CS7IS2: Artificial Intelligence Assignment 1

Nachiketh J (23337215) **26/02/2024**

1. Introduction-

In this research, we examine five distinct ways of creating and solving mazes: A*, value iteration, policy iteration, Depth-First Search (DFS), and Breadth-First Search (BFS). Our major objective is to evaluate and compare these approaches on various sizes of mazes to determine how fast and efficiently they can solve the puzzle. We are particularly interested in knowing how forward-looking decision-making procedures (value iteration and policy iteration)—also known as Markov Decision Processes, or MDPs—compare to search approaches (such as DFS, BFS, and A*). Through this process, we want to reveal the advantages and disadvantages of each method and explore the reasons behind the effectiveness of particular tactics in particular maze-solving scenarios.

2. Methodology

a. The Maze Generation

A modified version of Prim's technique, which is typically used to identify the minimal spanning tree of a graph, is utilized in the maze generating process. This algorithm's intrinsic unpredictability and capacity to guarantee a single, solvable maze devoid of isolated portions make it especially well-suited for maze formation. The method begins with a grid in which every cell is originally designated as a wall. It then chooses a cell at random to serve as the starting point and marks it as a section of the maze. After that, it adds the walls of this first cell to a list of walls that need to be processed. Choosing Prim's algorithm for maze generation offers several benefits:

- Uniqueness: Each generated maze is unique, providing a wide variety of challenges for the algorithms to solve.
- Complexity and Solvability: The algorithm naturally introduces complexity while ensuring that the maze has a viable solution, making it ideal for testing maze-solving algorithms.
- Adaptability: It can easily be modified to adjust the complexity of the maze, for example, by varying the rules for how walls are knocked down or how cells are added to the maze.

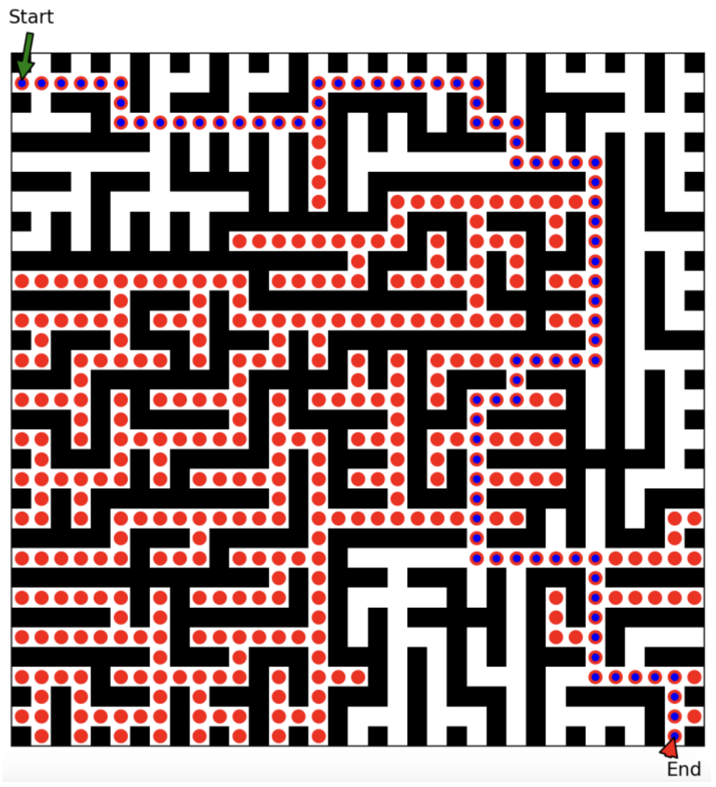
b. Algorithms implemented-

i. Search Algorithms

Depth First Search-

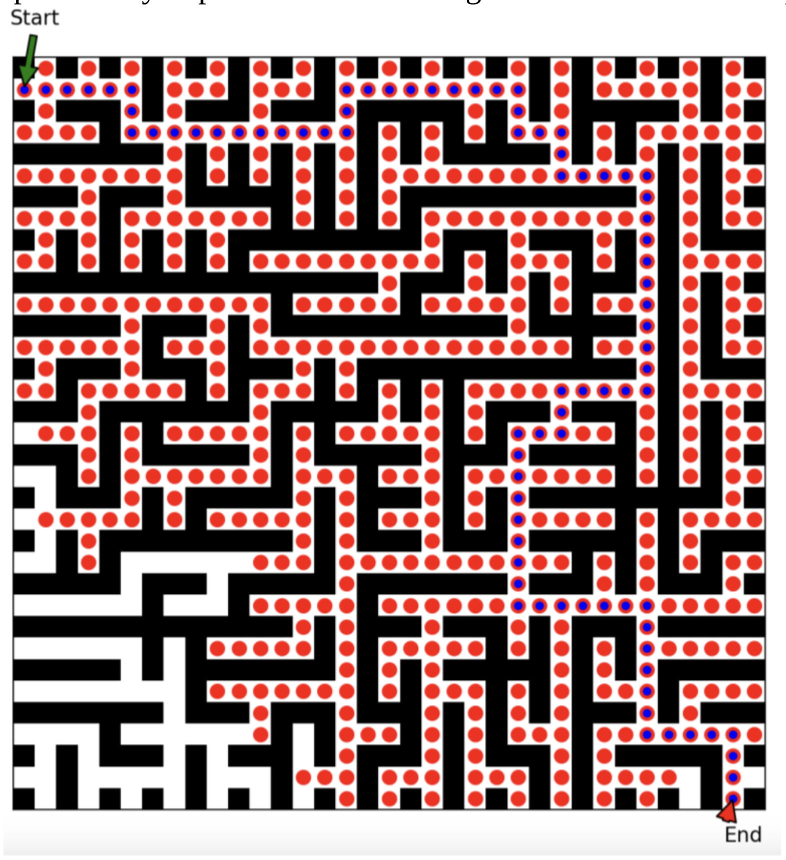
- The project utilizes Depth-First Search (DFS) with a recursive approach for solving mazes, starting from an entry point, and exploring adjacent, unvisited cells to delve deeper into the maze, using Python's call stack for path tracking without an explicit stack.
- Upon encountering a dead-end, the algorithm backtracks to explore alternative routes, continuing until the goal is achieved or all possible paths are explored, showcasing DFS's depth-seeking nature and its ability to navigate complex structures.
- While the method effectively demonstrates DFS's practical application in maze-solving, it is subject to the recursion depth limit of Python and may not guarantee the shortest path to the goal, highlighting both the strengths and limitations of this approach.

The visualization of the maze-solving process features red dots to mark nodes explored by the algorithm, illustrating the various paths considered during the search, and blue dots to indicate the optimal path identified, reflecting the most efficient route based on the algorithm's criteria for optimality.Start and end points of the maze are explicitly marked, enabling a clear understanding of the algorithm's entry and exit points, and the standardized representation facilitates a straightforward comparison of search patterns and solution efficiency across different algorithms.



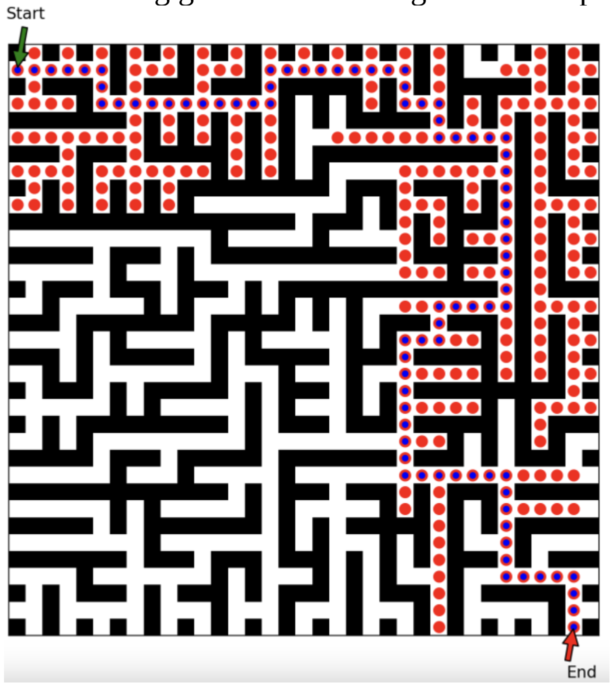
Breadth First Search-

- The project employs the Breadth-First Search (BFS) algorithm for maze navigation, characterized by its methodical exploration in expanding waves from the entry point, using a queue to manage exploration frontiers and ensure a level-by-level search that prioritizes the shortest path to the goal.
- By enqueuing adjacent, unvisited cells and marking visited ones, BFS systematically covers all possible paths, guaranteeing the identification of the optimal path in terms of length due to its iterative, breadth-first exploration strategy.
- BFS's simplicity and efficiency in finding the shortest path make it highly effective for maze-solving, highlighting its advantages over depth-first approaches by ensuring path optimality and demonstrating its practicality in puzzles where finding the shortest solution is paramount.



