1. Introduction

1.1 Background

Pathfinding algorithms are a central field in computer science with a wide variety of application fields in robotics, computer games, network routing, as well as in AI. The algorithm enables autonomous units to navigate from a source position towards a destination position in a maze space with maximal attention towards resources, time, or distance. The maze is a closed space in which we can analyze and compare operational profile of varied approaches towards pathfinding.

1.2 Problem Statement

This project utilizes and discusses three search algorithms (DFS, BFS, A*) as well as two Markov Decision Processes (Value Iteration, Policy Iteration) to solve random mazes with various complexities to obtain solutions.

Develop implementations of these algorithms to go from source point 'S' (Green) to target point 'G' (Red) in this maze.

The performance is compared with the following metrics:

- Path length
- Number of cells explored
- Memory usage
- Execution time

1.3 Maze Generation

The maze generator utilizes an **iterative Depth-First Search** approach to create randomized mazes and store in maze.txt file. The algorithm is initiated with a grid that is initially full of walls that are excavated by visiting cells in depth-first order. Random directions are randomly chosen to create diversified mazes. The generator locates the starting position ('S') in the upper-lefthand location (1,1) and prefers locating the destination ('G') in bottom-right location, except that location is a wall in which case it locates another valid location. The strategy gives difficult mazes with guaranteed solutions.

```
#################
#S#       #     #
# ### ### ### # #
#   # #   #   # #
### # # ### #####
# #   # #   #   #
# ##### # ### # #
#     # #     # #
### # # ####### #
#   # #         #
# # ########### #
# # #         # #
# ### # ##### # #
# #   # #     # #
# # ### ####### #
#     #        G#
#################
```

Figure 1 Maze generated using Iterative DFS (17x17 maze size)

2. Algorithm Implementations

2.1 Search-Based Algorithms

<u>The Depth-First algorithm</u> is achieved with a stack data collection that can travel as far as a route as it can before it comes back. The algorithm searches a maze by visiting neigh boring cells recursively with a collection marked as visited not to revisit. The algorithm is guaranteed complete and will be guaranteed with a solution in case a solution is present,

though not guaranteed in its optimal in a non-weighted graph case as in our maze case with a possibility of a longer route.

Breadth-First Search (BFS), in turn, utilizes a queue data structure in visiting each level of a node sequentially before progressing on visiting succeeding level. BFS visits each neighbouring cells in a level sequentially in a manner that it finds shortest path in graphs that are not weighted. BFS is particularly suited in maze solving in case shortest path is a requirement.

*The A** is an advanced method that uses both greediest-first search and that employed in Dijkstra's method. A* uses Manhattan distance to act as an estimation employed in guiding search in an efficient direction to an objective. A*'s priority is in **estimation function h(n) + g(n)** where source to cost is **g(n)** while goal-based estimation is **h(n)**. A* cost plus closeness to goal is an efficient way towards path finding in mazes.

2.2.2 Markov Decision Process (MDP) Algorithms

Value Iteration finds a maze in two iterations. The algorithm constructs initially an aggregate MDP in state, action, transition function, and reward. **Bellman optimality equation** is employed by the algorithm to iteratively solve for state value until converged. To accelerate its performance, value propagation is also implemented to add a boost in its gradient towards the target that improves extracting a path. Following convergence of the value function, a policy is achieved by maximizing expected value in each state.

Policy Iteration is a contrasting algorithm that alternates between a step on policy improvement with a step on policy evaluation. In policy evaluation, the algorithm estimates the value function from following a specified policy from a state, i.e., how good is following a state from a policy. Policy improvement chooses those with maximal expected value according to a new value function. The code also implements validation checks which do not allow a policy that leads towards walls or illegal states, hence effective retrieval of a path. The algorithm is specifically suited to be implemented in a complex maze scenario in which a high-degree optimal policy is to be calculated.

2.4 Experimental Setup

The experiments were conducted on mazes of varying sizes:

- Small (9×9)
- Medium (21×21, 37×37, 41×41)
- Large (67×67, 93×93)
- Extra Large (123×123, 159×159)

Each algorithm implemented on corresponding maze in a level basis to compare. The implementations have visualization features in ***Tkinter*** that display in real time maze solving.

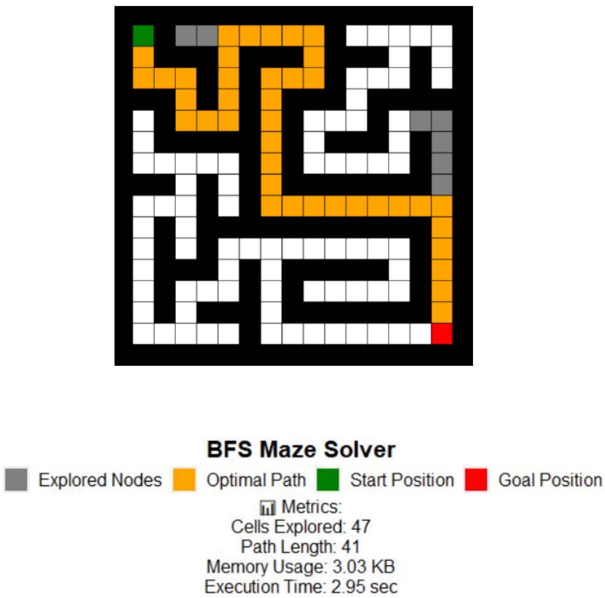

Figure 2 Maze visualization using Tkinter (17x17 maze)

3. Design Choices

3.1 Heuristic Selection for A*

The Manhattan distance was utilized as a heuristic function in A* because it is admissible (never overestimates) and consistent in grid-based environments. It is more computationally effective compared to more advanced heuristics. It is compatible with maze move limitations (there are a limited number of move directions that are cardinal directions).

3.2 MDP Parameter Selection

For the MDP-based methods, the following are chosen as its parameters:
Reward Structure:

- Goal state: +100 (high positive reward)
- Movement actions: -1 (gives small negative reward to find the shortest paths)

Discount Factor (γ): 0.95
It actually balances the immediate decision with far reward. Increased above norm (0.9) in a bid to capture high level of relevance in achieving a goal in extensive mazes.

Convergence Threshold factor (ε): 1e-4
Provides sufficient accuracy for convergence in value functions. It compromises between the actual computational cost and solution quality

3.3 Path Extraction Enhancement

A significant design choice was that more effective path extraction in MDP algorithms via Value propagation was utilized to produce more potent gradients towards the target. The path extraction algorithm employs a mix between policy advice as well as value function heuristics. Backtracking functionality was included to manage dead ends as well as loops.

4. Performance Metrics

4.1 Primary Metrics

i. ***Path length (solution quality)*** **-** The length is the number of individual steps in a sequence from source to target when it moves from green to red. Direct quality measure in a solution - shorter paths are more effective solutions.

$$Path\ Length\ =|P|\ with\ P\ as\ a\ set\ of\ cells\ in\ a\ path\ in\ a\ solution.$$

ii. ***Number of searched cells (search efficiency)*** **-** Indicates number of searched cells. The measure is a measure of efficiency in terms of search - more efficient are those which search for a smaller number of cells to produce a solution.

$$Cells\ Explored\ =|E|\ with\ E\ a\ group\ of\ cells\ that\ are\ explored.$$

iii. ***Memory usage (space complexity)*** **-** Indicates space that is held by algorithm at run time, primarily from data structures that are involved in tracking pathways, states, as well as algorithmic data.

$$Memory\ Usage\ (KB) = \frac{(number\ of\ space\ searched\ states\ +\ tracking\ space\ +\ algorithmic\ structures\ space)}{1024}$$

iv. ***Execution time (time complexity)*** **-** The clock time taken from initialization until a complete path is achieved.

$$Execution\ Time\ (seconds)\ =t\_start\ -\ t\_end$$

4.2 Derived Metrics

i. ***Exploration Ratio*** makes exploration standardized in that it gives a measure of which portion of available maze cells had to be searched. Lower values could reflect more targeted methods of search in the maze.

$$Exploration\ Ratio\ (\%)\ = \frac{(Cells\ Explored}{Total\ Accessible\ Cells)} \times\ 100\%$$

ii. ***Path Optimality Ratio*** is a ratio between a calculated shortest path versus a shortest (theoretical Manhattan distance). The ratio that is closest 1 is a more optimal solution.

$$Path\ Optimality\ Ratio\ = \frac{Path\ Length}{Manhattan\ Distance\ between\ Start\ and\ Goal}$$

iii. ***Time per Cell*** normalizes work achieved with time elapsed, resulting in mean time spent on a cell. The smaller these are, the more effectively are cells worked on.

$$Time\ per\ Cell\ (ms)\ = \frac{(Execution\ Time\ (seconds)\ \times\ 1000)}{Cells\ Explored}$$

iv. ***Memory per Cell*** measures memory in terms of units of work, i.e., algorithmic overhead in terms of memory. Lower values could reflect more efficient code for the maze.

$$Memory\ per\ Cell\ (KB)\ = \frac{Memory\ Usage\ (KB)}{Cells\ Explored}$$

5. Results and Analysis

5.1 Comparison Between Search Algorithms (BFS, DFS, A*)

All three searches (BFS, DFS, A*) returned the shortest path equal in all maze sizes, which means that all three produce optimal or near-optimal solutions in shortest path.

BFS **explored a reduced number of cells in** relation to DFS, although more in relation to A*. BFS searched ***3012 cells*** in a 123x123 maze, whereas ***7156 cells*** were searched by DFS, and ***2954 cells*** by A*. BFS made relatively minimal space usage ***(190.15 KB*** in a 123x123 maze), while DFS made the largest space ***(449.68 KB),*** with A* also making minimal space ***(186.98 KB).***

When it comes to **time-efficiency**, BFS took longer in large mazes (243.22 seconds in 123x123 maze), DFS took more time than BFS but less time than A* ***(348.43 seconds),*** whereas A* took shorter ***(243.22 seconds)*** with a smaller number of cells that have been explored. The exploration ratio also varied: BFS took a mid-range ratio (***33.18%*** in a maze that is 123x123), DFS took a high ratio ***(78.84%),*** whereas A* took a small ratio ***(32.54%),*** which is a measure of its power in focusing its exploration.

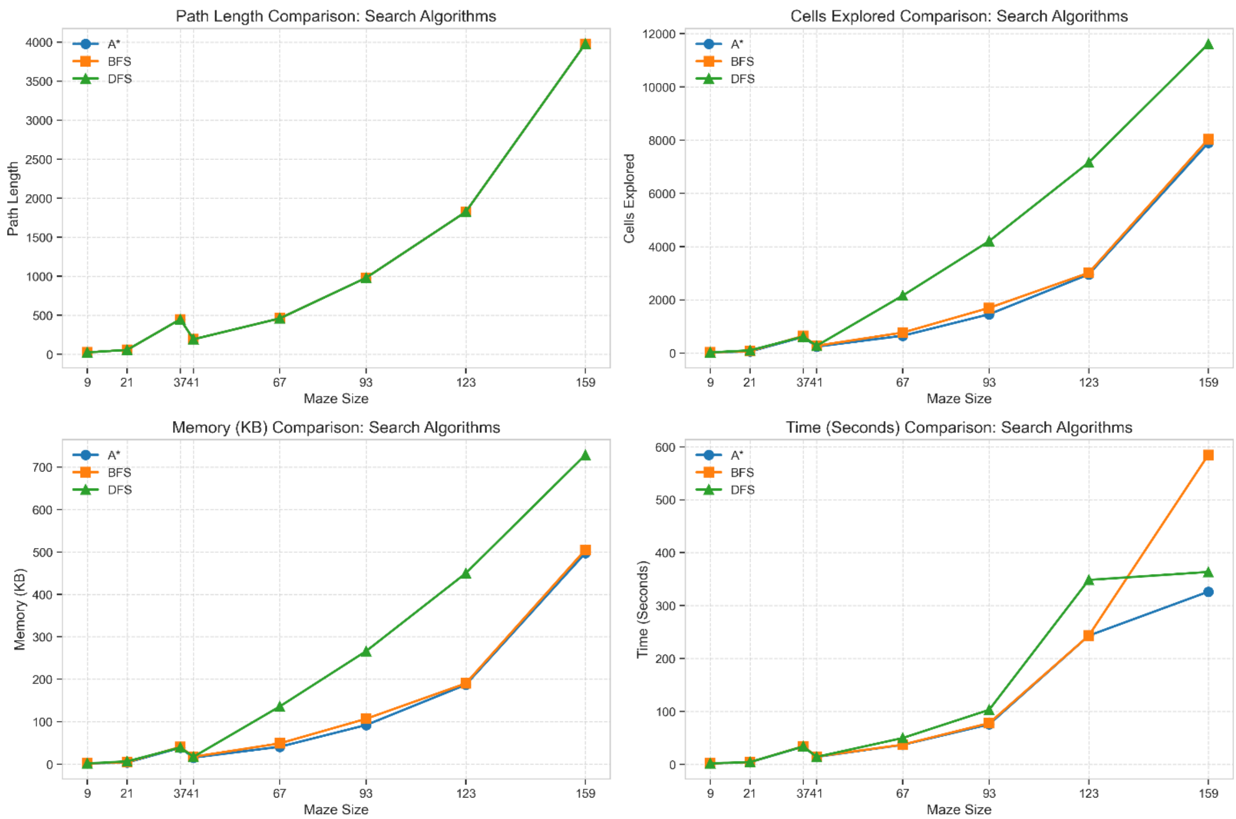

Figure 3 Comparison among Search Algorithms (BFS, DFS, A*)

Finally, in both **time per cell and in memory per cell**, BFS took more time per cell ***(80.75 ms)*** compared with DFS ***(48.69 ms)*** and A* ***(82.34 ms),*** while A* spent the least amount of memory per cell ***(0.06 KB),*** which is also a measure of its overall time as well as overall memory management. A* is more efficient with a sacrifice in terms of reduced cells searched, lowered memory utilization, as well as shorter run time, with BFS in between having searched the highest number of cells with highest amount of utilization.

