REPORT

1. Introduction:

The project aims to test and compare the effectiveness of a variety of Search and MDP (Markov Decision Process) algorithms by deploying and evaluating them in a specially constructed maze environment. This study uses a maze generator that can create mazes of varying sizes and applies different Search and MDP algorithms on these mazes. The report goes into detail on how the maze generator was made, how the Search and MDP algorithms were run, and how the data were analyzed to determine how effective each algorithm was in comparison.

2. Implementation:

2.1 Maze Generation:

The python code from "xhoanss" is used for maze generation. Recursive backtracking is used in the generation of mazes by the Python software Generate_maze_mod.py, which presents a variety of obstacles for the algorithms. The following algorithms are implemented in Python: DFS (Dfs_mod.py), BFS (Bfs_mod.py), A* (A-star_mod.py), MDP Policy Iteration (MDP_policy.py), and MDP Value Iteration (MDP_value.py). Their performance is evaluated using metrics like path length, search length, and execution time (in csv), and visualize_results.py is used to display the results in easy-to-understand graphs.

2.2 Search Algorithms:

With an emphasis on the project's core, the implementation section painstakingly outlines each algorithm's development process. The codebase organization, algorithmic adjustments made specifically for solving mazes, and the incorporation of performance monitoring tools are covered. This part provides a clear understanding of the operational dynamics of each algorithm in the context of the maze using visual aids and code snippets.

2.2.1 Breadth First Search(BFS):

In unweighted graphs, such as mazes, BFS is a layer-by-layer algorithm that discovers the closest nodes before advancing to those farther away. This ensures the shortest path. Within the project, BFS is implemented by the Bfs_mod.py code, which tracks visited nodes to prevent cycles while allowing for a methodical exploration of mazes. Performance measures like path length and execution time provide information on how effective BFS is, especially in dense mazes where the shortest path is a priority.

In order to traverse mazes, BFS is designed to investigate the closest nodes in the search space before advancing to those that are farther away. It has features to load mazes from CSV files, visualize the maze and the path to the solution, and run the BFS algorithm. This method guarantees that, in the event that a solution is found, the BFS algorithm will locate the shortest path by expanding its exploration of all potential paths from the starting

point. Performance metrics that can be exported for additional examination include path length, search length, and execution time. These metrics are also tracked. This session highlights the usefulness of the BFS algorithm in applications that need optimal solutions by showcasing its ability to ensure the identification of the shortest path.

2.2.2 Depth First Search(DFS):

DFS is ideally suited for maze investigation, where all possible paths must be taken into consideration, since it investigates a branch as far as feasible before turning around. The Dfs_mod.py implementation demonstrates the usefulness of DFS in situations when determining any path—rather than just the shortest—is acceptable. The algorithm's graphic representation highlights its extensive search nature, as it tends to wander deep into the maze before exploring alternate pathways.

The DFS algorithm for maze exploration is implemented by the Depth-First Search (DFS) module (Dfs_mod.py). It has the ability to load mazes from CSV files, display the maze next to the path that was found, and run the DFS algorithm. The script gives priority to depth when searching, exploring a branch as far as it can go before turning around. This permits the exploration of every potential path, even though it does not always lead to the shortest one. In addition, it records performance parameters like as path length and execution time, providing information about how effective DFS is at navigating mazes. This session demonstrates the use of DFS in intricate labyrinth architectures and exhaustive search settings.

2.2.3 A – star:

The A* search algorithm, which is used in A-star_mod.py, effectively finds the shortest path by combining elements of BFS's thoroughness with heuristic-driven shortcuts. It prioritizes pathways that appear to be closer to the objective by using the Manhattan distance as a heuristic to calculate the cost of reaching the goal from a given node. The performance of A* depends on this harmony between heuristic judgment and inquiry, which makes it highly effective in varied maze configurations.

The A* module (A-star_mod.py) is made to use the A* search algorithm to solve mazes. It has the ability to load mazes from CSV files, visualize the labyrinth and the path leading to the solution, and determine the path by utilizing the Manhattan distance heuristic in conjunction with the A* algorithm. The script can read a maze layout from a CSV file, find the start and finish locations, and, if one exists, find the shortest path between them. Metrics related to execution, like path length, search length, and execution time, are noted and can be stored in a CSV file for further examination. This module shows how A* balances path cost with heuristic estimation to navigate mazes efficiently by giving priority to paths that seem to lead closer to the destination.

2.3 Markov Decision Process Algorithms:

2.3.1 Value Iteration:

Value Iteration uses dynamic programming to solve decision processes by iteratively updating state values until they converge, as seen in MDP_value.py. By estimating the predicted utility of each state's activities, this method generates an optimal policy based on the values of each state, which directs the agent towards the objective. The implementation demonstrates how the algorithm can modify methods in response to a growing comprehension of the layout and rewards of the maze.

2.3.2 Policy Iteration:

Policy Iteration, exemplified in MDP_policy.py, alternates between policy improvement and evaluation to maximize decision-making. With this method, initial policy hypotheses are refined into an ideal policy that determines the appropriate course of action in every state. The careful application demonstrates how policy iteration, by gradually improving the policy according to the anticipated results, may successfully traverse intricate mazes.

3. Results and Discussion:

Analyzing the performance graphs for the various algorithms across different maze sizes reveals notable trends. The execution time for BFS, DFS, and A* remains relatively low for smaller mazes but shows a significant increase as the maze size grows, with BFS displaying the steepest rise. This suggests BFS's exhaustive nature becomes a hindrance in larger mazes. A* search maintains a lower execution time compared to BFS and DFS, reflecting its heuristic efficiency.