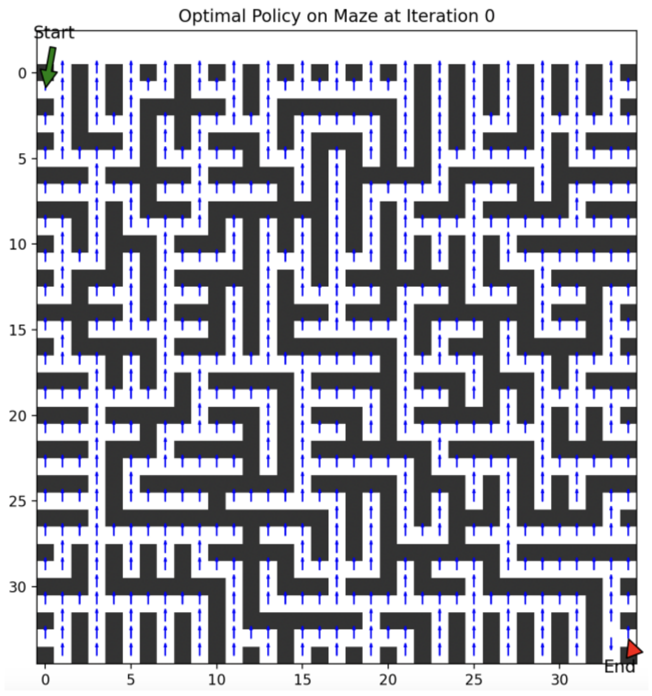
A star –

- The A* search algorithm is applied to maze solving, utilizing a combination of actual costs and heuristic estimates to efficiently find the shortest path, with priority given to paths showing promise based on their total estimated cost from start to goal.
- A priority queue orchestrates the exploration, ranking cells by their cumulative cost to ensure those with lower estimated costs are explored first, leveraging the Manhattan distance as a heuristic to suit the grid-based, non-diagonal movement of the maze.
- The algorithm meticulously tracks visited cells and maintains the most cost-effective path to each, enabling the swift reconstruction of the shortest path once the goal is reached, underscoring A*'s ability to balance systematic exploration with direct goal pursuit.
- This specific implementation of A* demonstrates its theoretical and practical strengths in maze solving, showcasing its adaptability and strategic efficiency in complex environments, and affirming its superiority in achieving goal-directed navigation with optimal results.



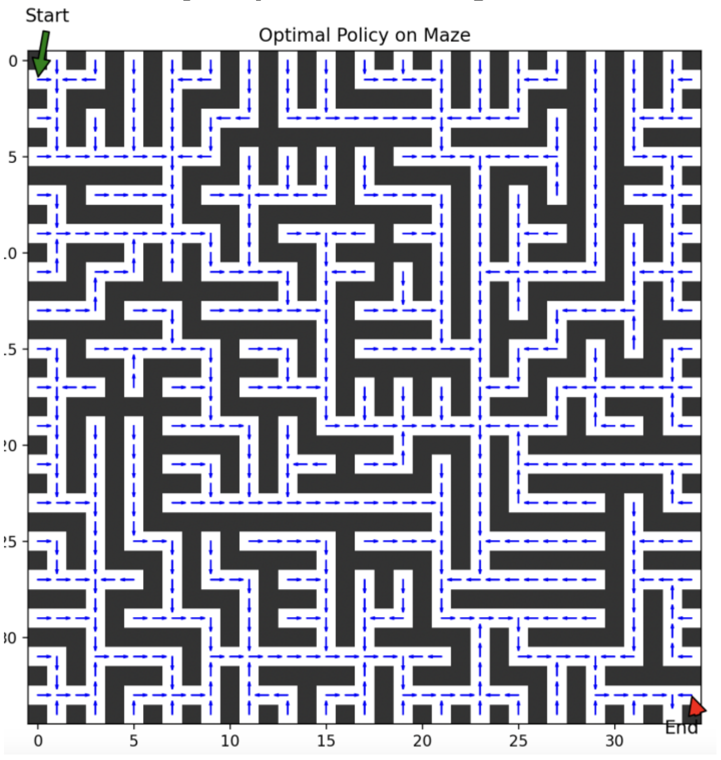
ii. Markov decision process Algorithms-

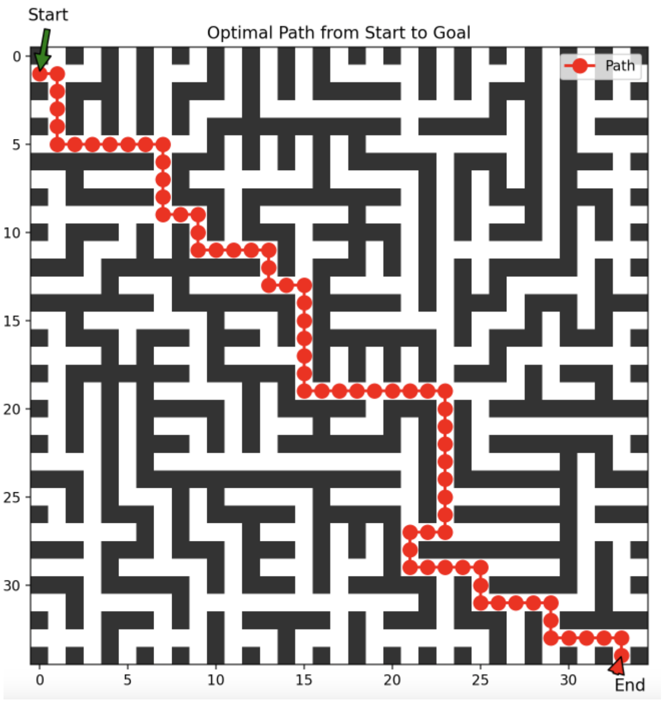
MDPs provide a framework for modeling decision-making where outcomes are partly random and partly under the control of a decision-maker. MDPs are characterized by states, actions, transition probabilities, and rewards.



1. Value iteration-

- The Value Iteration algorithm is adapted for maze solving within the framework of Markov Decision Processes (MDP), focusing on iteratively adjusting the estimated utility of each maze cell to find a balance between the costs of exploration and achieving the optimal exit path. This involves a reward structure that penalizes each move with -1, guiding the algorithm toward the most efficient goal route.
- A discount factor of 0.9 is selected to prioritize the importance of future rewards, ensuring that the algorithm's decision-making process is geared towards the overarching goal of finding the most advantageous path to the maze exit, illustrating the algorithm's foresight in valuing long-term gains over immediate outcomes.
- The algorithm employs a convergence threshold of 0.0000001 to determine when utility updates are minimal enough to signify that it has reached an optimal or near-optimal policy, highlighting the precision in the algorithm's refinement process and its effectiveness in navigating the complexity of maze-solving tasks.



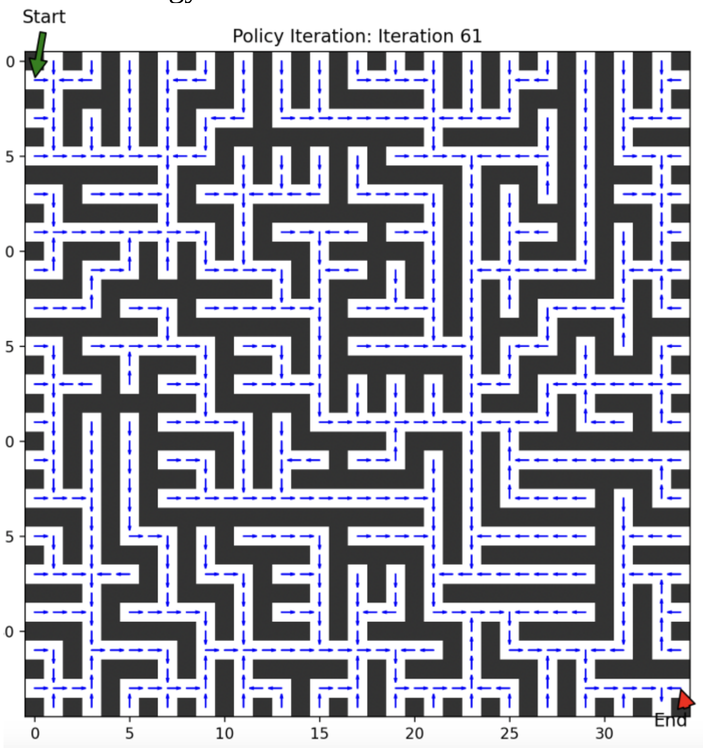


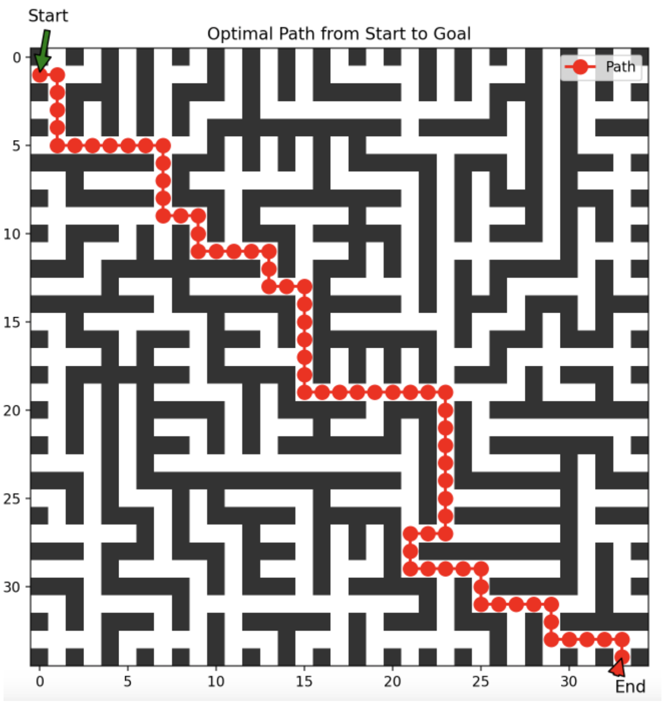
- The implementation includes a visualization component that periodically displays the evolving policy within the maze, offering intuitive insights into how the algorithm refines its strategy for navigation, thereby demystifying the abstract principles of Markov Decision Processes (MDPs) for a broader audience.
- The project's careful tuning of MDP parameters, such as the discount factor and convergence threshold, combined with a penalty for each move, showcases the algorithm's efficiency in identifying the shortest path through the maze while also highlighting the complex decision-making inherent in the value iteration process.
- By illustrating the algorithm's flexibility in adjusting its navigational strategy and the effectiveness of its parameter calibration, the project not only proves the value of Value Iteration for maze-solving tasks but also emphasizes its wide-ranging potential and adaptability in addressing diverse problem-solving scenarios, showcasing the deep relevance and applicability of its theoretical underpinnings in practical contexts.