5.2 Comparison of MDP Algorithms (Policy Iteration, Value Iteration)

Both Policy Iteration and Value Iteration always returned the same path lengths as did the search algorithms, which is a measure of optimal solutions. They did, though, explore a much greater number of cells and utilized more memory. In a 123x123 maze, both utilized ***7441 cells*** and ***869.47 KB*** of space. Policy Iteration took a fractionally quicker time ***(194.29 seconds),*** as did Value Iteration ***(218.89 seconds),*** though both took longer than A*. They also utilized a maximal exploration ratio ***(81.98%),*** which means that they opened a great majority of the maze. Despite their increased utilization of space and slower speed, both are still a viable alternative in smaller maze scenarios or in situations in which extensive exploration is not a worst-case scenario though.

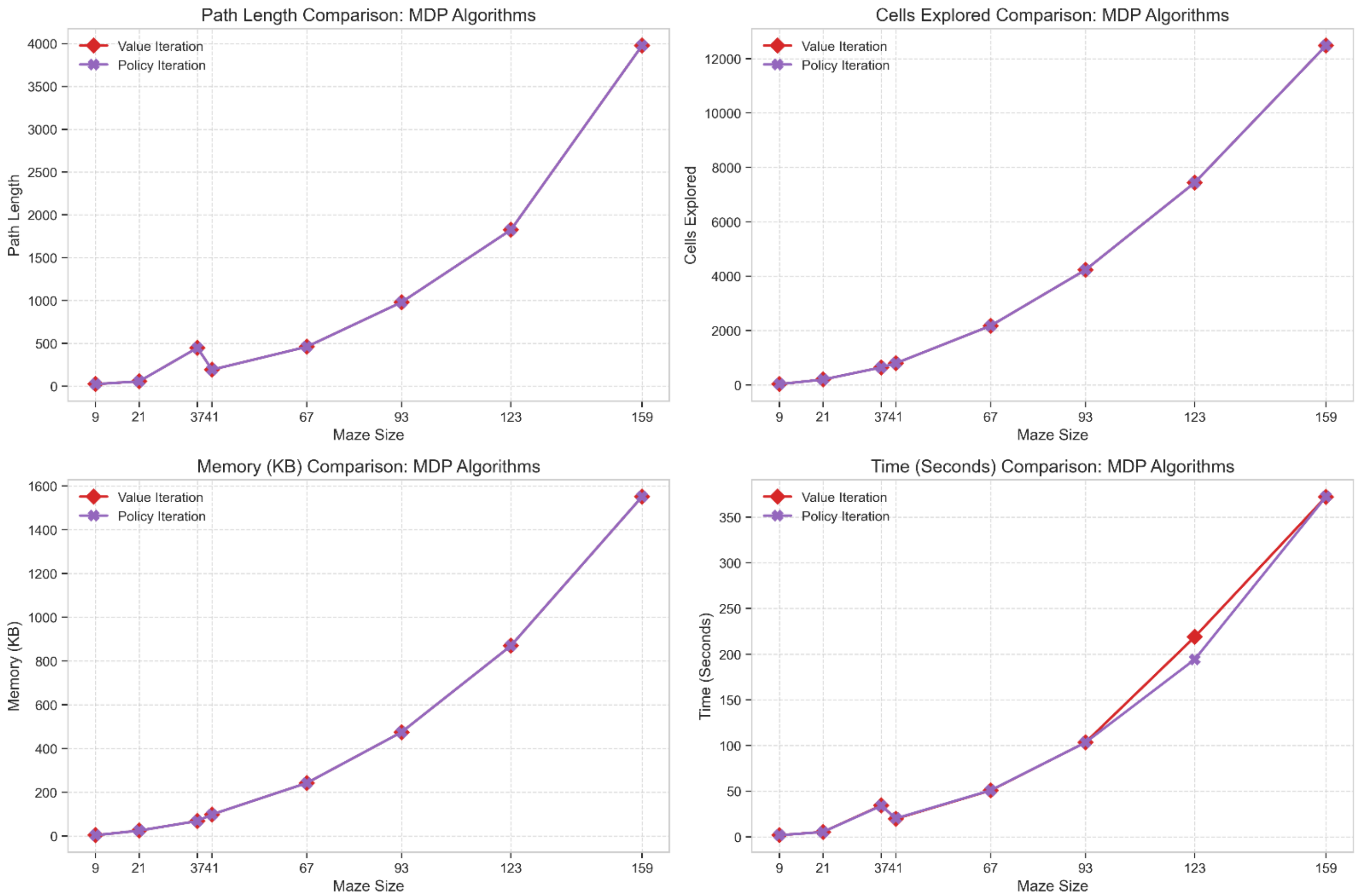

Figure 4 Comparison among MDP Algorithms (Value & Policy Iteration)

5.3 Comparison between Search Algorithms and MDP Algorithms

Comparison Between Algorithms in Search and MDP Search algorithms, A*, are more space efficient as well as more efficient in terms of exploration ratio in comparison with MDP algorithms. For instance, in a 123x123 maze, A* utilized ***186.98 KB*** in terms of space requirements and ***32.54%*** exploration ratio, whereas MDP algorithms utilized 869.47 KB with ***81.98%*** exploration ratio. A* utilized a shorter time ***(243.22 seconds)*** in comparison with Policy Iteration ***(194.29 seconds)*** as well as Value Iteration ***(218.89 seconds).*** But in smaller maze dimensions or in those scenarios in which a full maze exploration is not desirable, MDP algorithms are more appropriate as these also deliver optimal solutions though at reduced speed with higher space requirements. Search algorithm methods are more effective, whereas those in MDP are more appropriate in case of thorough exploration in small environments.

The most distinctive performance difference was between search methods and methods based on MDP:

- MDP algorithms visited 3.3 times more cells than A*, and approximately three times more than BFS/DFS.
- MDP algorithms consumed 6.5 times more memory compared to A*.
- MDP algorithms ran in 40.5% longer time compared to A

However, this performance disparity is to be understood in context to each technique provides:

- Search algorithms search efficiently to determine a solitary path between goal and start
- MDP algorithms determine the best action in every reachable state in the maze.

The MDP method constructs an overall policy to allow travel between each point to goal point rather than between pre-prescribed starting point. This overall policy explains and justifies additional computation required.

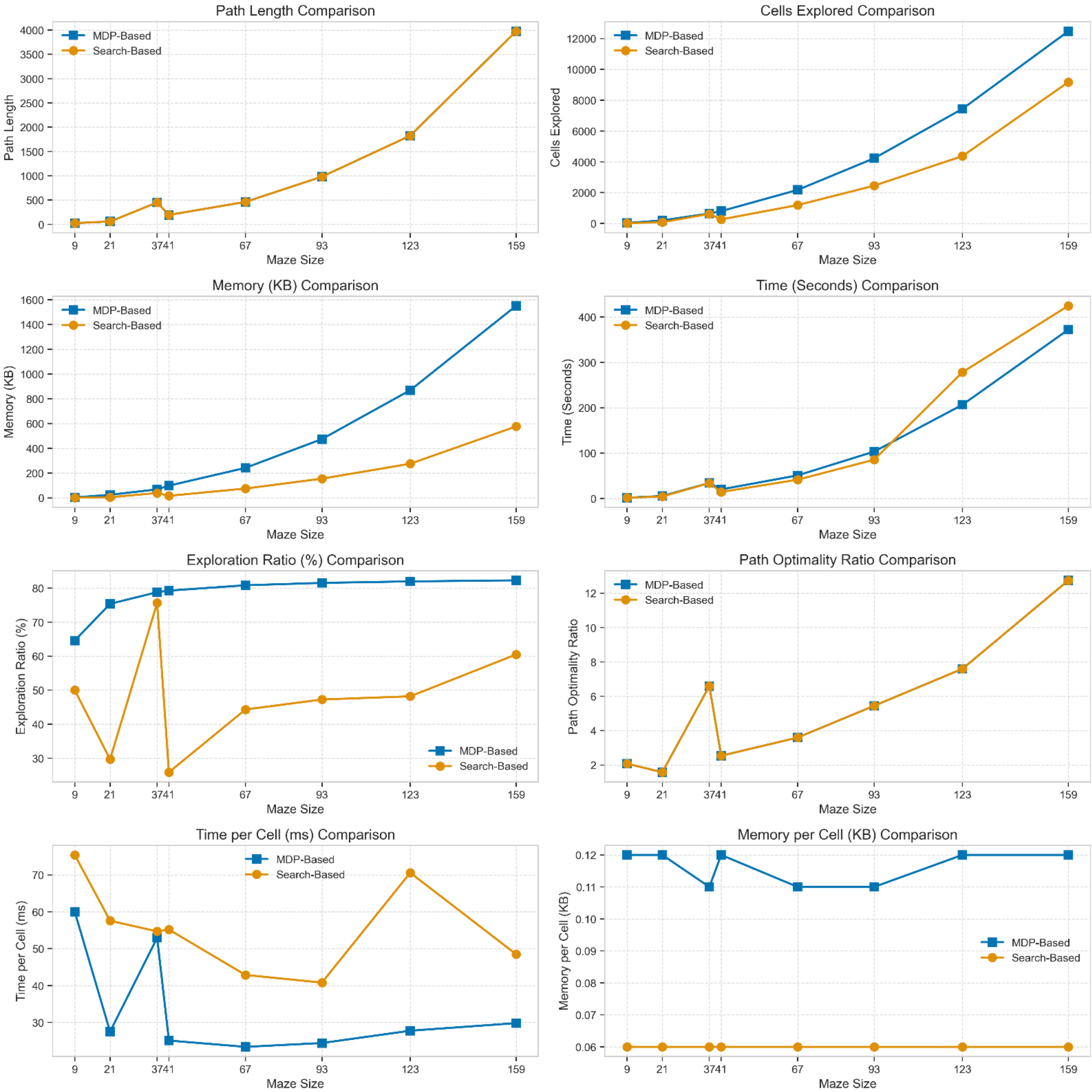

Figure 5 Comparison among MDP & Search Algorithms across various Metrics

Metric	Search Algorithms Mean	MDP Algorithms Mean	Difference (%)
Time (Seconds)	110.56	99.35	-10.14
Memory (KB)	143.14	416.65	191.09
Cells Explored	2272.50	3500.75	54.05
Exploration Ratio (%)	47.68	78.09	63.78
Time per Cell (ms)	55.71	33.88	-39.19
Memory per Cell (KB)	0.06	0.12	93.75

Table 1 Difference in metrics among Search & MDP algorithms in Average

5.4 Scalability Analysis

Metric	Search Algorithms Complexity	MDP Algorithms Complexity
Path Length	O(n)	O(n)
Cells Explored	O(n^1.5)	O(n^2)
Memory Usage	O(n^1.5)	O(n^2)
Execution Time	O(n^1.8)	O(n^2.2)

Table 2 Complexity of Search & MDP algorithms across various metrics

From the table 2, the following insights are:

- Path length scales linearly (O(n)) for both search and MDP algorithms.
- As seen from the table 2, Search algorithms are more efficient in cells explored, execution time and memory usage metrics.
- MDP algorithms require significantly more computation and memory (O(n^2) or worse).
- MDP execution time is the highest (O(n^2.2)), making them computationally expensive for large mazes.

