to find the optimal path from the starting point (S) to the goal point (G) in a maze.

1. Initialization:

- A deque (double-ended queue) named `queue` is initialized to store nodes to be explored. It starts with the `start` node.

- A set named `visited` is initialized to keep track of visited nodes to avoid revisiting them.

- A dictionary named `path` is initialized to trace the path from the start node to each explored node.

2. Exploration:

- The algorithm enters a loop that continues until the `queue` is empty or the goal is found.

- Inside the loop, it dequeues a node (`current`) from the front of the queue.

- If the `current` node is the goal, the algorithm breaks out of the loop.

- Otherwise, it explores the neighboring cells of the `current` node in four directions (up, down, left, right).

- For each neighboring cell, it checks if it's within the maze boundaries, not an obstacle, and not visited before. If so, it adds the neighboring cell to the `queue`, marks it as visited, and updates the `path` dictionary to trace the path.

3. Path Reconstruction:

- After reaching the goal, the algorithm reconstructs the path from the `goal` node back to the `start` node using the `path` dictionary.

- It starts from the `goal` node and iteratively follows the parent nodes recorded in the `path` dictionary until it reaches the `start` node.

- The reconstructed path is then returned.

4. Visualization and Performance Metrics:

- The code includes functions for visualizing the maze with the found path and for counting the number of nodes expanded during the BFS exploration.

- After running BFS on the maze for a specified number of times, the code prints out the solution path length, number of nodes expanded, execution time, and success status for each run.

- Finally, it calculates and prints the average solution path length, average number of nodes expanded, and average execution time over all successful runs.

A*

A* (pronounced "A-star") is a popular informed search algorithm commonly used in pathfinding and graph traversal problems. It is an extension of Dijkstra's algorithm with the addition of heuristic estimation to guide the search towards the goal more efficiently. The algorithm maintains two lists: open and closed. The open list contains nodes to be evaluated, sorted by their estimated cost from the start node plus the heuristic estimate to the goal. The closed list contains nodes that have already been evaluated. At each step, A* selects the node with the lowest combined cost from the open list, expands it, and updates its neighbors' costs. The algorithm terminates when the goal node is reached or when the open list becomes empty, indicating that there is no path to the goal.

In the context of the provided problem statement, implementing A* involves augmenting the search process with a heuristic function that provides an estimate of the remaining cost from each node to the goal. This heuristic guides the algorithm to explore paths that are more likely to lead to the goal, improving its efficiency compared to uninformed search algorithms like DFS and BFS. The implementation requires defining an appropriate heuristic function tailored to the maze problem, such as Manhattan distance or Euclidean distance between each cell and the goal. Additionally, A* typically utilizes a priority queue or heap data structure to efficiently manage the open list and select nodes with the lowest combined cost for expansion.

To implement A* in the given scenario, you would iterate through the maze while considering the heuristic value of each cell to the goal. At each step, you would calculate the cost of reaching each neighboring cell and update its cost based on the heuristic estimate. The algorithm would continue until the goal is reached or no more nodes can be explored. Finally, the optimal path from the start to the goal can be reconstructed by tracing back from the goal node using the parent pointers stored during the search process.

The code implements the A* algorithm for finding the optimal path from a start point to a goal point in a maze.

1. Heuristic Function: A* utilizes a heuristic function to estimate the cost from each node to the goal. In this implementation, the Manhattan distance heuristic is used, calculated by the `heuristic` function. The Manhattan distance is the sum of the absolute differences in the x and y coordinates between two points.

2. A* Algorithm: The `astar` function represents the A* algorithm implementation. It takes the maze, start point, and goal point as input arguments. Inside the function:

- A priority queue named `queue` is initialized with an initial tuple containing the heuristic value, cost, current node, and path taken to reach that node.

- A set named `visited` is initialized to keep track of visited nodes to avoid revisiting them.

- The algorithm continues until the priority queue `queue` is not empty. At each iteration:

- The node with the lowest combined cost and heuristic value is popped from the priority queue.

- If the current node is the goal node, the path from the start to the goal is returned.