2. Policy iteration-

- The project implements Policy Iteration, a critical Markov Decision Process (MDP) algorithm, for maze solving, starting with an arbitrary policy and iteratively refining it through policy evaluation (calculating the utility of each state under the current policy) and policy improvement (updating the policy to maximize expected utility based on evaluated state utilities).

- The iterative cycle of policy evaluation and improvement ensures the algorithm refines its strategy until reaching a stable policy, effectively finding the optimal or near-optimal path through the complex maze environment by balancing the consideration of immediate and future rewards with a discount factor of 0.9.
- Unlike some algorithms that require a specific convergence criterion, Policy Iteration's convergence to an optimal policy is naturally indicated by the stabilization of the policy itself, demonstrating the algorithm's efficiency and inherent mechanism for determining when the optimal navigational strategy has been achieved.





- This implementation distinguishes itself by incorporating a visualization component that provides periodic updates on the policy's evolution within the maze, offering tangible insights into the algorithm's strategic refinement and convergence towards an optimal navigation strategy, like the approach taken with Value Iteration.
- By carefully selecting MDP parameters, especially the discount factor, and applying Policy Iteration, the project not only demonstrates the algorithm's capability to efficiently find the most effective paths through mazes but also its ability to elucidate and visualize the intricate decision-making involved in strategic planning and execution in constrained settings.
- The incorporation of Policy Iteration, complemented by strategic parameter tuning and visual feedback, exemplifies the algorithm's robustness, adaptability, and its capacity to provide clear insights into complex decision-making processes, affirming its significant utility in solving challenges that require deep strategic analysis and thoughtful execution.

iii. Distinctions and Applications

Search algorithms are generally more straightforward and are used when the environment is fully observable, and the goal is to find a path.
MDP algorithms are used in situations where there's uncertainty in the outcome of actions and decisions need to be made over a series of steps to achieve a goal.

3. Result

The performance analysis of search and MDP algorithms in maze-solving tasks is presented in pivot tables that clearly delineate the impact of maze size on each algorithm's efficiency. This section will discuss the comparative results based on execution time, average nodes explored, and iterations taken for convergence.

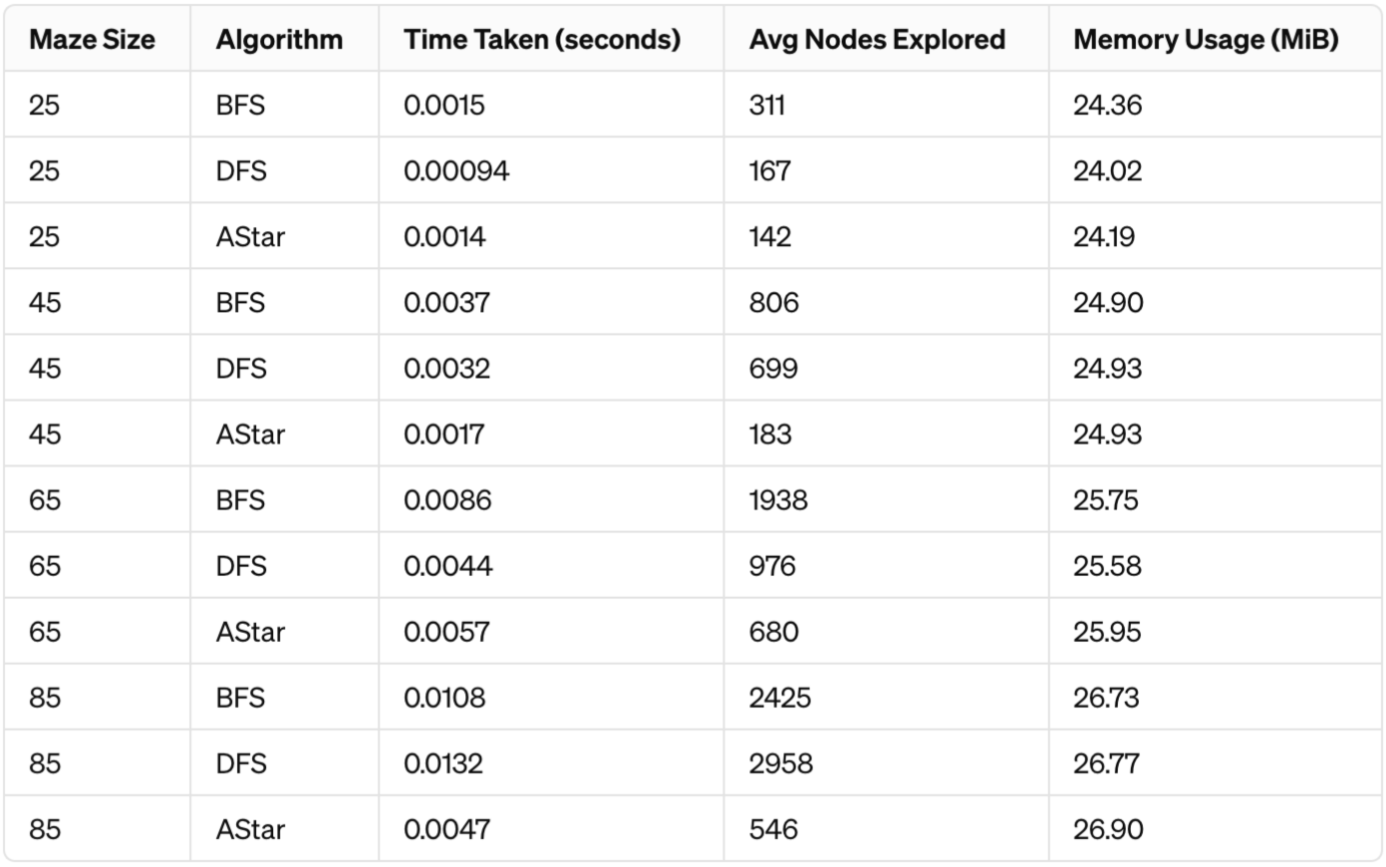
In the evaluation of maze-solving algorithms, three metrics were paramount: time taken, nodes explored or iterations, and memory usage.
1. Execution Time: The pivot tables show how maze size affects the execution time for each algorithm. Larger mazes generally increase the execution time due to the greater number of possible paths to evaluate. Algorithms with lower execution times are preferred for real-time applications where rapid response is essential.

2. Average Nodes Explored / Iterations: This metric indicates the algorithm's efficiency in exploring the maze. Algorithms that explore fewer nodes or take fewer iterations to converge are considered more efficient as they can find the shortest path without unnecessary exploration. This is critical in environments where computational resources are limited, or the cost of computation is high.

3. Memory Usage: The tables likely compare how each algorithm scales with maze size in terms of memory consumption. Efficient memory usage is key for deploying algorithms in systems with limited memory capacity. Algorithms with lower memory requirements are more scalable and can be applied to larger problems or run on less powerful machines.

4. Impact of Maze Size on Efficiency: The pivot tables would show a trend of how each of these metrics changes as the size of the maze increases. Efficient algorithms would show a slower growth in time, nodes explored, and memory as maze size increases. This would be indicative of good scalability, an important factor for practical applications where the environment can be large and complex.