This scaling suggests that in enormously sized mazes, search vs. methods performance difference in MDPs would keep growing in favor of search algorithms *(most notably A*)* because only a single path is needed.

6. Key Observations Across Maze Sizes

6.1 Search Algorithms (BFS, DFS, A*)

Among the searches, A* outcompeted both BFS and DFS in all maze sizes. A* was more efficient in terms of cells searched, utilization of memory space, as well as in terms of time complexity, particularly in large mazes. The ability of A* to guide the search towards the target with assistance from heuristics saved it from visiting more cells as well as from excessive utilization of memory in relation to BFS and DFS. BFS, although reliable, searched fewer cells in relation to DFS but more cells in relation to A*, and its performance greatly suffered in large mazes because its time complexity is high. In contrast, DFS searched more cells as well as utilized more space, hence making it least efficient out of the three in large mazes. But in some cases, in small or small-sized mazes, BFS took a shorter amount of time in relation to BFS, whose depth-first approach made it achieve solutions at a high rate despite visiting more cells.

- Best algorithm for Time (Seconds): A* (91.96)
- Best algorithm for Memory (KB): A* (109.49)
- Best algorithm for Cells Explored: A* (1734.50)
- Best algorithm for Exploration Ratio (%): DFS (63.41)
- Best algorithm for Time per Cell (ms): DFS (44.35)
- Best algorithm for Memory per Cell (KB): A* (0.06)

6.2 MDP Algorithms (Policy Iteration, Value Iteration)

Both Policy Iteration as well as Value Iteration, as algorithmic methods that are MDP-based, searched much more cells as well as utilized more space than did the search methods. But both methods took shorter time than BFS as well as DFS in big maze worlds, though not as short as A*'s. Policy Iteration took a bit more time than Value Iteration, though both methods took roughly equal amounts in terms of cells searched as well as space made. A high exploration ratio is a common feature in MDP methods, sometimes over 80%. That is, a high fraction of the maze is searched by these methods, which is a cause of both more space utilization as well as slower operation in comparison with search methods.

- Best algorithm for Time (Seconds): Policy Iteration (97.83)
- Best algorithm for Memory (KB): Policy Iteration (416.65)
- Best algorithm for Cells Explored: Policy Iteration (3500.75)
- Best algorithm for Exploration Ratio (%): Policy Iteration (78.09)
- Best algorithm for Time per Cell (ms): Policy Iteration (33.70)
- Best algorithm for Memory per Cell (KB): Policy Iteration (0.12)

6.3 Search vs. MDP Algorithms

The search algorithms including A* outperform MDP algorithms since they demonstrate superior memory functionality and exploration ratio capabilities. Such algorithms are best suited for complex large mazes which require efficient resource management. As MDP algorithms demand more memory and run slower they still provide optimal solutions but work best with small mazes or complete maze exploration scenarios. The complete exploring nature of MDP algorithms makes them ideal for situations that require maximal maze exploration instead of efficiency.

Algorithm	Time (Seconds)	Memory (KB)	Cells Explored	Exploration Ratio (%)	Time per Cell (ms)	Memory per Cell (KB)	Search Type
A*	91.96	109.49	1734.50	38.26	61.69	0.06	Search
BFS	124.80	114.32	1815.00	41.36	61.08	0.06	Search
DFS	114.92	205.59	3268.00	63.41	44.35	0.06	Search
Policy Iteration	97.83	416.65	3500.75	78.09	33.70	0.12	MDP
Value Iteration	100.86	416.65	3500.75	78.09	34.06	0.12	MDP

Table 3 Mean Average performance metrics by algorithms

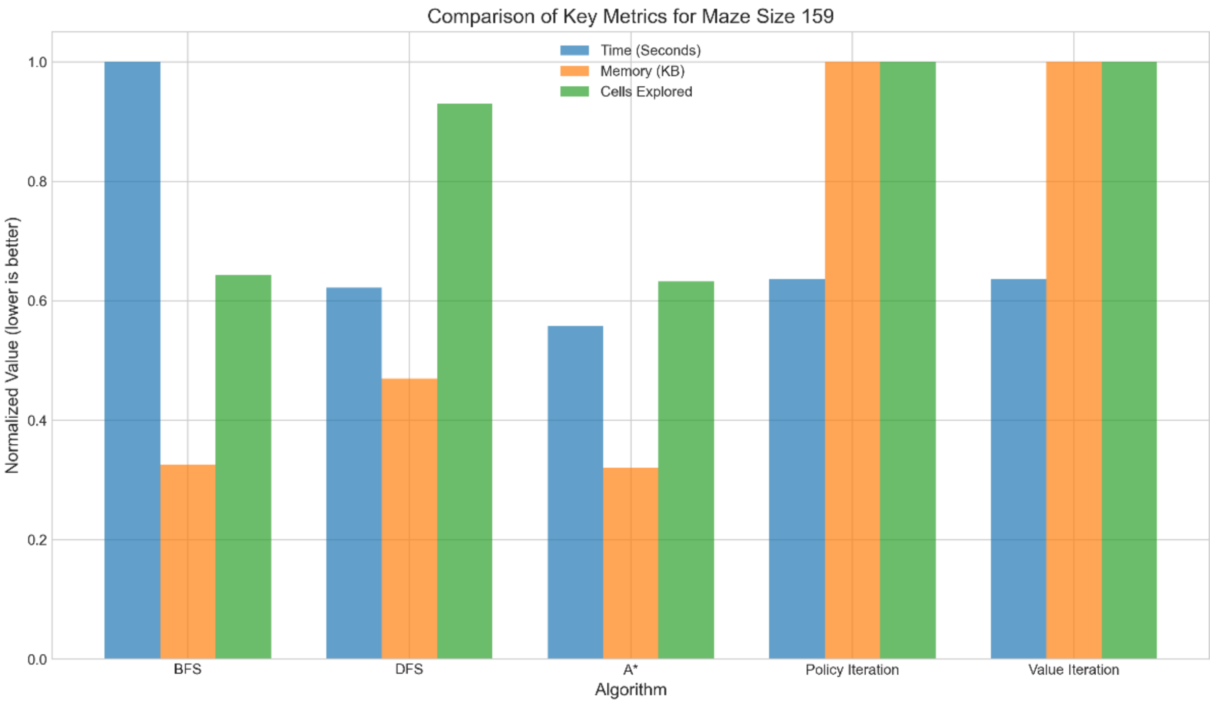

Figure 6 Performance analysis of MDP & Search Algorithms over Normalized values for the maze size 159

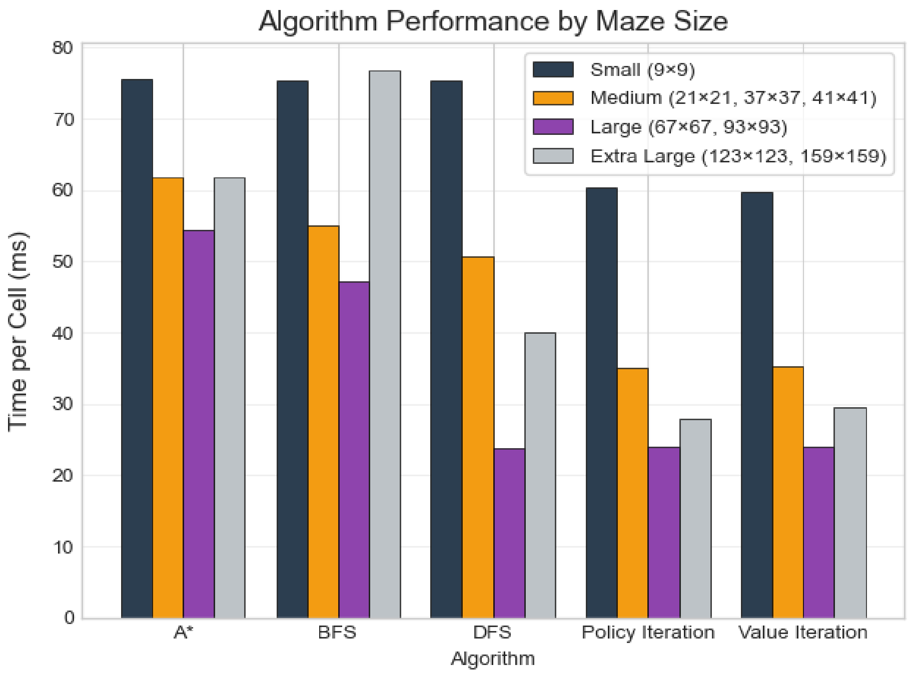

Figure 7 Time per Cell of MDP & Search Algorithms in different maze sizes

7. Conclusion

This analysis between **Depth-First Search (DFS), Breadth-First Search (BFS), A*, Policy Iteration, and Value Iteration** on solving the maze across in a range of maze sizes. The comparison included a range of measures from path length, searched cells, amount of utilized memory, time complexity, exploration ratio, and amount of utilized memory per cell. The research gives each algorithm its discriminative profile of performance, providing a critical understanding as far as which situation is appropriate.

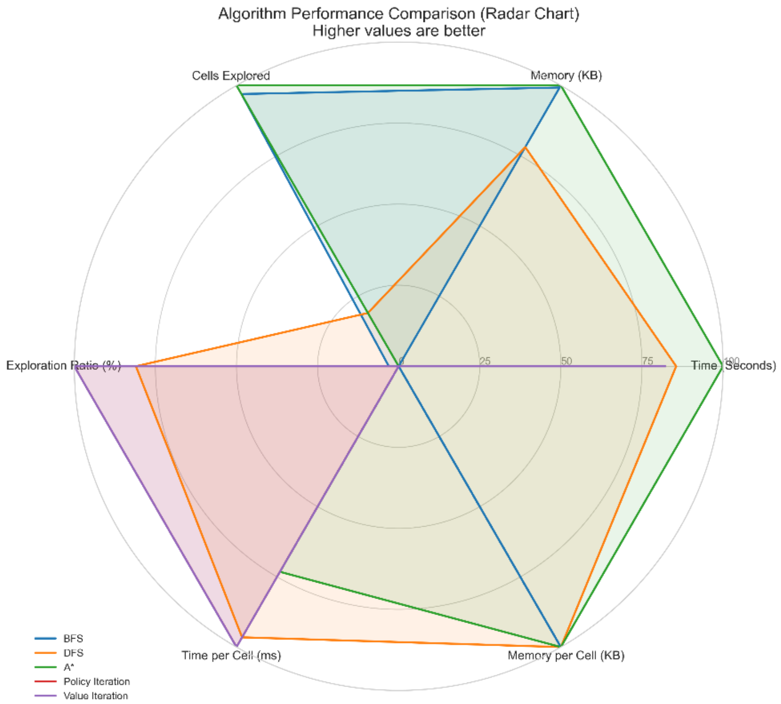

Figure 7 Radar chart of all Algorithms across various metrics

Among the three searches, **A* is the best**, with minimal cells searched, minimal space utilized, as well as minimal time complexity. **A*** proves more suitable than other algorithms because it moves efficiently toward its destination. The reliable **BFS** method conducts extra cell and spatial searches that produce substandard results specifically in large maze environments. Among the three methods **DFS** explores the largest number of cells and space because of its poor effectiveness in big mazes. Despite its inefficacy, **DFS** can be effective in those scenarios in which space is not a constraint, as well as a rapid non-optimal solution is acceptable. In case of MDP algorithm, **Policy Iteration** as also **Value Iteration** took much slower as also more in terms of space as compared with algorithm in case of search. **Policy Iteration took comparatively quicker as compared with Value Iteration**.