- Otherwise, the neighbors of the current node are explored. If a neighbor is valid (not an obstacle and within the maze boundaries) and has not been visited, its cost

and heuristic value are calculated and added to the priority queue. The algorithm continues until either the goal is reached or there are no more nodes to explore.

3. Visualization and Performance Evaluation: After implementing the A* algorithm, the code includes visualization and performance evaluation sections. It demonstrates how to visualize the maze exploration process and pathfinding results for multiple runs of the algorithm. The performance metrics such as solution path length, nodes expanded, and execution time are recorded and displayed for each run. Finally, the average performance metrics over multiple runs are calculated and printed.

RESULTS OF PERFORMANCE EVALUATION:

DFS

```
DFS Performance Table (10 Runs):
Run  Solution Path Length Nodes Expanded Execution Time     Status
1    -                    -              0.0000             Fail
2    45                   45             0.0002             Pass
3    35                   35             0.0001             Pass
4    -                    -              0.0002             Fail
5    29                   29             0.0001             Pass
6    37                   37             0.0002             Pass
7    -                    -              0.0001             Fail
8    25                   25             0.0001             Pass
9    -                    -              0.0000             Fail
10   43                   43             0.0002             Pass

Average DFS:
35.666666666666664   35.666666666666664 0.0002
```

BFS

BFS Performance Table (10 Runs):

Run	Solution Path Length	Nodes Expanded	Execution Time	Status
1	19	74	0.0003	Pass
2	-	-	0.0000	Fail
3	23	55	0.0002	Pass
4	19	69	0.0002	Pass
5	19	70	0.0003	Pass
6	-	-	0.0003	Fail
7	-	-	0.0000	Fail
8	-	-	0.0001	Fail
9	-	-	0.0003	Fail
10	19	69	0.0002	Pass

Average BFS:

19.8	67.4	0.0003	

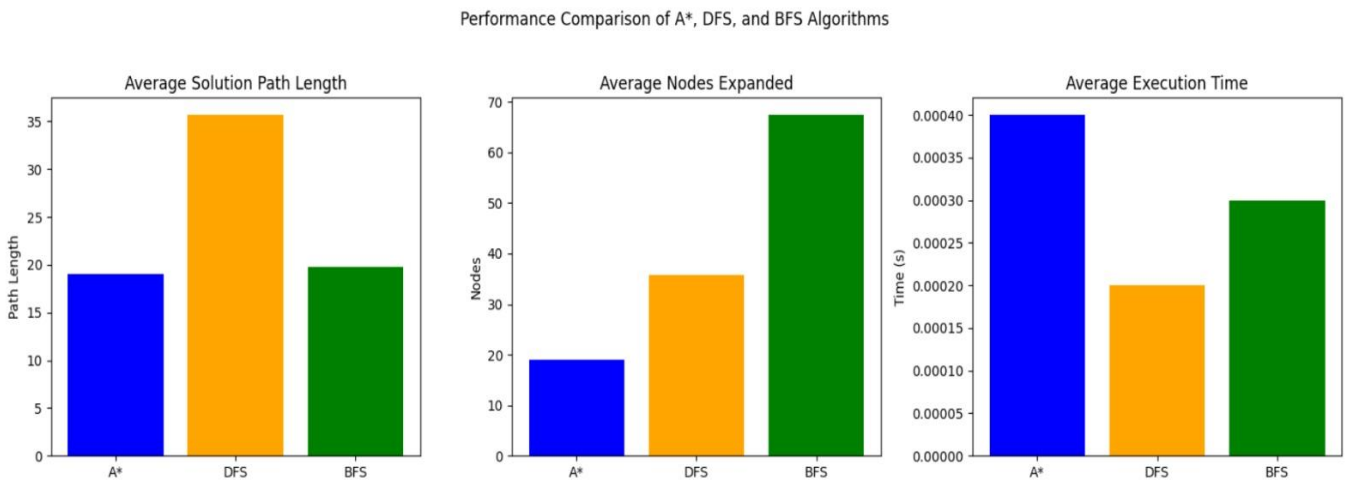
A*

A* Performance Table (10 Runs):