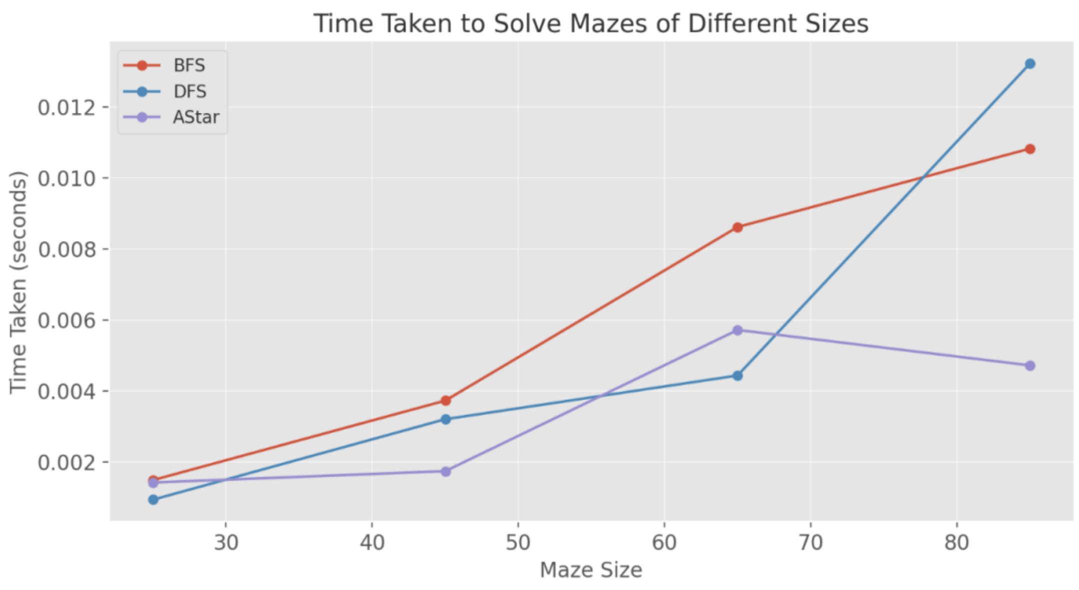
a. Search Algorithms Performance Metrics



Maze Size	Algorithm	Time Taken (seconds)	Avg Nodes Explored	Memory Usage (MiB)
25	BFS	0.0015	311	24.36
25	DFS	0.00094	167	24.02
25	AStar	0.0014	142	24.19
45	BFS	0.0037	806	24.90
45	DFS	0.0032	699	24.93
45	AStar	0.0017	183	24.93
65	BFS	0.0086	1938	25.75
65	DFS	0.0044	976	25.58
65	AStar	0.0057	680	25.95
85	BFS	0.0108	2425	26.73
85	DFS	0.0132	2958	26.77
85	AStar	0.0047	546	26.90

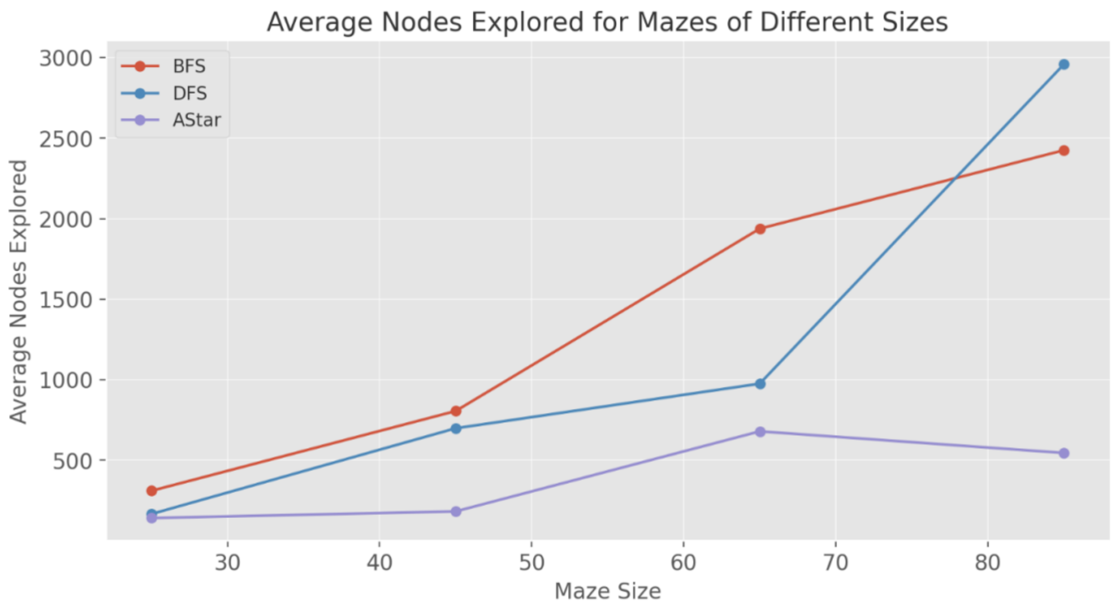
The provided table offers insights into the performance of three different search algorithms—Breadth-First Search (BFS), Depth-First Search (DFS), and A* (AStar)—across maze sizes of 25, 45, 65, and 85. We will discuss the performance based on three metrics: time taken, average nodes explored, and memory usage.

1. Time Taken:



   a. BFS generally takes more time than DFS and AStar, especially as the maze size increases. This could be due to BFS exploring all neighboring nodes level by level, which can be time-consuming, especially in larger mazes where the solution path might not be directly proportional to the size of the maze.
   b. DFS is faster than BFS in larger mazes, which could be due to its aggressive strategy of exploring as far as possible along each branch before backtracking. This can sometimes result in faster solutions in mazes where the goal can be reached without exploring all nodes, but it's not guaranteed.
   c. AStar is consistently the fastest across all maze sizes. This is expected as AStar uses a heuristic to guide its search, focusing on paths that are more likely to lead to the goal. This results in a more directed and generally efficient search, reducing the time to find the goal.

2. Average Nodes Explored:



   a. BFS explores many nodes, often more than the other two algorithms, because it systematically explores neighbor nodes at each depth before moving on to nodes at the next level.
   b. DFS explores fewer nodes than BFS, which aligns with its depth-first strategy. However, it can be inefficient in some cases, such as when the goal is closer to the start point but along a different path than the one chosen by DFS.
   c. AStar explores the fewest nodes on average. Its heuristic effectively guides it toward the goal, preventing it from exploring irrelevant paths, which is evident from the consistently lower number of explored nodes compared to BFS and DFS.

3. Memory Usage:
   a. The memory usage for all three algorithms is relatively similar across different maze sizes, with a slight increase as the maze size grows. This indicates that memory usage isn't significantly affected by the search strategy itself but rather by the size of the data structures (like the maze representation and node storage) which scale with the size of the maze.
   b. All three algorithms show a trend of increased memory usage with maze size, but this increase is minimal, suggesting that the underlying data structures used by these algorithms are not significantly memory intensive.

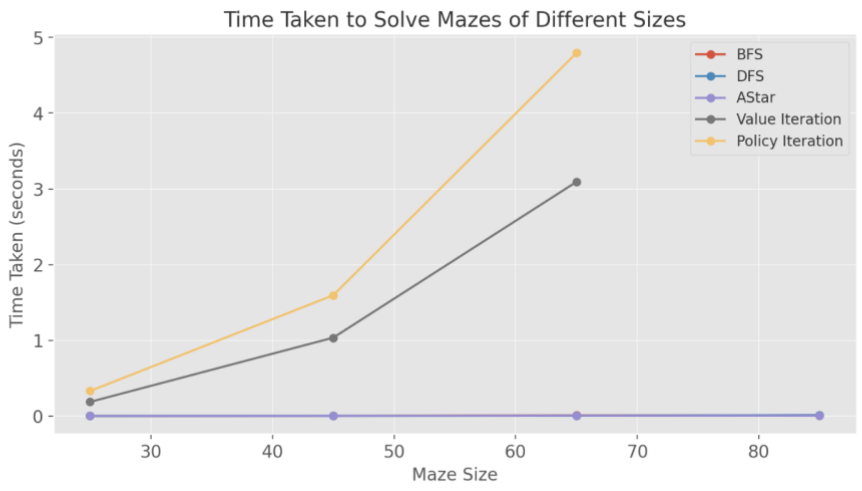
In summary, AStar demonstrates superior performance in terms of both speed and efficiency, as it finds the goal the fastest and with the least number of explored nodes. BFS, while comprehensive, is slower and more exhaustive, which makes it less efficient for larger mazes. DFS can be faster than BFS but is less consistent in performance, as it might get trapped exploring long paths that don't lead to the goal. The memory usage does not vary significantly between the algorithms, implying that the choice between these algorithms should be based on the time complexity and the nature of the maze rather than on memory constraints.

b. MDP Algorithms Performance Metrics

Maze Size	Algorithm	Time Taken (seconds)	Iterations	Memory Usage (MiB)
25	Value Iteration	0.186	60	78.68
25	Policy Iteration	0.330	53	74.99
45	Value Iteration	1.034	106	68.01
45	Policy Iteration	1.595	88	67.87
65	Value Iteration	3.092	153	68.05
65	Policy Iteration	4.795	151	68.18

The provided table shows the performance comparison between Value Iteration and Policy Iteration algorithms across different maze sizes (25, 45, 65). These two algorithms are commonly used in reinforcement learning for solving Markov Decision Processes (MDPs). Let's analyze the performance based on the time taken and the number of iterations required for convergence.

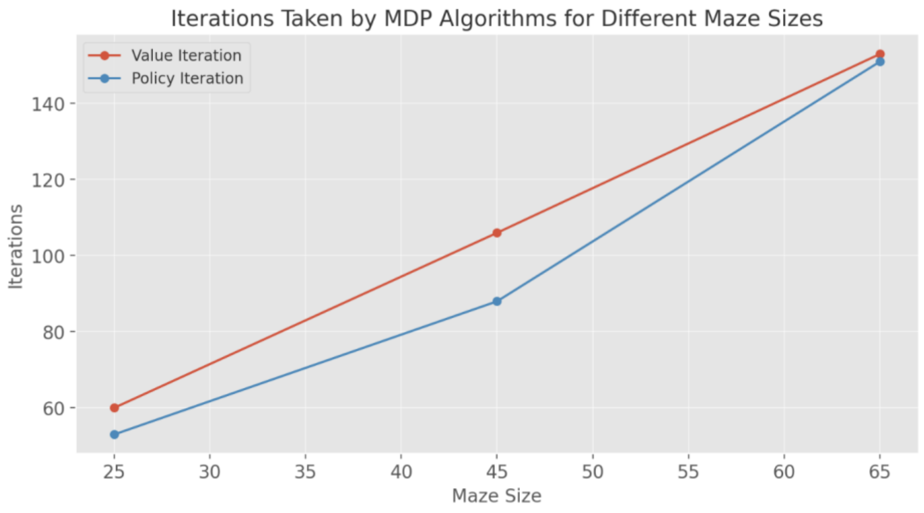
1. Time Taken:



All the search algorithms are overlapping each other.

a. Value Iteration shows a consistent increase in time taken as the maze size increases, which is expected since the number of states to iterate over increases with the size of the maze.
b. Policy Iteration also shows an increase in time with maze size, but it does not consistently take less time than Value Iteration despite sometimes requiring fewer iterations. This could be due to the overhead of the policy evaluation step, which can be time-consuming.