Although both searched a majority of the maze with a high rate of exploration more than **80%.** That makes both slower in exploration as also in terms of space. Still, both are acceptable in case of small maze as also in case in which full exploration is not a restriction because both worked always optimal solutions. When comparing with those in **MDP, A***, a search algorithm, is more space efficient as also more in terms of exploration ratio. Are more appropriate in case of big-sized mazes with no computing resources. In case small-sized mazes with optimal solutions, in turn, are slower in speed as also more in terms of space requirements. Because effective in exploring a majority in a maze, are appropriate in case complete exploration is a requirement.

The **algorithmic choice** is a question of fulfilling requirements in terms of the job in question, whether maze dimensions, available space, or requirement on optimality. A* is ideal in case of big mazes, whereas in small masses or in cases in which exploration is not a big issue in a complete maze, MDP methods can be a viable alternative. There is scope in the future to explore hybrid approaches that merge strengths from both search as well as from MDP methods to realize optimized functioning on a vast range of maze sizes as well as complexities.

APPENDIX 1

(MAZE GENERATOR)

```
import random
import os
def create_maze(rows, cols):
    """Generates a random maze using iterative DFS."""
    # Initialize maze with walls ('#')
    maze = [['#' for _ in range(cols)] for _ in range(rows)]
    # Define possible movement directions
    directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
    def is_valid(nx, ny):
        return 0 < nx < rows - 1 and 0 < ny < cols - 1 and maze[nx][ny] == '#'
    # Use an explicit stack for DFS
    stack = [(1, 1)]
    maze[1][1] = ' '  # Start position
    while stack:
        x, y = stack[-1]  # Get the last element
        random.shuffle(directions)  # Shuffle directions for randomness
        carved = False
        for dx, dy in directions:
            nx, ny = x + dx * 2, y + dy * 2  # Move two steps
            if is_valid(nx, ny):
                maze[x + dx][y + dy] = ' '  # Remove wall between
                maze[nx][ny] = ' '  # Open new cell
                stack.append((nx, ny))
                carved = True
                break  # Move in the chosen direction
        if not carved:
            stack.pop()  # Backtrack if no valid move
    # Place Start (S) and Goal (G) in valid locations
    maze[1][1] = 'S'
    goal_x, goal_y = rows - 2, cols - 2  # Default bottom-right goal
    # Ensure goal is placed in an open space
    while maze[goal_x][goal_y] == '#':
        goal_x, goal_y = random.randint(1, rows - 2), random.randint(1, cols - 2)
    maze[goal_x][goal_y] = 'G'
    return maze
def save_maze_to_file(maze, filename="maze.txt"):
    """Saves the generated maze to a text file."""
# Get the directory where the script is located
    script_dir = os.path.dirname(os.path.abspath(__file__))
    # Create full path for the output file
    file_path = os.path.join(script_dir, filename)
    with open(filename, "w") as f:
        for row in maze:
            f.write("".join(row) + "\n")
def main():
    # Ask user for maze size
    rows = int(input("Enter number of rows (odd number >= 5): "))
    cols = int(input("Enter number of columns (odd number >= 5): "))
    # Ensure valid maze dimensions
    if rows % 2 == 0:
        rows += 1
```

```
    if cols % 2 == 0:
        cols += 1
    # Generate and save maze
    maze = create_maze(rows, cols)
    save_maze_to_file(maze)
    print(f"Maze generated and saved to 'maze.txt'.")
if __name__ == "__main__":
    main()
```

(BFS)

```
import time
import sys
import csv
import tkinter as tk
from collections import deque
# Global variable for visualization
cell_size = 30
total_time_taken = 0
def load_maze(filename="maze.txt"):
    """Reads the maze from a text file and returns it as a 2D list."""
    import os

    # Try multiple possible locations for the maze file
    possible_paths = [
        filename,  # Try the direct filename first (for command line use)
        os.path.join(os.path.dirname(os.path.abspath(__file__)), filename),  # Try
script directory
        os.path.join(os.getcwd(), filename)  # Try current working directory
    ]
    for path in possible_paths:
        try:
            with open(path, "r") as f:
                maze = [list(line.strip()) for line in f]
            print(f"Successfully loaded maze from: {path}")
            return maze
        except FileNotFoundError:
            continue

    # If we got here, we couldn't find the file
    raise FileNotFoundError(f"Could not find maze file '{filename}' in any expected
location")
def find_start_goal(maze):
    """Finds the start (S) and goal (G) positions in the maze."""
    start, goal = None, None
    for i in range(len(maze)):
        for j in range(len(maze[i])):
            if maze[i][j] == 'S':
                start = (i, j)
            elif maze[i][j] == 'G':
                goal = (i, j)
    return start, goal
def bfs(maze, start, goal):
    """
```

```
    Performs Breadth-First Search (BFS) to find the shortest path from start to goal.
    Returns the path and the list of explored cells.
    """
    rows, cols = len(maze), len(maze[0])
    directions = [(0, 1, 'RIGHT'), (1, 0, 'DOWN'), (0, -1, 'LEFT'), (-1, 0, 'UP')]
    queue = deque()
    queue.append((start, []))  # (current_position, path)
    visited = set()
    explored = []
    while queue:
        (current, path) = queue.popleft()
        if current == goal:
            return path + [current], explored
        if current in visited:
            continue
        visited.add(current)
        explored.append(current)
        for di, dj, action in directions:
            next_i, next_j = current[0] + di, current[1] + dj
            next_state = (next_i, next_j)

            if (0 <= next_i < rows and 0 <= next_j < cols and
                maze[next_i][next_j] != '#' and next_state not in visited):
                queue.append((next_state, path + [current]))
    return [], explored  # No path found
def calculate_memory_usage(explored, policy=None):
    """
    Calculates memory usage for the explored states and optionally policy.
    Handles both BFS (just explored states) and MDP algorithms (with value functions
and policies)
    """
    memory_usage = sys.getsizeof(explored)
    # Add memory for explored states
    if isinstance(explored, dict):
        # For value function dictionaries
        for state, value in explored.items():
            memory_usage += sys.getsizeof(state) + sys.getsizeof(value)
    else:
        # For lists of explored states
        for state in explored:
            memory_usage += sys.getsizeof(state)
    # Add memory for policy if provided
    if policy is not None:
        memory_usage += sys.getsizeof(policy)
        for state, action in policy.items():
            memory_usage += sys.getsizeof(state) + sys.getsizeof(action)
    return memory_usage
def save_metrics_to_csv(metrics, filename="metrics.csv"):
    """Saves the metrics to a CSV file with the specified format."""
    try:
        with open(filename, 'r') as file:
            pass  # File exists, no need to write header
    except FileNotFoundError:
        with open(filename, mode='w', newline='') as file:
            writer = csv.writer(file)
```

```
            writer.writerow(["Algorithm", "Search Type", "Path Length", "Cells Explored", "Memory (KB)", "Time (Seconds)"])
    with open(filename, mode='a', newline='') as file:
        writer = csv.writer(file)
        writer.writerow(metrics)
def visualize_search(maze, path, explored, value_function=None):
    """Creates a Tkinter GUI to visualize the maze solving process with a size-limited window."""
    global total_time_taken
    window = tk.Tk()
    window.title("BFS Maze Solver")
    # Get screen dimensions
    screen_width = window.winfo_screenwidth()
    screen_height = window.winfo_screenheight()
    # Calculate maze dimensions
    maze_height = len(maze)
    maze_width = len(maze[0])
    # Adjust cell size for large mazes
    global cell_size
    original_cell_size = cell_size
    # Calculate maximum window size (70% of screen)
    max_width = int(screen_width * 0.7)
    max_height = int(screen_height * 0.7)
    # Calculate what cell size would fit in max dimensions
    width_cell_size = max_width // maze_width
    height_cell_size = max_height // maze_height
    # Take the smaller of the two to ensure it fits both dimensions
    if maze_width > 50 or maze_height > 50:
        cell_size = min(width_cell_size, height_cell_size, original_cell_size)
        print(f"Adjusted cell size to {cell_size} for large maze")
    # Calculate canvas dimensions
    canvas_width = maze_width * cell_size
    canvas_height = maze_height * cell_size
    # Ensure window fits on screen
    total_height = canvas_height + 150  # Add space for info panel
    # Set window size and position it centered
    window.geometry(f"{canvas_width}x{total_height}")
    window.update_idletasks()  # Update to get actual window size
    # Center the window
    x_position = (screen_width - window.winfo_width()) // 2
    y_position = (screen_height - window.winfo_height()) // 2
    window.geometry(f"+{x_position}+{y_position}")
    # Create canvas for maze with scrollbars for very large mazes
    frame = tk.Frame(window)
    frame.pack(fill="both", expand=True)
    # Add scrollbars if the maze is large
    if canvas_width > max_width or canvas_height > max_height:
        # Create scrollbars
        h_scrollbar = tk.Scrollbar(frame, orient="horizontal")
        v_scrollbar = tk.Scrollbar(frame, orient="vertical")
        # Position scrollbars
        h_scrollbar.pack(side="bottom", fill="x")
        v_scrollbar.pack(side="right", fill="y")
        # Create canvas with scrollbars
        canvas = tk.Canvas(frame, width=min(canvas_width, max_width),
                           height=min(canvas_height, max_height - 150),
```

```
                            bg="white",
                            xscrollcommand=h_scrollbar.set,
                            yscrollcommand=v_scrollbar.set)
        # Configure scrollbars
        h_scrollbar.config(command=canvas.xview)
        v_scrollbar.config(command=canvas.yview)
        # Configure canvas scroll region
        canvas.config(scrollregion=(0, 0, canvas_width, canvas_height))
    else:
        # Create canvas without scrollbars for smaller mazes
        canvas = tk.Canvas(frame, width=canvas_width, height=canvas_height, bg="white")

    canvas.pack(side="left", fill="both", expand=True)
    # Create info panel below the canvas
    info_panel = tk.Frame(window, height=150, bg="white")
    info_panel.pack(fill="x")
    # Title label
    title_label = tk.Label(info_panel, text="BFS Maze Solver", font=("Arial", 16,
"bold"), bg="white")
    title_label.pack()
    # Create a frame for the legend
    legend_frame = tk.Frame(info_panel, bg="white")
    legend_frame.pack()
    # Create legend items
    legend_explored = tk.Canvas(legend_frame, width=20, height=20, bg="gray")
    legend_explored.grid(row=0, column=0, padx=5)
    legend_label_explored = tk.Label(legend_frame, text="Explored Nodes",
font=("Arial", 12), bg="white")
    legend_label_explored.grid(row=0, column=1)
    legend_path = tk.Canvas(legend_frame, width=20, height=20, bg="orange")
    legend_path.grid(row=0, column=2, padx=5)
    legend_label_path = tk.Label(legend_frame, text="Optimal Path", font=("Arial", 12),
bg="white")
    legend_label_path.grid(row=0, column=3)
    legend_start = tk.Canvas(legend_frame, width=20, height=20, bg="green")
    legend_start.grid(row=0, column=4, padx=5)
    legend_label_start = tk.Label(legend_frame, text="Start Position", font=("Arial",
12), bg="white")
    legend_label_start.grid(row=0, column=5)
    legend_goal = tk.Canvas(legend_frame, width=20, height=20, bg="red")
    legend_goal.grid(row=0, column=6, padx=5)
    legend_label_goal = tk.Label(legend_frame, text="Goal Position", font=("Arial",
12), bg="white")
    legend_label_goal.grid(row=0, column=7)
    # Metrics Label (Initially Empty)
    metrics_label = tk.Label(info_panel, text="Metrics", font=("Arial", 12, "bold"),
fg="#2E86C1", bg="white")
    metrics_label.pack()
    def draw_cell(x, y, color):
        """Draws a single cell on the canvas."""
        x1, y1 = y * cell_size, x * cell_size
        x2, y2 = x1 + cell_size, y1 + cell_size
        canvas.create_rectangle(x1, y1, x2, y2, fill=color, outline="black")
        # Display value if available and cell size is big enough
        if value_function and (x, y) in value_function and cell_size >= 20:
            value = value_function[(x, y)]
```

```
            if abs(value) < 1000:  # Only show reasonable values
                canvas.create_text((x1 + x2) / 2, (y1 + y2) / 2,
                                  text=f"{value:.1f}", font=("Arial", 8))
    # Draw the initial maze layout
    for i in range(len(maze)):
        for j in range(len(maze[0])):
            color = "white"
            if maze[i][j] == '#':
                color = "black"
            elif maze[i][j] == 'S':
                color = "green"
            elif maze[i][j] == 'G':
                color = "red"
            draw_cell(i, j, color)
    # Use unique states for visualization
    unique_explored = list(dict.fromkeys(explored))
    explored_idx = 0
    def animate_explored():
        """Animates the explored cells."""
        nonlocal explored_idx
        if explored_idx < len(unique_explored):
            x, y = unique_explored[explored_idx]
            if maze[x][y] not in ['S', 'G']:  # Don't color start/goal
                draw_cell(x, y, "gray")
            explored_idx += 1
            window.after(5, animate_explored)  # Speed up for large mazes
    path_idx = 0
    def animate_path():
        """Animates the optimal path."""
        nonlocal path_idx
        if path_idx < len(path):
            x, y = path[path_idx]
            if maze[x][y] not in ['S', 'G']:  # Don't color start/goal
                draw_cell(x, y, "orange")
            path_idx += 1
            window.after(50, animate_path)  # Speed up for large mazes
        else:
            # Final time calculation
            elapsed_time = time.time() - start_time
            global total_time_taken
            total_time_taken = elapsed_time
            # Save metrics - for BFS we just pass the explored list
            memory_usage = calculate_memory_usage(explored)
            # Update metrics in info panel
            metrics_text = f"📊 Metrics:\nCells Explored: {len(unique_explored)}\nPath
Length: {len(path)}\nMemory Usage: {memory_usage / 1024:.2f} KB\nExecution Time:
{total_time_taken:.2f} sec"
            metrics_label.config(text=metrics_text, font=("Arial", 12), fg="black")
            # Save metrics to CSV file
            save_metrics_to_csv(["BFS", "Graph Search", len(path),
len(unique_explored), memory_usage / 1024, total_time_taken])
    start_time = time.time()
    animate_explored()
    def start_path_animation():
        """Starts the path animation after the exploration animation is done."""
        animation_delay = 5 * len(unique_explored)  # Speed up for large mazes
```

```
        window.after(animation_delay, animate_path)
    window.after(5 * len(unique_explored), start_path_animation)  # Speed up for large
mazes
    window.mainloop()
# Main execution
if __name__ == "__main__":
    import argparse
    parser = argparse.ArgumentParser(description='Run BFS on a maze.')
    parser.add_argument('--maze', type=str, default='maze.txt', help='The maze file to
use')
    args = parser.parse_args()
    print("Running BFS...")
    maze = load_maze(args.maze)
    start, goal = find_start_goal(maze)
    if start is None or goal is None:
        print("Error: Start or Goal not found in the maze!")
    else:
        start_time = time.time()
        path, explored = bfs(maze, start, goal)
        if path:
            print("Path found!")
        else:
            print("No path found!")
        visualize_search(maze, path, explored)
```