Run	Solution Path Length	Nodes Expanded	Execution Time	Status
1	19	19	0.0004	Pass
2	-	-	0.0006	Fail
3	19	19	0.0003	Pass
4	19	19	0.0004	Pass
5	-	-	0.0000	Fail
6	-	-	0.0001	Fail
7	19	19	0.0003	Pass
8	19	19	0.0003	Pass
9	19	19	0.0003	Pass
10	19	19	0.0006	Pass

Average A*:

19.0	19.0	0.0004	



Comparing the three algorithms based on above performance tables and graph in terms of key metrics: solution path length, nodes expanded, execution time is as follows:

Solution Path Length

- A* consistently finds a solution path of length 19 when it succeeds. This indicates that A is likely finding an optimal path to the goal.

- DFS has variable path lengths ranging from 25 to 45, reflecting its non-optimal nature. DFS can find a path but not necessarily the shortest one.

- BFS shows a bit more consistency in path lengths, with successful runs finding paths of lengths 19 and 23. BFS is known for finding the shortest path in unweighted graphs, which seems partially reflected here.

Nodes Expanded

- A* expands 19 nodes in each of its successful runs, demonstrating high efficiency in searching.

- DFS expands a number of nodes equal to the path length in its successful runs, indicating it explores less efficiently than A*, often going deeper into the search space without pruning.

- BFS expands significantly more nodes (ranging from 55 to 74 in successful runs), which is typical for BFS as it explores widely, examining all neighboring nodes at each depth.

Execution Time

- A* has an average execution time of 0.0004 seconds, showing it is quite fast, likely due to effective pruning of the search space.

- DFS has a slightly lower average execution time of 0.0002 seconds. This is expected as DFS can quickly reach a solution by going deep, even though it may not be optimal.

- BFS also has an average execution time of 0.0003 seconds, which is impressive considering the larger number of nodes it expands but still shows efficient implementation.

VISUALIZATION:

The visualization tool used is matplotlib, a comprehensive library for creating static, interactive, and animated visualizations in Python.

- Library Used: The code utilizes matplotlib, a comprehensive Python library for creating static, interactive, and animated visualizations.

- Functionality: The visualize_maze_with_path function leverages matplotlib to graphically represent the maze and the path discovered by search algorithms like DFS and A.

- Plotting Cells: The plt.scatter function from matplotlib.pyplot is called to plot individual cells of the maze as points on the graph.

- Color Coding: Each type of cell in the maze (such as start, goal, wall, or path) is associated with a specific color, improving the visual distinction between different elements.

- Text Labels: The plt.text function from matplotlib.pyplot adds text labels to the cells, which is useful for indicating the start (S), goal (G), walls (X), and path (P) within the maze.

- Aspect Ratio: plt.gca().set_aspect is used to set the aspect ratio of the plot to 'equal', ensuring that the maze is displayed proportionally without distortion.

- Clean Presentation: The plt.xticks([]) and plt.yticks([]) functions remove x and y-axis tick marks for a cleaner visual representation of the maze.

-Displaying Plot: The plt.show function is called at the end of visualize_maze_with_path to render the plot on the screen.

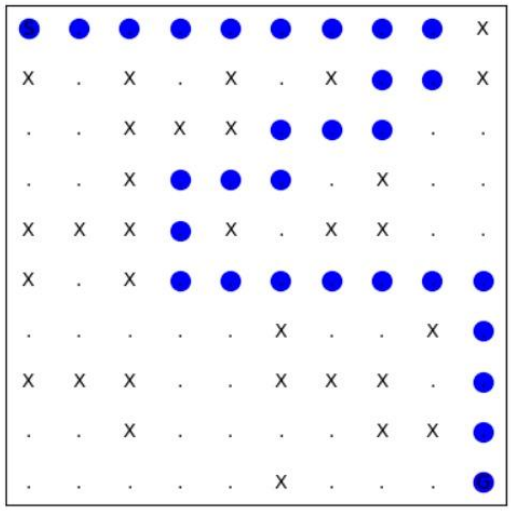
- Grid-Based Data Representation: matplotlib is particularly effective for visualizing grid-based data like mazes, where each cell's position and type can be easily mapped to a color-coded grid on the plot.