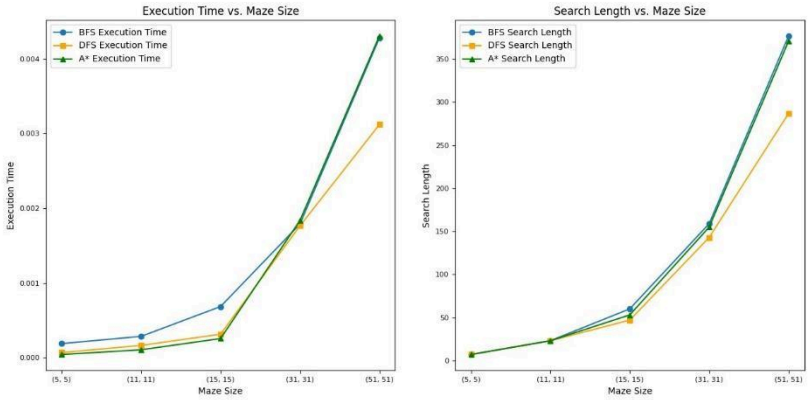

Fig 1: BFS vs DFS vs A-star

MDP algorithms exhibit a different pattern. Initially, for smaller mazes, MDP Value Iteration and Policy Iteration have a higher execution time, likely due to the overhead of evaluating and improving policies. However, as maze size increases, their execution time does not surge as dramatically as BFS, indicating a more scalable approach to larger mazes.

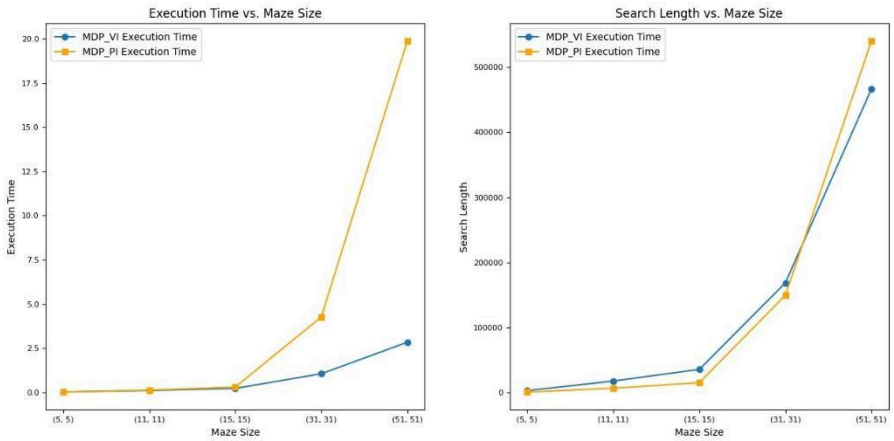

Fig 2: MDP Value Iteration vs MDP Policy Iteration

Regarding search length, BFS and DFS show minimal change across maze sizes, implying a consistent approach to pathfinding. A* search exhibits a slight increase in search length with larger mazes, which is expected due to its heuristic nature. MDP algorithms, while not primarily focused on search length, show an increase, suggesting that as the maze complexity increases, the policy's refinement process becomes more extensive.

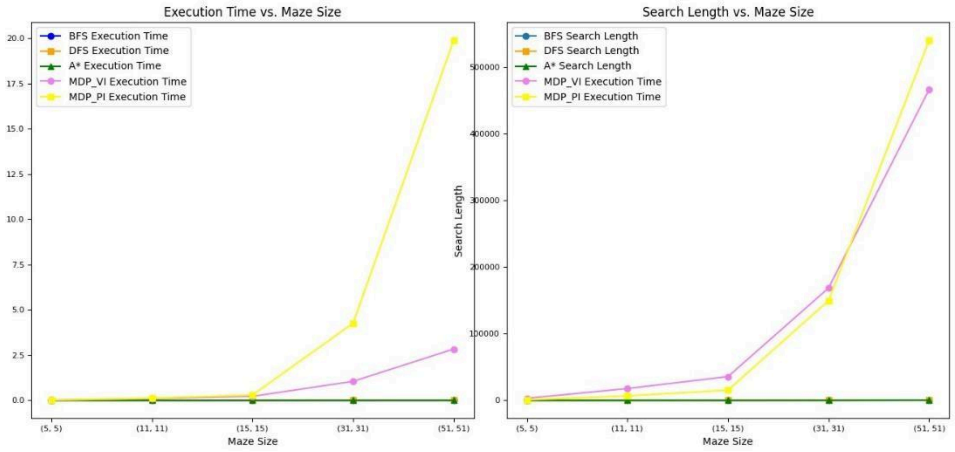

Fig 3: BFS vs DFS vs A-star vs MDP Value Iteration vs MDP Policy Iteration

4. Conclusion:

The information suggests that although BFS and DFS work well in smaller mazes, their scalability is constrained in more intricate and larger mazes. A* search shows up as a more effective substitute, providing a trade-off between computational resources and path optimality. Although MDP-based algorithms are slower at first because of their policy evaluation process, they have the potential to be useful in complicated decision-making contexts because of their ability to handle labyrinth complexity effectively. Future research can examine hybrid strategies that combine MDP frameworks and heuristic search to take use of both approaches' advantages in challenging, dynamic contexts.