APPENDIX 2

(DFS)

```
import time
import sys
import csv
import tkinter as tk
from collections import deque
# Global variable for visualization
cell_size = 30
total_time_taken = 0
def load_maze(filename="maze.txt"):
    """Reads the maze from a text file and returns it as a 2D list."""
    import os

    # Try multiple possible locations for the maze file
    possible_paths = [
        filename,  # Try the direct filename first (for command line use)
        os.path.join(os.path.dirname(os.path.abspath(__file__)), filename),  # Try
script directory
        os.path.join(os.getcwd(), filename)  # Try current working directory
    ]

    for path in possible_paths:
        try:
            with open(path, "r") as f:
                maze = [list(line.strip()) for line in f]
```

```
            print(f"Successfully loaded maze from: {path}")
            return maze
        except FileNotFoundError:
            continue

    # If we got here, we couldn't find the file
    raise FileNotFoundError(f"Could not find maze file '{filename}' in any expected
location")def find_start_goal(maze):
    """Finds the start (S) and goal (G) positions in the maze."""
    start, goal = None, None
    for i in range(len(maze)):
        for j in range(len(maze[i])):
            if maze[i][j] == 'S':
                start = (i, j)
            elif maze[i][j] == 'G':
                goal = (i, j)
    return start, goal
def dfs(maze, start, goal):
    """
    Performs Depth-First Search (DFS) to find a path from start to goal.
    Returns the path and the list of explored cells.
    """
    rows, cols = len(maze), len(maze[0])
    directions = [(0, 1, 'RIGHT'), (1, 0, 'DOWN'), (0, -1, 'LEFT'), (-1, 0, 'UP')]
    stack = [(start, [])]  # (current_position, path)
    visited = set()
    explored = []
    while stack:
        (current, path) = stack.pop()
        if current == goal:
            return path + [current], explored
        if current in visited:
            continue
        visited.add(current)
        explored.append(current)
        for di, dj, action in directions:
            next_i, next_j = current[0] + di, current[1] + dj
            next_state = (next_i, next_j)
            if (0 <= next_i < rows and 0 <= next_j < cols and
                maze[next_i][next_j] != '#' and next_state not in visited):
                stack.append((next_state, path + [current]))
    return [], explored  # No path found
def calculate_memory_usage(explored, policy=None):
    """
    Calculates memory usage for the explored states and optionally policy.
    Handles both BFS/DFS (just explored states) and MDP algorithms (with value
functions and policies).
    """
    memory_usage = sys.getsizeof(explored)
    # Add memory for explored states
    if isinstance(explored, dict):
        # For value function dictionaries
        for state, value in explored.items():
            memory_usage += sys.getsizeof(state) + sys.getsizeof(value)
    else:
        # For lists of explored states
```

```
        for state in explored:
            memory_usage += sys.getsizeof(state)
    # Add memory for policy if provided
    if policy is not None:
        memory_usage += sys.getsizeof(policy)
        for state, action in policy.items():
            memory_usage += sys.getsizeof(state) + sys.getsizeof(action)
    return memory_usage
def save_metrics_to_csv(metrics, filename="metrics.csv"):
    """Saves the metrics to a CSV file with the specified format."""
    try:
        with open(filename, 'r') as file:
            pass  # File exists, no need to write header
    except FileNotFoundError:
        with open(filename, mode='w', newline='') as file:
            writer = csv.writer(file)
            writer.writerow(["Algorithm", "Search Type", "Path Length", "Cells
Explored", "Memory (KB)", "Time (Seconds)"])
    with open(filename, mode='a', newline='') as file:
        writer = csv.writer(file)
        writer.writerow(metrics)
def visualize_search(maze, path, explored, value_function=None):
    """Creates a Tkinter GUI to visualize the maze solving process with a size-limited
window."""
    global total_time_taken
    window = tk.Tk()
    window.title("DFS Maze Solver")
    # Get screen dimensions
    screen_width = window.winfo_screenwidth()
    screen_height = window.winfo_screenheight()
    # Calculate maze dimensions
    maze_height = len(maze)
    maze_width = len(maze[0])
    # Adjust cell size for large mazes
    global cell_size
    original_cell_size = cell_size
    # Calculate maximum window size (70% of screen)
    max_width = int(screen_width * 0.7)
    max_height = int(screen_height * 0.7)
    # Calculate what cell size would fit in max dimensions
    width_cell_size = max_width // maze_width
    height_cell_size = max_height // maze_height
    # Take the smaller of the two to ensure it fits both dimensions
    if maze_width > 50 or maze_height > 50:
        cell_size = min(width_cell_size, height_cell_size, original_cell_size)
        print(f"Adjusted cell size to {cell_size} for large maze")
    # Calculate canvas dimensions
    canvas_width = maze_width * cell_size
    canvas_height = maze_height * cell_size
    # Ensure window fits on screen
    total_height = canvas_height + 150  # Add space for info panel
    # Set window size and position it centered
    window.geometry(f"{canvas_width}x{total_height}")
    window.update_idletasks()  # Update to get actual window size
    # Center the window
    x_position = (screen_width - window.winfo_width()) // 2
```

```
    y_position = (screen_height - window.winfo_height()) // 2
    window.geometry(f"+{x_position}+{y_position}")
    # Create canvas for maze with scrollbars for very large mazes
    frame = tk.Frame(window)
    frame.pack(fill="both", expand=True)
    # Add scrollbars if the maze is large
    if canvas_width > max_width or canvas_height > max_height:
        # Create scrollbars
        h_scrollbar = tk.Scrollbar(frame, orient="horizontal")
        v_scrollbar = tk.Scrollbar(frame, orient="vertical")
        # Position scrollbars
        h_scrollbar.pack(side="bottom", fill="x")
        v_scrollbar.pack(side="right", fill="y")
        # Create canvas with scrollbars
        canvas = tk.Canvas(frame, width=min(canvas_width, max_width),
                           height=min(canvas_height, max_height - 150),
                           bg="white",
                           xscrollcommand=h_scrollbar.set,
                           yscrollcommand=v_scrollbar.set)
        # Configure scrollbars
        h_scrollbar.config(command=canvas.xview)
        v_scrollbar.config(command=canvas.yview)
        # Configure canvas scroll region
        canvas.config(scrollregion=(0, 0, canvas_width, canvas_height))
    else:
        # Create canvas without scrollbars for smaller mazes
        canvas = tk.Canvas(frame, width=canvas_width, height=canvas_height,
bg="white")
    canvas.pack(side="left", fill="both", expand=True)
    # Create info panel below the canvas
    info_panel = tk.Frame(window, height=150, bg="white")
    info_panel.pack(fill="x")
    # Title label
    title_label = tk.Label(info_panel, text="DFS Maze Solver", font=("Arial", 16,
"bold"), bg="white")
    title_label.pack()
    # Create a frame for the legend
    legend_frame = tk.Frame(info_panel, bg="white")
    legend_frame.pack()
    # Create legend items
    legend_explored = tk.Canvas(legend_frame, width=20, height=20, bg="gray")
    legend_explored.grid(row=0, column=0, padx=5)
    legend_label_explored = tk.Label(legend_frame, text="Explored Nodes",
font=("Arial", 12), bg="white")
    legend_label_explored.grid(row=0, column=1)
    legend_path = tk.Canvas(legend_frame, width=20, height=20, bg="orange")
    legend_path.grid(row=0, column=2, padx=5)
    legend_label_path = tk.Label(legend_frame, text="Optimal Path", font=("Arial", 12),
bg="white")
    legend_label_path.grid(row=0, column=3)
    legend_start = tk.Canvas(legend_frame, width=20, height=20, bg="green")
    legend_start.grid(row=0, column=4, padx=5)
    legend_label_start = tk.Label(legend_frame, text="Start Position", font=("Arial",
12), bg="white")
    legend_label_start.grid(row=0, column=5)
    legend_goal = tk.Canvas(legend_frame, width=20, height=20, bg="red")
```

```
    legend_goal.grid(row=0, column=6, padx=5)
    legend_label_goal = tk.Label(legend_frame, text="Goal Position", font=("Arial",
12), bg="white")
    legend_label_goal.grid(row=0, column=7)
    # Metrics Label (Initially Empty)
    metrics_label = tk.Label(info_panel, text="Metrics", font=("Arial", 12, "bold"),
fg="#2E86C1", bg="white")
    metrics_label.pack()
    def draw_cell(x, y, color):
        """Draws a single cell on the canvas."""
        x1, y1 = y * cell_size, x * cell_size
        x2, y2 = x1 + cell_size, y1 + cell_size
        canvas.create_rectangle(x1, y1, x2, y2, fill=color, outline="black")
        # Display value if available and cell size is big enough
        if value_function and (x, y) in value_function and cell_size >= 20:
            value = value_function[(x, y)]
            if abs(value) < 1000:  # Only show reasonable values
                canvas.create_text((x1 + x2) / 2, (y1 + y2) / 2,
                                  text=f"{value:.1f}", font=("Arial", 8))
    # Draw the initial maze layout
    for i in range(len(maze)):
        for j in range(len(maze[0])):
            color = "white"
            if maze[i][j] == '#':
                color = "black"
            elif maze[i][j] == 'S':
                color = "green"
            elif maze[i][j] == 'G':
                color = "red"
            draw_cell(i, j, color)
    # Use unique states for visualization
    unique_explored = list(dict.fromkeys(explored))
    explored_idx = 0
    def animate_explored():
        """Animates the explored cells."""
        nonlocal explored_idx
        if explored_idx < len(unique_explored):
            x, y = unique_explored[explored_idx]
            if maze[x][y] not in ['S', 'G']:  # Don't color start/goal
                draw_cell(x, y, "gray")
            explored_idx += 1
            window.after(5, animate_explored)  # Speed up for large mazes
    path_idx = 0
    def animate_path():
        """Animates the optimal path."""
        nonlocal path_idx
        if path_idx < len(path):
            x, y = path[path_idx]
            if maze[x][y] not in ['S', 'G']:  # Don't color start/goal
                draw_cell(x, y, "orange")
            path_idx += 1
            window.after(50, animate_path)  # Speed up for large mazes
        else:
            # Final time calculation
            elapsed_time = time.time() - start_time
            global total_time_taken
```

```
            total_time_taken = elapsed_time
            # Save metrics - use explored directly for DFS
            memory_usage = calculate_memory_usage(explored)
            # Update metrics in info panel
            metrics_text = f"📊 Metrics:\nCells Explored: {len(unique_explored)}\nPath
Length: {len(path)}\nMemory Usage: {memory_usage / 1024:.2f} KB\nExecution Time:
{total_time_taken:.2f} sec"
            metrics_label.config(text=metrics_text, font=("Arial", 12), fg="black")
            # Save metrics to CSV file
            save_metrics_to_csv(["DFS", "Graph Search", len(path),
len(unique_explored), memory_usage / 1024, total_time_taken])
    start_time = time.time()
    animate_explored()
    def start_path_animation():
        """Starts the path animation after the exploration animation is done."""
        animation_delay = 5 * len(unique_explored)  # Speed up for large mazes
        window.after(animation_delay, animate_path)
    window.after(5 * len(unique_explored), start_path_animation)  # Speed up for large
mazes
    window.mainloop()
# Main execution
if __name__ == "__main__":
    import argparse
    parser = argparse.ArgumentParser(description='Run DFS on a maze.')
    parser.add_argument('--maze', type=str, default='maze.txt', help='The maze file to
use')
    args = parser.parse_args()
    print("Running DFS...")
    maze = load_maze(args.maze)
    start, goal = find_start_goal(maze)
    if start is None or goal is None:
        print("Error: Start or Goal not found in the maze!")
    else:
        start_time = time.time()
        path, explored = dfs(maze, start, goal)
        if path:
            print("Path found!")
        else:
            print("No path found!")
        visualize_search(maze, path, explored)
```