- Versatility: While primarily used here for maze visualization, matplotlib is capable of generating a wide range of plots and charts, making it a versatile tool for many different types of data visualization tasks.

Following are few screenshots from 10 runs,

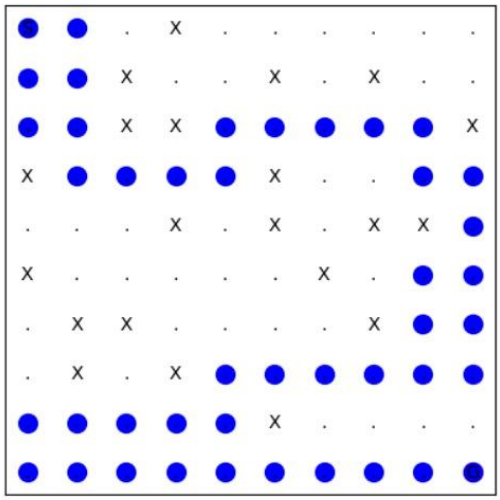
DFS

Run 5:



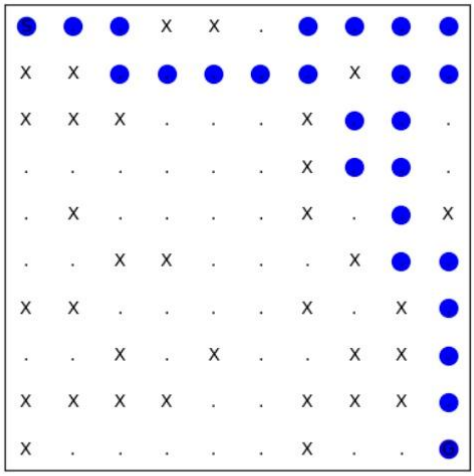
Solution Path Length: 29
Number of Nodes Expanded: 29
Execution Time: 0.00014472007751464844

Run 10:



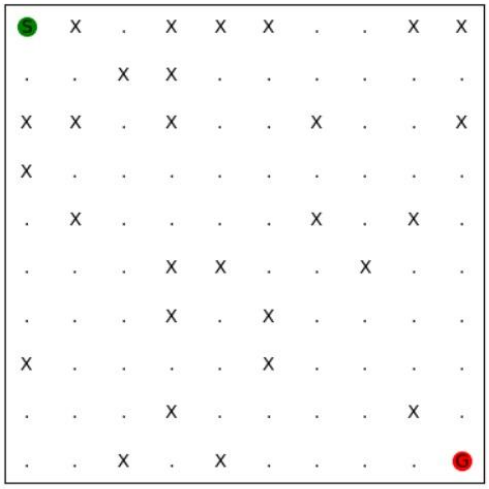
Solution Path Length: 43
Number of Nodes Expanded: 43
Execution Time: 0.0002148151397705078

Run 8:



Solution Path Length: 25
Number of Nodes Expanded: 25
Execution Time: 8.082389831542969e-05

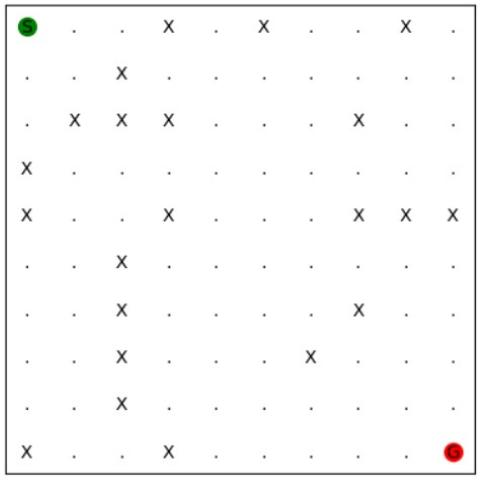
Run 1:



Solution Path Length: Not found
Number of Nodes Expanded: Not found
Execution Time: 2.288818359375e-05