2. Iterations Taken:



a. For the smallest maze size (25), Value Iteration requires more iterations than Policy Iteration to converge. However, as the maze size increases, this trend does not hold; for example, Value Iteration requires fewer iterations for a size 45 maze.
b. Policy Iteration requires fewer iterations for maze size 25, but more iterations for larger sizes (45 and 65). This suggests that for smaller problems, Policy Iteration might refine a policy more quickly, but as the problem size increases, the advantage may not persist.

3. Memory Usage:
a. Value Iteration tends to use more memory than Policy Iteration, especially in the smallest maze size. This could be because Value Iteration maintains a complete copy of the value function across iterations, which may consume more memory.
b. For larger maze sizes, the memory usage difference between the two algorithms becomes less pronounced, suggesting that the memory overhead is more related to the maze size rather than the algorithmic differences.

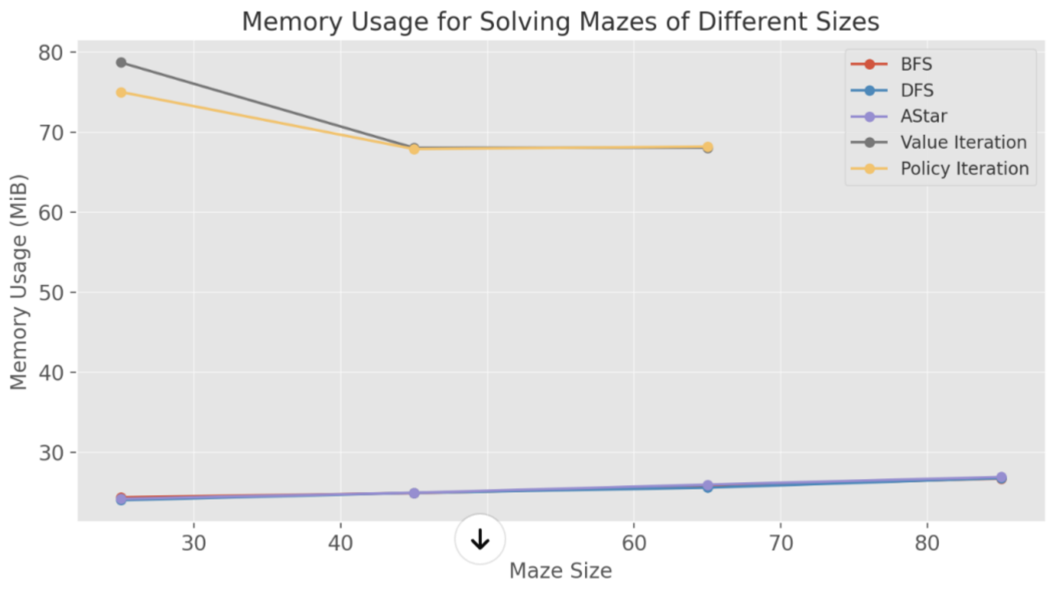
Observations:

- Value Iteration appears to be more time-efficient for larger mazes, despite sometimes taking more iterations to converge. This suggests that each iteration of Value Iteration may be computationally cheaper than each policy evaluation step in Policy Iteration.
- Policy Iteration may be a better choice for smaller problems where the number of iterations is a more significant factor in overall computation time than the complexity of each iteration.
- Memory Usage is generally higher for Value Iteration, indicating that its data structures might be more memory intensive. However, the memory usage for both algorithms increase similarly with maze size, showing that the main factor in memory consumption is related to the state space rather than the specific algorithm.

In conclusion, the choice between Value Iteration and Policy Iteration should consider both the size of the problem and the computational resources available. Value Iteration seems preferable for larger mazes where time efficiency is paramount, while Policy Iteration might be advantageous for smaller problems where fewer iterations can lead to faster convergence. The memory usage difference, while present, may not be the deciding factor in choosing between these two algorithms for these problem sizes.

4. Comparative Analysis and Discussions

To conduct a comprehensive analysis, let's break down the comparisons into three sections, as requested.



a. Comparison Between Different Search Algorithms:

The search algorithms in question are Breadth-First Search (BFS), Depth-First Search (DFS), and A* (AStar).

1. BFS is usually thorough as it explores all possible nodes at each depth level before moving deeper. It often finds the shortest path but can be slow and memory-intensive, as evidenced by its higher memory usage and time taken, especially in larger mazes.

2. DFS explores as deep as possible along each branch before backtracking. It can be faster, particularly in mazes where the goal is closer to the start in terms of depth. However, it may not find the shortest path and can be inefficient in terms of path length, as seen in the lower average nodes explored, which suggests it may take less optimal paths.

3. AStar uses heuristics to guide its search, making it generally faster and more efficient, as shown by the lower time taken and fewer nodes explored across maze sizes. Its heuristic approach directly aims for the goal, avoiding exploration of unnecessary paths.

In conclusion, for most practical pathfinding problems where the shortest path is desired, AStar is the best choice due to its efficiency in time and path optimization. However, when memory is a concern, DFS may be considered, albeit with the trade-off of potentially less optimal paths.

b. Comparison Between Different MDP Algorithms:

The MDP algorithms are Value Iteration and Policy Iteration.

1. Value Iteration is generally faster, particularly for larger problems. It directly updates the values of states based on the maximum expected utility, which, despite the higher iteration counts in some cases, tends to converge faster in terms of real-time.

2. Policy Iteration tends to be slower, potentially due to the two-step process of policy evaluation and improvement. However, it can require fewer iterations to converge, suggesting that while each iteration

is more computationally intensive, it may make more significant progress toward the optimal policy with each step.

In conclusion, Value Iteration should be preferred in larger state spaces where computational time is critical, while Policy Iteration could be advantageous in smaller problems or when iteration count rather than time is the primary concern.

c. Comparison Between Search and MDP Algorithms:

1. When comparing search algorithms (which are used for pathfinding) with MDP algorithms (which are used for decision-making under uncertainty), several key differences are apparent:

2. Purpose: Search algorithms are designed to find a path from a start point to a goal in the least possible time or with the least number of steps. In contrast, MDP algorithms aim to find an optimal policy that maximizes the expected utility over time, taking into account all possible future states.

3. Performance Metrics: For search algorithms, the primary metrics are the number of nodes explored and the time taken to find the goal. For MDP algorithms, the metrics are the number of iterations to convergence and the expected utility of the resulting policy.

4. Complexity and Scalability: AStar, being the most efficient search algorithm, still has to search through the state space to find the goal. MDP algorithms, on the other hand, consider the entire model of the environment, which can be computationally intensive as the state space grows, but they are better suited to situations where the agent must make a series of decisions that influence future states.

5. Memory Usage: Memory usage is relatively similar among search algorithms, slightly increasing with maze size. MDP algorithms tend to use more memory, which can be attributed to the storage of the utility values or policy for every state.

In conclusion, for deterministic pathfinding problems, search algorithms, particularly AStar, are recommended for their effectiveness, while for problems involving stochastic environments and long-term planning, MDP algorithms such as Value Iteration provide a more comprehensive solution.

5. Conclusions

The investigation into maze-solving algorithms concludes that A* is the superior choice for scenarios prioritizing speed and computational efficiency. Its performance across various maze sizes showcases its scalability and ability to navigate complex environments swiftly while minimizing resource usage. This positions A* as an ideal algorithm for real-time applications, such as autonomous navigation systems, where quick decision-making is crucial.

However, when solution optimality is paramount, MDP algorithms like Value Iteration and Policy Iteration become relevant despite their computational intensity. They excel in deterministic environments where each decision's impact is significant, such as in strategic game AI or complex logistical planning. Future studies could expand on this work by exploring hybrid approaches that combine the heuristic focus of A* with the thoroughness of MDP methods, potentially leading to algorithms that can adaptively balance between speed and precision based on dynamic environmental factors.