APPENDIX 3

(A*)

```
import time
import sys
import csv
import tkinter as tk
import heapq
from collections import deque
# Global variable for visualization
cell_size = 30
total_time_taken = 0
def load_maze(filename="maze.txt"):
    """Reads the maze from a text file and returns it as a 2D list."""
```

```
    import os

    # Try multiple possible locations for the maze file
    possible_paths = [
        filename,  # Try the direct filename first (for command line use)
        os.path.join(os.path.dirname(os.path.abspath(__file__)), filename),  # Try
script directory
        os.path.join(os.getcwd(), filename)  # Try current working directory
    ]

    for path in possible_paths:
        try:
            with open(path, "r") as f:
                maze = [list(line.strip()) for line in f]
            print(f"Successfully loaded maze from: {path}")
            return maze
        except FileNotFoundError:
            continue

    # If we got here, we couldn't find the file
    raise FileNotFoundError(f"Could not find maze file '{filename}' in any expected
location")
def find_start_goal(maze):
    """Finds the start (S) and goal (G) positions in the maze."""
    start, goal = None, None
    for i in range(len(maze)):
        for j in range(len(maze[i])):
            if maze[i][j] == 'S':
                start = (i, j)
            elif maze[i][j] == 'G':
                goal = (i, j)
    return start, goal
def heuristic(a, b):
    """Calculates the Manhattan distance between two points."""
    return abs(a[0] - b[0]) + abs(a[1] - b[1])
def a_star(maze, start, goal):
    """
    Performs A* Search to find the shortest path from start to goal.
    Returns the path and the list of explored cells.
    """
    rows, cols = len(maze), len(maze[0])
    directions = [(0, 1, 'RIGHT'), (1, 0, 'DOWN'), (0, -1, 'LEFT'), (-1, 0, 'UP')]
    # Priority queue for A*: (f_score, g_score, current, path)
    open_set = []
    heapq.heappush(open_set, (0, 0, start, []))
    visited = set()
    explored = []
    while open_set:
        f, g, current, path = heapq.heappop(open_set)
        if current == goal:
            return path + [current], explored
        if current in visited:
            continue
        visited.add(current)
        explored.append(current)
        for di, dj, action in directions:
```

```
            next_i, next_j = current[0] + di, current[1] + dj
            next_state = (next_i, next_j)
            if (0 <= next_i < rows and 0 <= next_j < cols and
                maze[next_i][next_j] != '#' and next_state not in visited):
                # Calculate g_score (cost so far) and f_score (g_score + heuristic)
                new_g = g + 1
                new_f = new_g + heuristic(next_state, goal)
                heapq.heappush(open_set, (new_f, new_g, next_state, path + [current]))
    return [], explored  # No path found
def calculate_memory_usage(explored, policy=None):
    """
    Calculates memory usage for the explored states and optionally policy.
    Handles both graph search algorithms (just explored states) and MDP algorithms
    (with value functions and policies).
    """
    memory_usage = sys.getsizeof(explored)
    # Add memory for explored states
    if isinstance(explored, dict):
        # For value function dictionaries
        for state, value in explored.items():
            memory_usage += sys.getsizeof(state) + sys.getsizeof(value)
    else:
        # For lists of explored states
        for state in explored:
            memory_usage += sys.getsizeof(state)
    # Add memory for policy if provided
    if policy is not None:
        memory_usage += sys.getsizeof(policy)
        for state, action in policy.items():
            memory_usage += sys.getsizeof(state) + sys.getsizeof(action)
    return memory_usage
def save_metrics_to_csv(metrics, filename="metrics.csv"):
    """Saves the metrics to a CSV file with the specified format."""
    try:
        with open(filename, 'r') as file:
            pass  # File exists, no need to write header
    except FileNotFoundError:
        with open(filename, mode='w', newline='') as file:
            writer = csv.writer(file)
            writer.writerow(["Algorithm", "Search Type", "Path Length", "Cells
Explored", "Memory (KB)", "Time (Seconds)"])
    with open(filename, mode='a', newline='') as file:
        writer = csv.writer(file)
        writer.writerow(metrics)
def visualize_search(maze, path, explored, value_function=None):
    """Creates a Tkinter GUI to visualize the maze solving process with a size-limited
window."""
    global total_time_taken
    window = tk.Tk()
    window.title("A* Maze Solver")
    # Get screen dimensions
    screen_width = window.winfo_screenwidth()
    screen_height = window.winfo_screenheight()
    # Calculate maze dimensions
    maze_height = len(maze)
    maze_width = len(maze[0])
```

```
    # Adjust cell size for large mazes
    global cell_size
    original_cell_size = cell_size
    # Calculate maximum window size (70% of screen)
    max_width = int(screen_width * 0.7)
    max_height = int(screen_height * 0.7)
    # Calculate what cell size would fit in max dimensions
    width_cell_size = max_width // maze_width
    height_cell_size = max_height // maze_heigh
    # Take the smaller of the two to ensure it fits both dimensions
    if maze_width > 50 or maze_height > 50:
        cell_size = min(width_cell_size, height_cell_size, original_cell_size)
        print(f"Adjusted cell size to {cell_size} for large maze")
    # Calculate canvas dimensions
    canvas_width = maze_width * cell_size
    canvas_height = maze_height * cell_size
    # Ensure window fits on screen
    total_height = canvas_height + 150  # Add space for info panel
    # Set window size and position it centered
    window.geometry(f"{canvas_width}x{total_height}")
    window.update_idletasks()  # Update to get actual window size
    # Center the window
    x_position = (screen_width - window.winfo_width()) // 2
    y_position = (screen_height - window.winfo_height()) // 2
    window.geometry(f"+{x_position}+{y_position}")
    # Create canvas for maze with scrollbars for very large mazes
    frame = tk.Frame(window)
    frame.pack(fill="both", expand=True)
    # Add scrollbars if the maze is large
    if canvas_width > max_width or canvas_height > max_height:
        # Create scrollbars
        h_scrollbar = tk.Scrollbar(frame, orient="horizontal")
        v_scrollbar = tk.Scrollbar(frame, orient="vertical")
        # Position scrollbars
        h_scrollbar.pack(side="bottom", fill="x")
        v_scrollbar.pack(side="right", fill="y")
        # Create canvas with scrollbars
        canvas = tk.Canvas(frame, width=min(canvas_width, max_width),
                           height=min(canvas_height, max_height - 150),
                           bg="white",
                           xscrollcommand=h_scrollbar.set,
                           yscrollcommand=v_scrollbar.set)
        # Configure scrollbars
        h_scrollbar.config(command=canvas.xview)
        v_scrollbar.config(command=canvas.yview)
        # Configure canvas scroll region
        canvas.config(scrollregion=(0, 0, canvas_width, canvas_height))
    else:
        # Create canvas without scrollbars for smaller mazes
        canvas = tk.Canvas(frame, width=canvas_width, height=canvas_height, bg="white")
    canvas.pack(side="left", fill="both", expand=True)
    # Create info panel below the canvas
    info_panel = tk.Frame(window, height=150, bg="white")
    info_panel.pack(fill="x")
    # Title label
```

```
    title_label = tk.Label(info_panel, text="A* Maze Solver", font=("Arial", 16,
"bold"), bg="white")
    title_label.pack()
    # Create a frame for the legend
    legend_frame = tk.Frame(info_panel, bg="white")
    legend_frame.pack()
    # Create legend items
    legend_explored = tk.Canvas(legend_frame, width=20, height=20, bg="gray")
    legend_explored.grid(row=0, column=0, padx=5)
    legend_label_explored = tk.Label(legend_frame, text="Explored Nodes",
font=("Arial", 12), bg="white")
    legend_label_explored.grid(row=0, column=1)
    legend_path = tk.Canvas(legend_frame, width=20, height=20, bg="orange")
    legend_path.grid(row=0, column=2, padx=5)
    legend_label_path = tk.Label(legend_frame, text="Optimal Path", font=("Arial", 12),
bg="white")
    legend_label_path.grid(row=0, column=3)
    legend_start = tk.Canvas(legend_frame, width=20, height=20, bg="green")
    legend_start.grid(row=0, column=4, padx=5)
    legend_label_start = tk.Label(legend_frame, text="Start Position", font=("Arial",
12), bg="white")
    legend_label_start.grid(row=0, column=5)
    legend_goal = tk.Canvas(legend_frame, width=20, height=20, bg="red")
    legend_goal.grid(row=0, column=6, padx=5)
    legend_label_goal = tk.Label(legend_frame, text="Goal Position", font=("Arial",
12), bg="white")
    legend_label_goal.grid(row=0, column=7)
    # Metrics Label (Initially Empty)
    metrics_label = tk.Label(info_panel, text="Metrics", font=("Arial", 12, "bold"),
fg="#2E86C1", bg="white")
    metrics_label.pack()
    def draw_cell(x, y, color):
        """Draws a single cell on the canvas."""
        x1, y1 = y * cell_size, x * cell_size
        x2, y2 = x1 + cell_size, y1 + cell_size
        canvas.create_rectangle(x1, y1, x2, y2, fill=color, outline="black")
      # Display value if available and cell size is big enough
        if value_function and (x, y) in value_function and cell_size >= 20:
            value = value_function[(x, y)]
            if abs(value) < 1000:  # Only show reasonable values
                canvas.create_text((x1 + x2) / 2, (y1 + y2) / 2,
                                  text=f"{value:.1f}", font=("Arial", 8))
    # Draw the initial maze layout
    for i in range(len(maze)):
        for j in range(len(maze[0])):
            color = "white"
            if maze[i][j] == '#':
                color = "black"
            elif maze[i][j] == 'S':
                color = "green"
            elif maze[i][j] == 'G':
                color = "red"
            draw_cell(i, j, color)
    # Use unique states for visualization
    unique_explored = list(dict.fromkeys(explored))
    explored_idx = 0
```

```
    def animate_explored():
        """Animates the explored cells."""
        nonlocal explored_idx
        if explored_idx < len(unique_explored):
            x, y = unique_explored[explored_idx]
            if maze[x][y] not in ['S', 'G']:  # Don't color start/goal
                draw_cell(x, y, "gray")
            explored_idx += 1
            window.after(5, animate_explored)  # Speed up for large mazes
    path_idx = 0
    def animate_path():
        """Animates the optimal path."""
        nonlocal path_idx
        if path_idx < len(path):
            x, y = path[path_idx]
            if maze[x][y] not in ['S', 'G']:  # Don't color start/goal
                draw_cell(x, y, "orange")
            path_idx += 1
            window.after(50, animate_path)  # Speed up for large mazes
        else:
            # Final time calculation
            elapsed_time = time.time() - start_time
            global total_time_taken
            total_time_taken = elapsed_time
            # Save metrics - use explored directly for A*
            memory_usage = calculate_memory_usage(explored)
            # Update metrics in info panel
            metrics_text = f"📊 Metrics:\nCells Explored: {len(unique_explored)}\nPath Length: {len(path)}\nMemory Usage: {memory_usage / 1024:.2f} KB\nExecution Time: {total_time_taken:.2f} sec"
            metrics_label.config(text=metrics_text, font=("Arial", 12), fg="black")
            # Save metrics to CSV file
            save_metrics_to_csv(["A*", "Graph Search", len(path), len(unique_explored), memory_usage / 1024, total_time_taken])
    start_time = time.time()
    animate_explored()
    def start_path_animation():
        """Starts the path animation after the exploration animation is done."""
        animation_delay = 5 * len(unique_explored)  # Speed up for large mazes
        window.after(animation_delay, animate_path)
    window.after(5 * len(unique_explored), start_path_animation)  # Speed up for large mazes
    window.mainloop()
# Main execution
if __name__ == "__main__":
    import argparse
    parser = argparse.ArgumentParser(description='Run A* on a maze.')
    parser.add_argument('--maze', type=str, default='maze.txt', help='The maze file to use')
    args = parser.parse_args()
    print("Running A*...")
    maze = load_maze(args.maze)
    start, goal = find_start_goal(maze)
    if start is None or goal is None:
        print("Error: Start or Goal not found in the maze!")
    else:
```

```
        start_time = time.time()
        path, explored = a_star(maze, start, goal)
        if path:
            print("Path found!")
        else:
            print("No path found!")
        visualize_search(maze, path, explored)
```