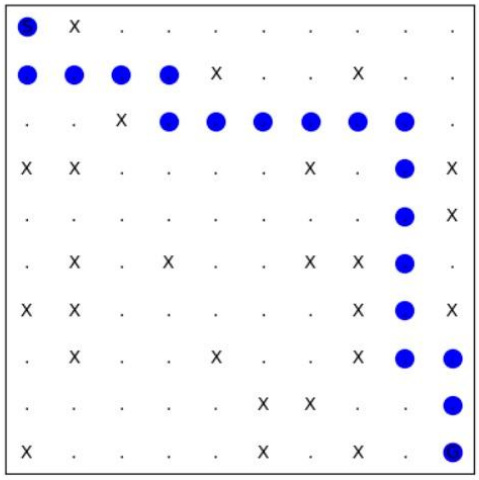
BFS

Run 2:



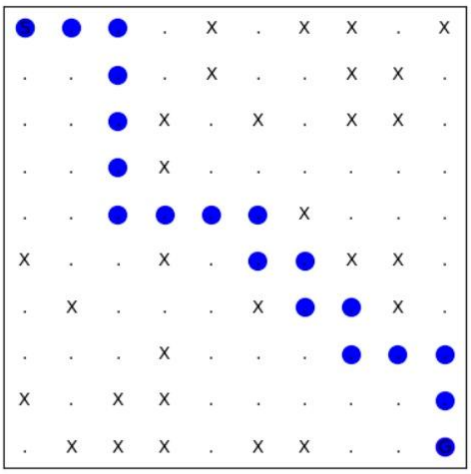
Solution Path Length: Not found
Number of Nodes Expanded: Not found
Execution Time: 3.9577484130859375e-05

Run 1:



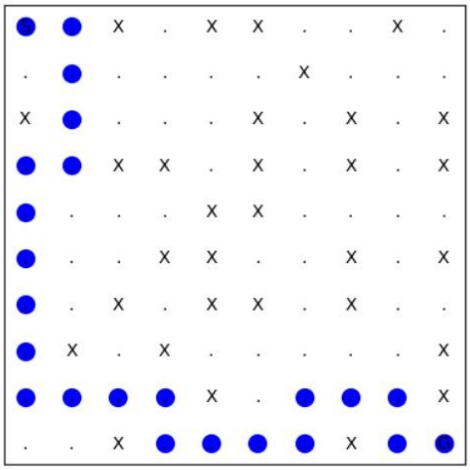
Solution Path Length: 19
Number of Nodes Expanded: 74
Execution Time: 0.00025844573974609375

Run 10:



Solution Path Length: 19
Number of Nodes Expanded: 69
Execution Time: 0.00021123886108398438

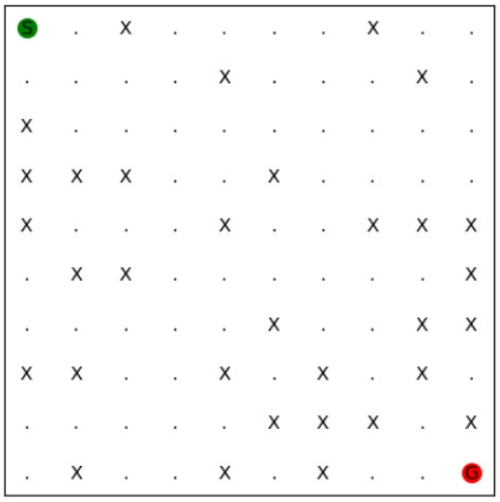
Run 3:



Solution Path Length: 23
Number of Nodes Expanded: 55
Execution Time: 0.00024199485778808594

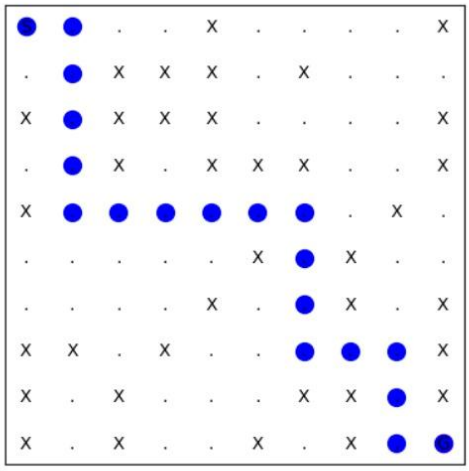
A*

Run 2:



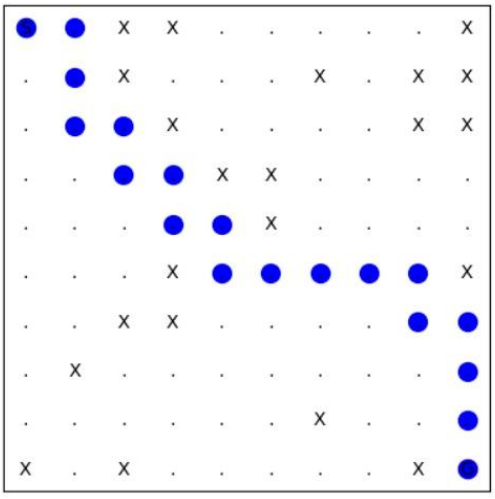
Solution Path Length: Not found
Number of Nodes Expanded: Not found
Execution Time: 0.0006244182586669922

Run 1:



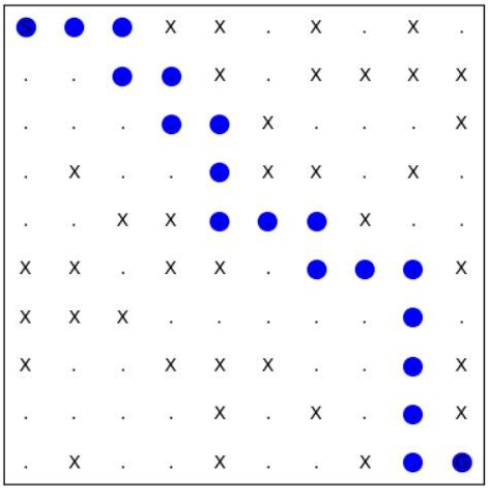
Solution Path Length: 19
Number of Nodes Expanded: 19
Execution Time: 0.0004401206970214844

Run 10:



Solution Path Length: 19
Number of Nodes Expanded: 19
Execution Time: 0.0005807876586914062

Run 7:



Solution Path Length: 19
Number of Nodes Expanded: 19
Execution Time: 0.0002570152282714844

OBSERVATIONS:

From the above screenshots, the following are the observations:

DFS,

- The algorithm is effective at finding a direct path to the goal when possible, and does so with a high degree of efficiency, as indicated by the short execution times.

- The consistency in the number of nodes expanded being equal to the path length in successful runs suggests that the algorithm is finding an optimal path without unnecessary exploration.

- The variation in path lengths likely reflects different maze configurations, indicating the algorithm's adaptability to different scenarios

BFS,

- The algorithm can find paths efficiently with generally short execution times.

- The number of nodes expanded varies, which may depend on the maze's layout and complexity.

- In cases where no path is found, the algorithm terminates without expanding any nodes.

A*,

- When the algorithm finds a path, it tends to be of consistent length (19 in these cases) and directly connects the start to the goal without unnecessary moves.

- The execution times are generally very short, even in the run where no path was found, suggesting the algorithm is efficient in its search process.

- The visual representation of the maze clearly shows the successful paths as a line of blue dots and highlights the start (S) and goal (G) positions with distinct colors.

Therefore, from the results of performance evaluation and above observations we can say that A* is the preferred algorithm for problems where finding an optimal solution efficiently is critical. DFS could be considered for speed in scenarios where solution optimality is less critical, and BFS offers a balance between the two in scenarios where finding the shortest path is necessary and resources permit extensive search.

CONCLUSION:

In summary, the performance evaluation reveals that A* algorithm outperforms DFS and BFS in finding optimal paths efficiently. While DFS is simple but lacks optimality, and BFS guarantees optimality but can be computationally expensive, A* strikes a balance between optimality and efficiency. Its intelligent use of heuristic estimation guides the search towards the goal, resulting in shorter solution paths and fewer nodes expanded. Therefore, A* is the preferred choice for maze navigation problems where finding an optimal solution efficiently is crucial.