Appendix-

Code for search algorithms-

```
import numpy as np
import random
import matplotlib.pyplot as plt
from collections import deque
from queue import PriorityQueue
import time
import sys
from memory_profiler import memory_usage
from statistics import mean




def generate_maze(size=15):
    if size % 2 == 0:
        size += 1  # Ensure size is odd for maze generation

    # Create a grid where walls are True and cells are False
    maze = np.full((size, size), True)
    # Start with a random cell
    start_x, start_y = (random.randrange(1, size, 2), random.randrange(1, size, 2))
    maze[start_x, start_y] = False

    # List of walls to process, start with walls of the first cell
    walls = []
    for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
        if 0 <= start_x+dx < size and 0 <= start_y+dy < size:
            walls.append((start_x+dx, start_y+dy))

    while walls:
        # Choose a random wall
        wall = random.choice(walls)
        walls.remove(wall)

        # Find the cells that this wall divides
        x, y = wall
        if x % 2 == 0:  # Vertical wall
            cell1 = (x-1, y)
            cell2 = (x+1, y)
        else:  # Horizontal wall
            cell1 = (x, y-1)
            cell2 = (x, y+1)

        # If the cells that the wall divides are in different states, remove the wall
        if 0 <= cell2[0] < size and 0 <= cell2[1] < size and maze[cell1] != maze[cell2]:
            maze[wall] = False
            new_open_cell = cell2 if maze[cell1] == False else cell1
            maze[new_open_cell] = False

            # Add the neighboring walls of the new cell to the wall list
            for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                nx, ny = new_open_cell[0]+dx, new_open_cell[1]+dy
                if 0 <= nx < size and 0 <= ny < size and maze[nx, ny] == True:
                    if (nx, ny) not in walls:
                        walls.append((nx, ny))

    # Create entrance and exit
    maze[1][0] = False  # Entrance
    maze[size-2][size-1] = False  # Exit

    return maze
def bfs_solve_maze_with_bool(maze, start, end, vis=False):
```

```python
start_time = time.time()  # Start timing

queue = deque([start])
visited = set()
path = {}
directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
nodes_explored = 0  # Initialize node counter

if vis:
    # Set up the plot
    plt.figure(figsize=(8, 8))
    plt.imshow(maze, cmap='binary')
    plt.xticks([]), plt.yticks([])
    plt.title("Breadth First Search")  # Title for the plot


    # Annotate start and end
    start_annotation_x = start[1] - 0.5 if start[1] > 0 else start[1] + 0.5
    start_annotation_y = start[0] - 3  # Adjusting this line to correctly define start_annotation_y
    plt.annotate('Start', xy=(start[1], start[0]), xytext=(start_annotation_x, start_annotation_y),
                 arrowprops=dict(facecolor='green', shrink=0.05), fontsize=12, ha='center')

    # Annotate end outside the maze with an arrow pointing towards it
    end_annotation_x = end[1] + 0.5 if end[1] < maze.shape[1] - 1 else end[1] - 0.5
    end_annotation_y = end[0] + 2  # Adjusting for a more consistent position relative to the maze boundary
    plt.annotate('End', xy=(end[1], end[0]), xytext=(end_annotation_x, end_annotation_y),
                 arrowprops=dict(facecolor='red', shrink=0.05), fontsize=12, ha='center')
while queue:
    current = queue.popleft()
    nodes_explored += 1  # Increment nodes explored when a node is processed

    if vis:
        plt.plot(current[1], current[0], 'ro', markersize=8)
        plt.pause(0.0001)  # Pause to display the update

    if current == end:
        # Reconstruct the path from end to start
        solution_path = []
        while current != start:
            solution_path.append(current)
            current = path[current]
        solution_path.append(start)
        solution_path.reverse()
```

```
            end_time = time.time()  # End timing
            if vis:
                # Visualize the final path
                for step in solution_path:
                    plt.plot(step[1], step[0], 'bo', markersize=4)
                    plt.pause(0.0001)
                plt.show()

            return solution_path, end_time - start_time, nodes_explored  # Return path, time taken, and nodes explored

        if current not in visited:
            visited.add(current)
            for dx, dy in directions:
                next_cell = (current[0] + dx, current[1] + dy)
                if 0 <= next_cell[0] < maze.shape[0] and 0 <= next_cell[1] < maze.shape[1] and maze[next_cell] == False:
                    if next_cell not in visited:
                        queue.append(next_cell)
                        path[next_cell] = current

    end_time = time.time()  # End timing
    if vis:
        plt.show()
    return None, end_time - start_time, nodes_explored


def dfs_solve_maze_with_bool(maze, start, end, vis=False):
    start_time = time.time()
    stack = [start]
    visited = set()
    path = {}
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    nodes_explored = 0
    solution_path = []

    if vis:
        # Set up the plot if visualization is enabled
        plt.figure(figsize=(8, 8))
        plt.imshow(maze, cmap='binary')
        plt.xticks([]), plt.yticks([])
        plt.title("Depth First Search")
        plt.annotate('Start', xy=(start[1], start[0]), xytext=(start[1] - 0.5, start[0] - 3),
                     arrowprops=dict(facecolor='green', shrink=0.05), ha='center')
        plt.annotate('End', xy=(end[1], end[0]), xytext=(end[1] + 0.5, end[0] + 3),
                     arrowprops=dict(facecolor='red', shrink=0.05), ha='center')
```

```python
    while stack:
        current = stack.pop()
        if vis:
            plt.plot(current[1], current[0], 'ro', markersize=8)
            plt.pause(0.0001)

        if current == end:
            while current in path:
                solution_path.append(current)
                current = path[current]
            solution_path.append(start)
            solution_path.reverse()
            end_time = time.time()

            if vis:
                for step in solution_path:
                    plt.plot(step[1], step[0], 'bo', markersize=4)
                    plt.pause(0.0001)
                plt.show()

            return solution_path, end_time - start_time, nodes_explored

        if current not in visited:
            visited.add(current)
            nodes_explored += 1
            for dx, dy in directions:
                next_cell = (current[0] + dx, current[1] + dy)
                if 0 <= next_cell[0] < maze.shape[0] and 0 <= next_cell[1] < maze.shape[1] and maze[next_cell] == False:
                    if next_cell not in visited:
                        stack.append(next_cell)
                        path[next_cell] = current

    end_time = time.time()
    if vis:
        plt.show()
    return None, end_time - start_time, nodes_explored

def heuristic(a, b):
    return abs(b[0] - a[0]) + abs(b[1] - a[1])


def a_star_solve_maze_with_bool(maze, start, end, vis=False):
    start_time = time.time()
```

```python
open_set = PriorityQueue()
open_set.put((0, start))
came_from = {}
g_score = {start: 0}
f_score = {start: heuristic(start, end)}
nodes_explored = 0
directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]


if vis:
    plt.figure(figsize=(8, 8))
    plt.imshow(maze, cmap='binary')
    plt.xticks([]), plt.yticks([])
    plt.title("A Star Search")
    plt.annotate('Start', xy=(start[1], start[0]), xytext=(start[1] - 0.5, start[0] - 3),
                 arrowprops=dict(facecolor='green', shrink=0.05), ha='center')
    plt.annotate('End', xy=(end[1], end[0]), xytext=(end[1] + 0.5, end[0] + 3),
                 arrowprops=dict(facecolor='red', shrink=0.05), ha='center')

while not open_set.empty():
    current = open_set.get()[1]
    nodes_explored += 1

    if vis:
        plt.plot(current[1], current[0], 'ro', markersize=8)
        plt.pause(0.0001)

    if current == end:
        solution_path = []
        while current in came_from:
            solution_path.append(current)
            current = came_from[current]
        solution_path.append(start)
        solution_path.reverse()
        end_time = time.time()

        if vis:
            for step in solution_path:
                plt.plot(step[1], step[0], 'bo', markersize=4)
                plt.pause(0.0001)
            plt.show()

        return solution_path, end_time - start_time, nodes_explored
        for dx, dy in directions:
            neighbor = (current[0] + dx, current[1] + dy)
            if 0 <= neighbor[0] < maze.shape[0] and 0 <= neighbor[1] < maze.shape[1] and maze[neighbor] == 0:
                tentative_g_score = g_score[current] + 1
                if neighbor not in g_score or tentative_g_score < g_score[neighbor]:
                    came_from[neighbor] = current
                    g_score[neighbor] = tentative_g_score
                    f_score[neighbor] = tentative_g_score + heuristic(neighbor, end)
                    open_set.put((f_score[neighbor], neighbor))


    end_time = time.time()
    if vis:
        plt.show()
    return None, end_time - start_time, nodes_explored # Return None and time taken if no path found


# Code for performance comparision of different algorithms on same maze
def main(arg1=35):
    maze_sizes=[25,45,65,85]

    for maze_size in maze_sizes:
        bfstimes=list()
        bfs_nodes=list()
        bfs_mem=list()
        dfs_times=list()
        dfs_nodes=list()
        dfs_mem=list()
        astartimes=list()
        astar_nodes=list()
        astar_mem=list()
        for i in  range(3):
            maze = generate_maze(maze_size)  # Adjust size as needed
            start_point = (1, 0)
            end_point = (maze.shape[0] - 2, maze.shape[1] - 1)
            def wrap_bfs():
                solution_path, bfs_time, bfs_explore = bfs_solve_maze_with_bool(maze, start_point, end_point)
                bfstimes.append(bfs_time)
                bfs_nodes.append(bfs_explore)
            def wrap_dfs():
                dfs_solution_path, dfs_time, dfs_explore = dfs_solve_maze_with_bool(maze, start_point, end_point)
                dfs_times.append(dfs_time)
                dfs_nodes.append(dfs_explore)
            def wrap_astar():
                astar, atime, astar_explore = a_star_solve_maze_with_bool(maze, start_point, end_point)
                astartimes.append(atime)
                astar_nodes.append(astar_explore)
            bfs_mem.append(mean(memory_usage(proc=wrap_bfs)))
            dfs_mem.append(mean(memory_usage(proc=wrap_dfs)))
            astar_mem.append(mean(memory_usage(proc=wrap_astar)))
        print("time taken by BFS for size "+str(maze_size)+" : "+str(sum(bfstimes) / len(bfstimes))+" seconds & Avg nodes explored
are "+str(int(sum(bfs_nodes) / len(bfs_nodes))))
        print(f"Memory usage of BFS: {str(sum(bfs_mem) / len(bfs_mem))} MiB")
        print("time taken by DFS for size "+str(maze_size)+" : "+str(sum(dfs_times) / len(dfs_times))+" seconds & Avg nodes explored
are "+str(int(sum(dfs_nodes) / len(dfs_nodes))))
        print(f"Memory usage of DFS: {str(sum(dfs_mem) / len(dfs_mem))} MiB")
        print("time taken by Astar for size "+str(maze_size)+" : "+str(sum(astartimes) / len(astartimes))+" seconds & Avg nodes
explored are "+str(int(sum(astar_nodes) / len(astar_nodes))))
        print(f"Memory usage of AStar: {str(sum(astar_mem) / len(astar_mem))} MiB")
    time.sleep(2)


    #code for visualisation of different algorithms on same maze
    maze_size = int(arg1)
    maze = generate_maze(maze_size)

    start_point = (1, 0)
    end_point = (maze_size - 1, maze_size - 2)

    solution_path = bfs_solve_maze_with_bool(maze, start_point, end_point,True)
    dfs_solution_path = dfs_solve_maze_with_bool(maze, start_point, end_point, True)
    astar = a_star_solve_maze_with_bool(maze, start_point, end_point,True)

if __name__ == '__main__':
    # This code won't run if this file is imported.
    args = sys.argv[1:]

    main(*args)
```