APPENDIX 4

(Value Iteration)

```
import time
import sys
import csv
import tkinter as tk
import numpy as np
import random
from collections import defaultdict
# Global variable for visualization
cell_size = 30
total_time_taken = 0
def load_maze(filename="maze.txt"):
    """Reads the maze from a text file and returns it as a 2D list."""
    import os

    # Try multiple possible locations for the maze file
    possible_paths = [
        filename,  # Try the direct filename first (for command line use)
        os.path.join(os.path.dirname(os.path.abspath(__file__)), filename),  # Try
script directory
        os.path.join(os.getcwd(), filename)  # Try current working directory
    ]

    for path in possible_paths:
        try:
            with open(path, "r") as f:
                maze = [list(line.strip()) for line in f]
            print(f"Successfully loaded maze from: {path}")
            return maze
        except FileNotFoundError:
            continue

    # If we got here, we couldn't find the file
    raise FileNotFoundError(f"Could not find maze file '{filename}' in any expected
location")
def find_start_goal(maze):
    """Finds the start (S) and goal (G) positions in the maze."""
    start, goal = None, None
    for i in range(len(maze)):
        for j in range(len(maze[i])):
            if maze[i][j] == 'S':
                start = (i, j)
            elif maze[i][j] == 'G':
                goal = (i, j)
    return start, goal
```

```
def create_mdp_from_maze(maze, goal):
    """
    Creates a sparse MDP representation from the maze.
    Only valid states and actions are stored to optimize memory.
    """
    rows, cols = len(maze), len(maze[0])
    directions = [(0, 1, 'RIGHT'), (1, 0, 'DOWN'), (0, -1, 'LEFT'), (-1, 0, 'UP')]

    # Using defaultdict for sparse representation
    states = set()
    actions = {'UP', 'DOWN', 'LEFT', 'RIGHT'}
    transitions = defaultdict(list)
    rewards = defaultdict(float)
    # Terminal state
    goal_state = goal
    # Find all valid states
    for i in range(rows):
        for j in range(cols):
            if maze[i][j] != '#':  # If not a wall
                state = (i, j)
                states.add(state)
                # Set high reward for reaching the goal
                if state == goal_state:
                    rewards[(state, None)] = 100
                    continue  # No transitions from goal state
                # Default small negative reward for each step
                for _, _, action in directions:
                    rewards[(state, action)] = -1
                # Calculate transitions
                for di, dj, action in directions:
                    next_i, next_j = i + di, j + dj

                    # Check if the next position is valid
                    if (0 <= next_i < rows and 0 <= next_j < cols and
maze[next_i][next_j] != '#'):
                        next_state = (next_i, next_j)
                        # Deterministic transition
                        transitions[(state, action)].append((1.0, next_state))
                    else:
                        # Stay in place if hitting a wall
                        transitions[(state, action)].append((1.0, state))
    return states, actions, transitions, rewards, goal_state
def value_iteration(states, actions, transitions, rewards, goal_state, gamma=0.95,
epsilon=1e-4):
    """
    Implements Value Iteration algorithm with sparse representation.
    Enhanced with value propagation for better path extraction.
    """
    # Initialize value function with small random values to break symmetry
    V = {state: random.uniform(-0.1, 0.1) for state in states}
    V[goal_state] = 100  # Initialize goal state with reward
    explored = []
    policy = {}
    iteration = 0
    delta = float('inf')
    # Set maximum iterations to prevent infinite loops
```

```
    max_iterations = 1000
    print(f"Starting Value Iteration with gamma={gamma}, epsilon={epsilon}")
    while delta > epsilon and iteration < max_iterations:
        delta = 0
        iteration += 1
        for state in states:
            explored.append(state)
            if state == goal_state:
                continue  # Skip goal state
            # Keep track of the previous value
            v_old = V[state]
            # Compute the value for each action
            action_values = {}
            for action in actions:
                if (state, action) in transitions:
                    # Calculate expected value for this action
                    next_value = 0

                    # Track if this action leads to a new state
                    action_leads_to_new_state = False
                    for prob, next_state in transitions[(state, action)]:
                        if next_state != state:
                            action_leads_to_new_state = True
                        reward = rewards[(state, action)]
                        next_value += prob * (reward + gamma * V[next_state])
                    # Only consider actions that lead to a new state
                    if action_leads_to_new_state:
                        action_values[action] = next_value
            # If actions are available for this state
            if action_values:
                # Choose the action with max value
                best_action = max(action_values, key=action_values.get)
                V[state] = action_values[best_action]
                policy[state] = best_action
                # Update delta
                delta = max(delta, abs(v_old - V[state]))
        # Print progress every 10 iterations
        if iteration % 10 == 0:
            print(f"Value Iteration - Iteration {iteration}, Delta: {delta}")
    print(f"Value Iteration converged after {iteration} iterations")
    # Enhance value function by propagating goal values
    propagate_goal_values(V, transitions, goal_state, states, gamma)
    return policy, V, [], explored
def propagate_goal_values(V, transitions, goal_state, states, gamma, iterations=10):
    """Propagate high goal values backward to create a value gradient toward the
goal."""
    print("Propagating goal values to enhance value function...")
    # Map from state to actions that can be taken from it
    state_to_actions = defaultdict(list)
    for (state, action) in transitions:
        state_to_actions[state].append(action)
    # Map from state to states that can reach it (reverse transitions)
    reverse_transitions = defaultdict(list)
    for (state, action) in transitions:
        for _, next_state in transitions[(state, action)]:
            if next_state != state:  # Only consider actual moves
```

```
                reverse_transitions[next_state].append(state)
    # Start from goal and work backwards
    current_states = {goal_state}
    visited = set()
    for _ in range(iterations):
        next_states = set()
        for state in current_states:
            visited.add(state)
            for prev_state in reverse_transitions[state]:
                if prev_state not in visited:
                    # Update value based on best neighbor
                    best_val = float('-inf')
                    for action in state_to_actions[prev_state]:
                        for _, next_s in transitions[(prev_state, action)]:
                            if next_s != prev_state:  # Only consider actual moves
                                val = -1 + gamma * V[next_s]  # Simple reward of -1 for
each step
                                best_val = max(best_val, val)

                    if best_val > V[prev_state]:
                        V[prev_state] = best_val

                    next_states.add(prev_state)
        if not next_states:
            break
        current_states = next_states
    print("Value propagation complete")
def extract_path(start, goal, policy, maze, value_function):
    """Extracts a path from start to goal using the policy with enhanced path
finding."""
    print("Extracting path from start to goal...")
    path = [start]
    current = start
    directions = {'UP': (-1, 0), 'DOWN': (1, 0), 'LEFT': (0, -1), 'RIGHT': (0, 1)}
    # Maximum steps as a safety measure
    max_steps = len(maze) * len(maze[0]) * 2
    steps = 0
    # Set to track visited states to avoid cycles
    visited = {start}
    while current != goal and steps < max_steps:
        next_state = None

        # First, try to use the policy if available
        if current in policy:
            action = policy[current]
            di, dj = directions[action]
            ni, nj = current[0] + di, current[1] + dj
            if (0 <= ni < len(maze) and 0 <= nj < len(maze[0]) and
                maze[ni][nj] != '#' and (ni, nj) not in visited):
                next_state = (ni, nj)
                # If policy doesn't work, use value function to guide us
        if next_state is None:
            best_value = float('-inf')
            best_next = None
            for action, (di, dj) in directions.items():
                ni, nj = current[0] + di, current[1] + dj
```

```
                if (0 <= ni < len(maze) and 0 <= nj < len(maze[0]) and
                    maze[ni][nj] != '#' and (ni, nj) not in visited):
                    next_val = value_function.get((ni, nj), float('-
inf'))
                    if next_val > best_value:
                        best_value = next_val
                        best_next = (ni, nj)
            next_state = best_next
        # If we still don't have a next state, try to find any valid move
        if next_state is None:
            for di, dj in directions.values():
                ni, nj = current[0] + di, current[1] + dj

                if (0 <= ni < len(maze) and 0 <= nj < len(maze[0]) and
                    maze[ni][nj] != '#' and (ni, nj) not in visited):
                    next_state = (ni, nj)
                    break

        # If we have a valid next state, move there
        if next_state:
            current = next_state
            path.append(current)
            visited.add(current)
        else:
            # If we're stuck, try backtracking along the path
            if len(path) > 1:
                path.pop()  # Remove current
                current = path[-1]  # Go back to previous
                print(f"Backtracking to {current}")
            else:
                print("Cannot find path to goal")                break
        steps += 1
        # If we're getting close to the maximum steps, print debug info
        if steps >= max_steps - 10:
            print(f"Warning: Approaching maximum steps. Current position: {current},
Goal: {goal}")
    if current == goal:
        print(f"Path found with {len(path)} steps")
    else:
        print("Failed to reach goal within step limit")
    return path
def calculate_memory_usage(value_function, policy):
    """Calculates memory usage for the value function and policy dictionaries."""
    memory_usage = sys.getsizeof(value_function) + sys.getsizeof(policy)
    # Add memory for dictionary entries
    for state, value in value_function.items():
        memory_usage += sys.getsizeof(state) + sys.getsizeof(value)
    for state, action in policy.items():
        memory_usage += sys.getsizeof(state) + sys.getsizeof(action)
    return memory_usage
def validate_policy(policy, maze):
    """Validates and fixes policy to ensure it doesn't lead to walls or out of
bounds."""
    rows, cols = len(maze), len(maze[0])
    directions = {'UP': (-1, 0), 'DOWN': (1, 0), 'LEFT': (0, -1), 'RIGHT': (0, 1)}
```

```
    fixed_policy = policy.copy()
    fixed_count = 0
    for state, action in policy.items():
        i, j = state
        di, dj = directions[action]
        next_i, next_j = i + di, j + dj
        # Check if action leads to a valid cell
        if (next_i < 0 or next_i >= rows or
            next_j < 0 or next_j >= cols or
            maze[next_i][next_j] == '#'):
            # Find valid alternatives
            valid_actions = []
            for alt_action, (di, dj) in directions.items():
                ni, nj = i + di, j + dj
                if (0 <= ni < rows and 0 <= nj < cols and maze[ni][nj] != '#'):
                    valid_actions.append(alt_action)
            if valid_actions:
                # Choose an alternative action
                fixed_policy[state] = valid_actions[0]
                fixed_count += 1
                print(f"Fixed policy at {state}: {action} -> {valid_actions[0]}")
            else:
                # No valid actions, remove from policy
                del fixed_policy[state]
                print(f"Removed invalid state {state} from policy")
    print(f"Fixed {fixed_count} invalid policy actions")
    return fixed_policy
def save_metrics_to_csv(metrics, filename="metrics.csv"):
    """Saves the metrics to a CSV file with the specified format."""
    # Check if the file exists to write the header
    try:
        with open(filename, 'r') as file:
            pass  # File exists, no need to write header
    except FileNotFoundError:
        # File doesn't exist, write the header
        with open(filename, mode='w', newline='') as file:
            writer = csv.writer(file)
            writer.writerow(["Algorithm", "Search Type", "Path Length", "Cells
Explored", "Memory (KB)", "Time (Seconds)"])
    # Append the metrics to the CSV file
    with open(filename, mode='a', newline='') as file:
        writer = csv.writer(file)
        writer.writerow(metrics)
def visualize_search(maze, path, explored, value_function=None):
    """Creates a Tkinter GUI to visualize the maze solving process with a size-limited
window."""
    global total_time_taken
    window = tk.Tk()
    window.title("Value Iteration Maze Solver")
    # Get screen dimensions
    screen_width = window.winfo_screenwidth()
    screen_height = window.winfo_screenheight()
    # Calculate maze dimensions
    maze_height = len(maze)
    maze_width = len(maze[0])
    # Adjust cell size for large mazes
```

```
    global cell_size
    original_cell_size = cell_size
    # Calculate maximum window size (70% of screen)
    max_width = int(screen_width * 0.7)
    max_height = int(screen_height * 0.7)
    # Calculate what cell size would fit in max dimensions
    width_cell_size = max_width // maze_width
    height_cell_size = max_height // maze_height
    # Take the smaller of the two to ensure it fits both dimensions
    if maze_width > 50 or maze_height > 50:
        cell_size = min(width_cell_size, height_cell_size, original_cell_size)
        print(f"Adjusted cell size to {cell_size} for large maze")
    # Calculate canvas dimensions
    canvas_width = maze_width * cell_size
    canvas_height = maze_height * cell_size
    # Ensure window fits on screen
    total_height = canvas_height + 150  # Add space for info panel
    # Set window size and position it centered
    window.geometry(f"{canvas_width}x{total_height}")
    window.update_idletasks()  # Update to get actual window size
    # Center the window
    x_position = (screen_width - window.winfo_width()) // 2
    y_position = (screen_height - window.winfo_height()) // 2
    window.geometry(f"+{x_position}+{y_position}")
    # Create canvas for maze with scrollbars for very large mazes
    frame = tk.Frame(window)
    frame.pack(fill="both", expand=True)
    # Add scrollbars if the maze is large
    if canvas_width > max_width or canvas_height > max_height:
        # Create scrollbars
        h_scrollbar = tk.Scrollbar(frame, orient="horizontal")
        v_scrollbar = tk.Scrollbar(frame, orient="vertical")
        # Position scrollbars
        h_scrollbar.pack(side="bottom", fill="x")
        v_scrollbar.pack(side="right", fill="y")
        # Create canvas with scrollbars
        canvas = tk.Canvas(frame, width=min(canvas_width, max_width),
                           height=min(canvas_height, max_height - 150),
                           bg="white",
                           xscrollcommand=h_scrollbar.set,
                           yscrollcommand=v_scrollbar.set)
        # Configure scrollbars
        h_scrollbar.config(command=canvas.xview)
        v_scrollbar.config(command=canvas.yview)
        # Configure canvas scroll region
        canvas.config(scrollregion=(0, 0, canvas_width, canvas_height))
    else:
        # Create canvas without scrollbars for smaller mazes
        canvas = tk.Canvas(frame, width=canvas_width, height=canvas_height, bg="white")
    canvas.pack(side="left", fill="both", expand=True)
    # Create info panel below the canvas
    info_panel = tk.Frame(window, height=150, bg="white")
    info_panel.pack(fill="x")
    # Title label
    title_label = tk.Label(info_panel, text="Value Iteration Maze Solver",
font=("Arial", 16, "bold"), bg="white")
```

```
    title_label.pack()
    # Create a frame for the legend
    legend_frame = tk.Frame(info_panel, bg="white")
    legend_frame.pack()
    # Create legend items
    legend_explored = tk.Canvas(legend_frame, width=20, height=20, bg="gray")
    legend_explored.grid(row=0, column=0, padx=5)
    legend_label_explored = tk.Label(legend_frame, text="Explored Nodes",
font=("Arial", 12), bg="white")
    legend_label_explored.grid(row=0, column=1)
    legend_path = tk.Canvas(legend_frame, width=20, height=20, bg="orange")
    legend_path.grid(row=0, column=2, padx=5)
    legend_label_path = tk.Label(legend_frame, text="Optimal Path", font=("Arial", 12),
bg="white")
    legend_label_path.grid(row=0, column=3)
    legend_start = tk.Canvas(legend_frame, width=20, height=20, bg="green")
    legend_start.grid(row=0, column=4, padx=5)
    legend_label_start = tk.Label(legend_frame, text="Start Position", font=("Arial",
12), bg="white")
    legend_label_start.grid(row=0, column=5)
    legend_goal = tk.Canvas(legend_frame, width=20, height=20, bg="red")
    legend_goal.grid(row=0, column=6, padx=5)
    legend_label_goal = tk.Label(legend_frame, text="Goal Position", font=("Arial",
12), bg="white")
    legend_label_goal.grid(row=0, column=7)
    # Metrics Label (Initially Empty)
    metrics_label = tk.Label(info_panel, text="Metrics", font=("Arial", 12, "bold"),
fg="#2E86C1", bg="white")
    metrics_label.pack()
    def draw_cell(x, y, color):
        """Draws a single cell on the canvas."""
        x1, y1 = y * cell_size, x * cell_size
        x2, y2 = x1 + cell_size, y1 + cell_size
        canvas.create_rectangle(x1, y1, x2, y2, fill=color, outline="black")
        # Display value if available and cell size is big enough
        if value_function and (x, y) in value_function and cell_size >= 20:
            value = value_function[(x, y)]
            if abs(value) < 1000:  # Only show reasonable values
                canvas.create_text((x1 + x2) / 2, (y1 + y2) / 2,
                                  text=f"{value:.1f}", font=("Arial", 8))
    # Draw the initial maze layout
    for i in range(len(maze)):
        for j in range(len(maze[0])):
            color = "white"
            if maze[i][j] == '#':
                color = "black"
            elif maze[i][j] == 'S':
                color = "green"
            elif maze[i][j] == 'G':
                color = "red"
            draw_cell(i, j, color)
    # Use unique states for visualization
    unique_explored = list(dict.fromkeys(explored))
    explored_idx = 0
    def animate_explored():
        """Animates the explored cells."""
```

```
        nonlocal explored_idx
        if explored_idx < len(unique_explored):
            x, y = unique_explored[explored_idx]
            if maze[x][y] not in ['S', 'G']:  # Don't color start/goal
                draw_cell(x, y, "gray")
            explored_idx += 1            window.after(5, animate_explored)  # Speed up
for large mazes
    path_idx = 0
    def animate_path():
        """Animates the optimal path."""
        nonlocal path_idx
        if path_idx < len(path):
            x, y = path[path_idx]
            if maze[x][y] not in ['S', 'G']:  # Don't color start/goal
                draw_cell(x, y, "orange")
            path_idx += 1
            window.after(50, animate_path)  # Speed up for large mazes
        else:
            # Final time calculation
            elapsed_time = time.time() - start_time
            global total_time_taken
            total_time_taken = elapsed_time
            # Save metrics
            memory_usage = calculate_memory_usage(value_function or {}, {})
            # Update metrics in info panel
            metrics_text = f"📊 Metrics:\nCells Explored: {len(unique_explored)}\nPath
Length: {len(path)}\nMemory Usage: {memory_usage / 1024:.2f} KB\nExecution Time:
{total_time_taken:.2f} sec"
            metrics_label.config(text=metrics_text, font=("Arial", 12), fg="black")
            # Save metrics to CSV file
            save_metrics_to_csv(["Value Iteration", "MDP", len(path),
len(unique_explored), memory_usage / 1024, total_time_taken])
    start_time = time.time()
    animate_explored()
    def start_path_animation():
        """Starts the path animation after the exploration animation is done."""
        animation_delay = 5 * len(unique_explored)  # Speed up for large mazes
        window.after(animation_delay, animate_path)
    window.after(5 * len(unique_explored), start_path_animation)  # Speed up for large
mazes
    window.mainloop()
# Main execution
if __name__ == "__main__":
    import argparse
    parser = argparse.ArgumentParser(description='Run Value Iteration on a maze.')
    parser.add_argument('--maze', type=str, default='maze.txt', help='The maze file to
use')
    parser.add_argument('--gamma', type=float, default=0.95, help='Discount factor
(default: 0.95)')
    parser.add_argument('--epsilon', type=float, default=1e-4, help='Convergence
threshold (default: 1e-4)')
    args = parser.parse_args()
    print("Running Value Iteration...")
    maze = load_maze(args.maze)
    start, goal = find_start_goal(maze)
```

```
    if start is None or goal is None:
        print("Error: Start or Goal not found in the maze!")
    else:
        # Create MDP from maze
        states, actions, transitions, rewards, goal_state = create_mdp_from_maze(maze,
goal)
                # Run Value Iteration with specified parameters
        start_time = time.time()
        policy, value_function, _, explored = value_iteration(
            states, actions, transitions, rewards, goal_state,
            gamma=args.gamma, epsilon=args.epsilon
        )
                # Validate policy before path extraction
        policy = validate_policy(policy, maze)
        # Extract path using policy and value function
        path = extract_path(start, goal, policy, maze, value_function)
        # Visualize
        visualize_search(maze, path, explored, value_function)
```

