Introduction:

The main focus of our project is on generating randomized mazes with the help of randomized depth first search algorithm, and then applying various searching algorithms(uninformed and informed) techniques for the agents to navigate from the start to the end points of the maze. The searching algorithm used in our project include: Breadth-first search, Depth-first search, Greedy best-first search, and A* search algorithm. If the user wants to evaluate the paths using different searching algorithms at once then the performance of these algorithms is calculated on the basis of the time taken by them to find the end path.

Generating the Maze:

As mentioned before the maze generation takes place with the help of randomized depth-first search algorithm, where the grid cell formed can either be walls or path. The user has the option to input the dimensions of the maze so for example if they enter 30 so this means that the maze about to be generated would be of 30x30 grid of cells. The maze generated is dynamic in nature i.e. it can accommodate multiple agents simultaneously, which can move around the maze. To increase the complexity of the maze, additional paths are created by removing some walls, so that the agent makes use of every shortest possible path while solving the maze. The maze has the start(usually the top left corner) and the end(usually the bottom right corner) points to make sure that the maze is solvable.

Searching Algorithms:

Breadth-First Search:

The Breadth-First Search algorithm explores nodes level by level from the start to the end node, to make sure it is able to find the shortest path.

In our code:

- We use a queue to explore nodes in order of their distance from the start.
- We use a ‘visited’ array to track the nodes that have already been explored, so that there would not be any cycles formed.
- When the queue is not empty:
    - It dequeues the front node and its path, then checks all possible directions i.e. right, left, up, down.
    - Then for each direction, its next node is calculated and if the next node is the end node, then append the complete path to ‘agent_paths’ and return.
    - If the next node is not the end node and valid, then it is marked as visited and enqueued with the updated path

Advantage: BFS is guaranteed to find the shortest path if it exists.
Disadvantage: BFS can be memory-intensive for large mazes.

Depth-First Search:

The Depth-First Search algorithm explores the paths as far as possible in each branch before backtracking, which is memory efficient, but it does not guarantee the shortest path like BFS does.

In our code:

- We use a stack to explore the nodes and their paths as far as possible.
- We use ‘visited’ boolean array to keep a track of the visited nodes
- When the stack is not empty:
    - It pops the top node with its path and checks if the node is the end node or not, in case of end node, it appends the complete path to ‘agent_paths’ and returns.
    - Then all possible directions are checked up,down,left,right and for each of these directions, the next node is calculated.
    - If the next node is valid then it is marked as visited and pushed into the stack with the updated path.

Advantage: DFS uses less memory as compared to BFS
Disadvantage: DFS does not guarantees the shortest path.

Greedy Best-First Search:

The Greedy Best-First Search algorithm explores the nodes which appear to be the closest to the goal based on the heuristic function, and is able to find the solution faster, but it is not guaranteed to be the shortest path.

In our code:

- We use a priority queue where the starting node, its heuristic value, and its path is used to expand the nodes based on their heuristic distance from the goal.
- We use a ‘visited’ boolean array to keep a track of the visited nodes.
- While the priority queue is not empty:
    - It dequeues the node with the highest priority i.e. lowest heuristic value and checks if the node is the end node, if it is then it appends the complete path to ‘agents_paths’ and returns
    - Then all possible directions up,down,right,left are checked and for each of these directions their next node is calculated.
    - If the next node is valid, then it is marked as visited and enqueued with its heuristic value and the path is updated

Advantage: Greedy Best-First Search can be faster than BFS and DFS as it focuses on the closest possible path.
Disadvantage: Greedy Best-First Search does not guarantees to find the shortest path.

A* Search:

The A* Search algorithm makes use of the combined strengths of Dijkstra and Greedy Best-First algorithms and by using both the actual and the heuristic cost to reach the goal.

In our code:

- We use priority queue with the starting node along with its total cost(0) and its path.
- We use a dictionary ‘cost_so_far’ to keep a track of the cost to reach each node.
- While the priority queue is not empty:
    - It dequeues the node with the lowest total cost i.e. (actual cost + heuristic) and checks if the node is the end node, if it is then it appends the complete path to ‘agent_paths’ and returns.
    - Then all possible directions right,left,up,down are checked and then for each direction their next node is calculated along with its new cost.
    - It checks if the next node is valid, if it is then its cost and priority is updated if it has a cheaper path. Then it is marked as visited and enqueued with its updated total cost and path.

Advantage: A* guarantees to find the shortest path if the heuristic is admissible i.e. it never overestimates the true cost. (We have used the Manhattan distance in our heuristic function)
Disadvantage: A* may use significant memory and it heavily relies on the quality of the heuristic function.

Visualization and Performance:

Once the agents utilize the different searching algorithms to solve a maze, the paths which they found are visualized on the maze using Matplotlib. As each algorithm has a unique way to navigate through the maze and as different agents are used at the same time so to distinguish between their paths, different colors are used. This visualization allows the user to see how each algorithm found the path from the start to the end in the maze. The users have an option to compare all the searching algorithms in the program where they can see which algorithm performed better visually.

The timing for each algorithm is recorded during the searching process. The execution times are then compared to check the efficiency of each algorithm. The comparison is done using a bar chart, where the time for each algorithm to solve the same maze is displayed. The bar chart provides a quick visual representation of which algorithms were faster and which were slower.

Logic Reasoning with Prolog:

The project has included a function for logic reasoning using Prolog, which briefly describes the rules about node adjacency and queries these relationships for each agent's path. While we have tried to integrate this functionality to enhance the pathfinding analysis, it has not been possible to use this function in the current implementation. Maybe in the future development this feature could be further improved so it can be used for path validation and reasoning.

Conclusion:

In this project, we have implemented and comparious different searching algorithms to solve a randomized maze, including Breadth-First Search (BFS), Depth-First Search (DFS), Greedy Best-First Search, and A* search. The paths generated by these algorithms were visualized using Matplotlib,so the user can see a clear comparison of their performance. Moreover, each algorithm's execution time was recorded and then visualized using bar charts, resulting in providing insights into their efficiency.

The logic reasoning component using Prolog was included in the project to define and query relationships between maze nodes, but this feature was not made use of. And it provides an opportunity for future enhancements, where this could be further improved and used in the project.

Overall, the project offers visual and performance comparisons which provide valuable insights into selecting the appropriate algorithms for specific maze-solving tasks.

References:

- Russell, S. J. and Norvig, P. (2002). Artificial Intelligence: A Modern Approach (2nd Edition). Prentice Hall.
- https://www.swi-prolog.org/
- https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/
- https://en.wikipedia.org/wiki/Breadth-first_search
- https://en.wikipedia.org/wiki/Depth-first_search
- https://en.wikipedia.org/wiki/A*_search_algorithm#:~:text=A*%20is%20an%20informed%20search,shortest%20time%2C%20etc.).
- https://www.geeksforgeeks.org/a-search-algorithm/
- https://www.geeksforgeeks.org/greedy-best-first-search-algorithm/
- https://www.youtube.com/watch?v=ySN5Wnu88nE
- https://www.youtube.com/watch?v=dv1m3L6QXWs
- https://www.youtube.com/watch?v=7XVTnCrWDPY
- https://www.youtube.com/watch?v=3Xc3CA655Y4&t=3185s

- https://www.youtube.com/watch?v=5F9YzkpnaRw
- https://www.youtube.com/watch?v=GL28VbiFkD0
- https://numpy.org/doc/1.24/
- https://matplotlib.org/stable/index.html

Members:

Hamza
Redon
Ramil