code for Mdp algorithms-

```
import numpy as np
import random
import matplotlib.pyplot as plt
import time
import sys
from memory_profiler import memory_usage
from statistics import mean
```

```python
def generate_maze(size=15):
    if size % 2 == 0:
        size += 1  # Ensure size is odd for maze generation

    # Create a grid where walls are True and cells are False
    maze = np.full((size, size), True)

    # Start with a random cell
    start_x, start_y = (random.randrange(1, size, 2), random.randrange(1, size, 2))
    maze[start_x, start_y] = False

    # List of walls to process, start with walls of the first cell
    walls = []
    for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
        if 0 <= start_x+dx < size and 0 <= start_y+dy < size:
            walls.append((start_x+dx, start_y+dy))

    while walls:
        # Choose a random wall
        wall = random.choice(walls)
        walls.remove(wall)

        # Find the cells that this wall divides
        x, y = wall
        if x % 2 == 0:  # Vertical wall
            cell1 = (x-1, y)
            cell2 = (x+1, y)
        else:  # Horizontal wall
            cell1 = (x, y-1)
            cell2 = (x, y+1)

        # If the cells that the wall divides are in different states, remove the wall
        if 0 <= cell2[0] < size and 0 <= cell2[1] < size and maze[cell1] != maze[cell2]:
            maze[wall] = False
            new_open_cell = cell2 if maze[cell1] == False else cell1
            maze[new_open_cell] = False

            # Add the neighboring walls of the new cell to the wall list
            for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                nx, ny = new_open_cell[0]+dx, new_open_cell[1]+dy
                if 0 <= nx < size and 0 <= ny < size and maze[nx, ny] == True:
                    if (nx, ny) not in walls:
                        walls.append((nx, ny))
    # Create entrance and exit
    maze[1][0] = False  # Entrance
    maze[size-2][size-1] = False  # Exit

    return maze

def get_next_state_value(state, action, maze):
    directions = {'up': (-1, 0), 'right': (0, 1), 'down': (1, 0), 'left': (0, -1)}
    next_state = (state[0] + directions[action][0], state[1] + directions[action][1])
    if 0 <= next_state[0] < maze.shape[0] and 0 <= next_state[1] < maze.shape[1] and maze[next_state] == 0:
        return next_state
    return state  # Return the original state if the action is not feasible

def extract_policy_value(value_map, maze, goal_state, discount_factor=0.9):
    policy = {}
    for state in np.ndindex(maze.shape):
        if maze[state] == 1 or state == goal_state:
            continue
        max_value = float('-inf')
        best_action = None
        for action in ['up', 'right', 'down', 'left']:
            next_state = get_next_state_value(state, action, maze)
            value = value_map[next_state]
            if value > max_value:
                max_value = value
                best_action = action
        policy[state] = best_action
    return policy

def value_iteration_with_bool(maze, goal_state, vis=False, discount_factor=0.9, threshold=0.0000001, plot_every_n_iterations=15):
    value_map = np.zeros(maze.shape)
    iteration = 0
    while True:
        delta = 0
        for state in np.ndindex(maze.shape):
            if maze[state] == 1 or state == goal_state:
                continue
            v = value_map[state]
            max_value = float('-inf')
            for action in ['up', 'right', 'down', 'left']:
                next_state = get_next_state_value(state, action, maze)
                value = value_map[next_state]
                max_value = max(max_value, value)
            new_v = -1 + discount_factor * max_value  # -1 accounts for step cost
            value_map[state] = new_v
            delta = max(delta, abs(v - new_v))

        # Optionally plot the policy at this iteration
        if vis==True:
            if iteration % plot_every_n_iterations == 0:
                temp_policy = extract_policy_value(value_map, maze, goal_state, discount_factor)
                plot_policy_on_maze_value(maze, temp_policy, iteration)

        iteration += 1
        if delta < threshold:
            break

    return value_map, iteration

def plot_policy_on_maze_value(maze, policy, iteration=None, start=(1, 0), end=None):
    if end is None:
        end = (maze.shape[0]-2, maze.shape[1]-1)  # Default end point if not provided

    plt.figure(figsize=(8, 8))
    plt.imshow(maze, cmap='Greys', interpolation='none', alpha=0.8)
    for state, action in policy.items():
        if maze[state] == 0:  # Only plot for open cells
            draw_action_value(state, action)

    # Annotate start
    start_annotation_x = start[1] - 0.5 if start[1] > 0 else start[1] + 0.5
    start_annotation_y = start[0] - 3
    plt.annotate('Start', xy=(start[1], start[0]), xytext=(start_annotation_x, start_annotation_y),
                 arrowprops=dict(facecolor='green', shrink=0.05), fontsize=12, ha='center')

    # Annotate end
    end_annotation_x = end[1] + 0.5 if end[1] < maze.shape[1] - 1 else end[1] - 0.5
    end_annotation_y = end[0] + 2
    plt.annotate('End', xy=(end[1], end[0]), xytext=(end_annotation_x, end_annotation_y),
                 arrowprops=dict(facecolor='red', shrink=0.05), fontsize=12, ha='center')

    title = 'Optimal Value Visualised on Maze'
    if iteration is not None:
        title += f' at Iteration {iteration}'
    plt.title(title)
    plt.show()
```

```python
def draw_action_value(state, action):
    y, x = state
    if action == 'up':
        plt.arrow(x, y, 0, -0.5, head_width=0.1, head_length=0.2, fc='blue', ec='blue')
    elif action == 'down':
        plt.arrow(x, y, 0, 0.5, head_width=0.1, head_length=0.2, fc='blue', ec='blue')
    elif action == 'left':
        plt.arrow(x, y, -0.5, 0, head_width=0.1, head_length=0.2, fc='blue', ec='blue')
    elif action == 'right':
        plt.arrow(x, y, 0.5, 0, head_width=0.1, head_length=0.2, fc='blue', ec='blue')

def find_path_value(policy, start, goal, maze):
    path = [start]
    current = start
    while current != goal:
        action = policy.get(current)
        if action == 'up':
            next_state = (current[0]-1, current[1])
        elif action == 'down':
            next_state = (current[0]+1, current[1])
        elif action == 'left':
            next_state = (current[0], current[1]-1)
        elif action == 'right':
            next_state = (current[0], current[1]+1)
        if next_state == current:  # Prevent infinite loops
            break  # No valid move available from current position
        path.append(next_state)
        current = next_state
    return path

def visualize_path_value(maze, path):
    plt.figure(figsize=(8, 8))
    plt.imshow(maze, cmap='Greys', interpolation='none', alpha=0.8)
    ys, xs = zip(*path)
    plt.plot(xs, ys, color='red', linewidth=2, markersize=10, marker='o', label='Path')
    start = path[0]  # The first coordinate in your path list
    end = path[-1]  # The last coordinate in your path list

    # Calculate and adjust the x, y coordinates for the start and end annotations for better appearance
    start_annotation_x = start[1] - 0.5 if start[1] > 0 else start[1] + 0.5
    start_annotation_y = start[0] - 3  # Adjusting this line to correctly define start_annotation_y
    plt.annotate('Start', xy=(start[1], start[0]), xytext=(start_annotation_x, start_annotation_y),
                 arrowprops=dict(facecolor='green', shrink=0.05), fontsize=12, ha='center')
    end_annotation_x = end[1] + 0.5 if end[1] < maze.shape[1] - 1 else end[1] - 0.5
    end_annotation_y = end[0] + 2  # Adjusting for a more consistent position relative to the maze boundary
    plt.annotate('End', xy=(end[1], end[0]), xytext=(end_annotation_x, end_annotation_y),
                 arrowprops=dict(facecolor='red', shrink=0.05), fontsize=12, ha='center')

    plt.title('Optimal Path from Start to Goal')
    plt.legend()
    plt.show()

############################################
def plot_policy_on_maze_policy(maze, policy, iteration, start=(1, 0), end=None):
    if end is None:
        end = (maze.shape[0]-2, maze.shape[1]-1)  # Default end point if not provided

    plt.figure(figsize=(8, 8))
    plt.imshow(maze, cmap='Greys', interpolation='none', alpha=0.8)
    for state, action in policy.items():
        if maze[state] == 0:  # Only plot for open cells
            draw_action_policy(state, action)

    # Annotate start
    start_annotation_x = start[1] - 0.5 if start[1] > 0 else start[1] + 0.5
    start_annotation_y = start[0] - 3
    plt.annotate('Start', xy=(start[1], start[0]), xytext=(start_annotation_x, start_annotation_y),
                 arrowprops=dict(facecolor='green', shrink=0.05), fontsize=12, ha='center')

    # Annotate end
    end_annotation_x = end[1] + 0.5 if end[1] < maze.shape[1] - 1 else end[1] - 0.5
    end_annotation_y = end[0] + 2
    plt.annotate('End', xy=(end[1], end[0]), xytext=(end_annotation_x, end_annotation_y),
                 arrowprops=dict(facecolor='red', shrink=0.05), fontsize=12, ha='center')

    plt.title(f'Policy Iteration: Iteration {iteration}')
    plt.show()
    plt.close()   # Close the plot automatically after the pause

def draw_action_policy(state, action):
    y, x = state
    if action == 'up':
        plt.arrow(x, y, 0, -0.5, head_width=0.1, head_length=0.2, fc='blue', ec='blue')
    elif action == 'down':
        plt.arrow(x, y, 0, 0.5, head_width=0.1, head_length=0.2, fc='blue', ec='blue')
    elif action == 'left':
        plt.arrow(x, y, -0.5, 0, head_width=0.1, head_length=0.2, fc='blue', ec='blue')
    elif action == 'right':
        plt.arrow(x, y, 0.5, 0, head_width=0.1, head_length=0.2, fc='blue', ec='blue')

def find_path_policy(policy, start, goal, maze):
    path = [start]
    current = start
    while current != goal:
        action = policy.get(current)
        if action == 'up':
            next_state = (current[0]-1, current[1])
        elif action == 'down':
            next_state = (current[0]+1, current[1])
        elif action == 'left':
            next_state = (current[0], current[1]-1)
        elif action == 'right':
            next_state = (current[0], current[1]+1)
        path.append(next_state)
        current = next_state
    return path

def visualize_path_policy(maze, path, start=(1, 0), end=None):
    if end is None:
        end = (maze.shape[0]-2, maze.shape[1]-1)  # Assuming the default end point if not provided

    plt.figure(figsize=(8, 8))
    plt.imshow(maze, cmap='Greys', interpolation='none', alpha=0.8)
    ys, xs = zip(*path)
    plt.plot(xs, ys, color='red', linewidth=2, markersize=10, marker='o')

    # Annotate start
    start_annotation_x = start[1] - 0.5 if start[1] > 0 else start[1] + 0.5
    start_annotation_y = start[0] - 3
    plt.annotate('Start', xy=(start[1], start[0]), xytext=(start_annotation_x, start_annotation_y),
                 arrowprops=dict(facecolor='green', shrink=0.05), fontsize=12, ha='center')

    # Annotate end
    end_annotation_x = end[1] + 0.5 if end[1] < maze.shape[1] - 1 else end[1] - 0.5
    end_annotation_y = end[0] + 2
    plt.annotate('End', xy=(end[1], end[0]), xytext=(end_annotation_x, end_annotation_y),
                 arrowprops=dict(facecolor='red', shrink=0.05), fontsize=12, ha='center')

    plt.title('Path from Start to Goal')
    plt.show()
```

```python
def policy_evaluation(policy, utility, maze, discount_factor, goal_state):
    threshold = 0.001
    while True:
        delta = 0
        for state, action in policy.items():
            if maze[state] == 1 or state == goal_state:  # Skip walls and the goal state
                continue
            u = utility[state]
            reward = -1  # Constant step cost for non-goal states
            new_u = reward + discount_factor * utility[get_next_state_policy(state, action, maze)]
            utility[state] = new_u
            delta = max(delta, abs(u - new_u))
        if delta < threshold:
            break
    # Set the utility of the goal state to the highest positive reward
    utility[goal_state] = 0
    return utility

def policy_improvement(utility, maze, discount_factor, goal_state):
    new_policy = {}
    for state in np.ndindex(maze.shape):
        if maze[state] == 1 or state == goal_state:  # Skip walls and the goal state
            continue
        best_action = None
        best_value = float('-inf')
        for action in ['up', 'right', 'down', 'left']:
            next_state = get_next_state_policy(state, action, maze)
            value = utility[next_state]
            if value > best_value:
                best_value = value
                best_action = action
        new_policy[state] = best_action
    # Set the policy at the goal state to None (no action needed)
    new_policy[goal_state] = None
    return new_policy

def get_next_state_policy(state, action, maze):
    directions = {'up': (-1, 0), 'right': (0, 1), 'down': (1, 0), 'left': (0, -1)}
    next_state = (state[0] + directions[action][0], state[1] + directions[action][1])
    if 0 <= next_state[0] < maze.shape[0] and 0 <= next_state[1] < maze.shape[1] and maze[next_state] == 0:
        return next_state
    return state  # Return the original state if the action is not feasible

def policy_iteration_with_bool(maze, goal_state, vis=False, discount_factor=0.9):
    policy = {state: random.choice(['up', 'right', 'down', 'left'])
              for state in np.ndindex(maze.shape) if maze[state] == 0}
    utility = np.zeros(maze.shape)

    iteration = 0
    while True:
        utility = policy_evaluation(policy, utility, maze, discount_factor, goal_state)
        new_policy = policy_improvement(utility, maze, discount_factor, goal_state)

        plt.close()  # Close the plot automatically

        # Check if the policy has changed significantly
        if all(new_policy[state] == policy.get(state, None) for state in policy):
            break  # If not, then we have reached convergence
        policy = new_policy
        iteration += 1
        if vis==True:
            if(iteration%15==0):
                plot_policy_on_maze_policy(maze, policy, iteration)
    if vis==True:
        plot_policy_on_maze_policy(maze, policy, iteration)
    return policy, iteration

# Code for performance comparision of different algorithms on same maze

def main(arg1=35):
    maze_sizes=[25,45,65]
    print("Performance of Value iteration and Policy iteration on predefined maze sizes of 25,45,65 are")
    for maze_size in maze_sizes:
        value_times=list()
        v_iteration=list()
        v_mem=list()
        p_iteration=list()
        policy_times=list()
        p_mem=list()
        for i in range(3):
            maze = generate_maze(maze_size)
            #value iteration algorithm
            start_point = (1, 0)
            goal_state = (maze_size-1, maze_size-2)
            def wrap_value():
                start_time = time.time()
                value_map, viteration = value_iteration_with_bool(maze, goal_state, False)
                policy = extract_policy_value(value_map, maze, goal_state)
                path = find_path_value(policy, start_point, goal_state, maze)
                end_time = time.time()
                value_times.append(end_time-start_time)
                v_iteration.append(viteration)

            #policy iteration algorithm
            def wrap_policy():
                start_time = time.time()
                # policy, piteration = policy_iteration(maze,goal_state)
                policy, piteration = policy_iteration_with_bool(maze,goal_state,False)
                path = find_path_policy(policy, start_point, goal_state, maze)
                end_time = time.time()
                policy_times.append(end_time-start_time)
                p_iteration.append(piteration)
            v_mem.append(mean(memory_usage(proc=wrap_value)))
            p_mem.append(mean(memory_usage(proc=wrap_policy)))
        print(f"Time taken for Value iteration for size "+str(maze_size)+" is: "+(str(sum(value_times) / len(value_times)))+ "seconds &
Iteration taken are "+str(int(sum(v_iteration) / len(v_iteration))))
        print(f"Memory usage of Value iteration: {str(sum(v_mem) / len(v_mem))} MiB")
        print(f"Time taken for Policy iteration for size "+str(maze_size)+" is: "+(str(sum(policy_times) / len(policy_times)))+ "seconds &
Iterations taken are "+ str(int(sum(p_iteration) / len(p_iteration))))
        print(f"Memory usage of Policy iteration: {str(sum(p_mem) / len(p_mem))} MiB")
    time.sleep(2)


#code for visualisation of different algorithms on same maze

    maze_size = int(arg1)  # Adjust size as needed
    maze = generate_maze(maze_size)
    start_point = (1, 0)  # Adjust as needed
    goal_state = (maze_size-1, maze_size-2)  # Adjust the goal state as needed

    value_map, _ = value_iteration_with_bool(maze, goal_state, True)
    policy = extract_policy_value(value_map, maze, goal_state)
    # Plot the policy on the maze
    plot_policy_on_maze_value(maze, policy)
    # Find and visualize the path using the final policy
    path = find_path_value(policy, start_point, goal_state, maze)
    visualize_path_value(maze, path)


    policy, _ = policy_iteration_with_bool(maze,goal_state, True)
    # Find and visualize the path using the final policy
    path = find_path_policy(policy, start_point, goal_state, maze)
```

```python
    visualize_path_policy(maze, path)


if __name__ == '__main__':
    # This code won't run if this file is imported.
    args = sys.argv[1:]


    main(*args)
```