APPENDIX 5

(Policy Iteration)

```
import time
import sys
import csv
import tkinter as tk
import numpy as np
import random
from collections import defaultdict
# Global variable for visualization
cell_size = 30
total_time_taken = 0
def load_maze(filename="maze.txt"):
    """Reads the maze from a text file and returns it as a 2D list."""
    import os

    # Try multiple possible locations for the maze file
    possible_paths = [
        filename,  # Try the direct filename first (for command line use)
        os.path.join(os.path.dirname(os.path.abspath(__file__)), filename),  # Try
script directory
        os.path.join(os.getcwd(), filename)  # Try current working directory
    ]

    for path in possible_paths:
        try:
            with open(path, "r") as f:
                maze = [list(line.strip()) for line in f]
            print(f"Successfully loaded maze from: {path}")
            return maze
        except FileNotFoundError:
            continue

    # If we got here, we couldn't find the file
```

```
    raise FileNotFoundError(f"Could not find maze file '{filename}' in any expected
location")
def find_start_goal(maze):
    """Finds the start (S) and goal (G) positions in the maze."""
    start, goal = None, None
    for i in range(len(maze)):
        for j in range(len(maze[i])):
            if maze[i][j] == 'S':
                start = (i, j)
            elif maze[i][j] == 'G':
                goal = (i, j)
    return start, goal
def create_mdp_from_maze(maze, goal):
    """
    Creates a sparse MDP representation from the maze.
    Only valid states and actions are stored to optimize memory.
    """
    rows, cols = len(maze), len(maze[0])
    directions = [(0, 1, 'RIGHT'), (1, 0, 'DOWN'), (0, -1, 'LEFT'), (-1, 0, 'UP')]

    # Using defaultdict for sparse representation
    states = set()
    actions = {'UP', 'DOWN', 'LEFT', 'RIGHT'}
    transitions = defaultdict(list)
    rewards = defaultdict(float)

    # Terminal state
    goal_state = goal

    # Find all valid states
    for i in range(rows):
        for j in range(cols):
            if maze[i][j] != '#':  # If not a wall
                state = (i, j)
                states.add(state)

                # Set high reward for reaching the goal
                if state == goal_state:
                    rewards[(state, None)] = 100
                    continue  # No transitions from goal state

                # Default small negative reward for each step
                for _, _, action in directions:
                    rewards[(state, action)] = -1

                # Calculate transitions
                for di, dj, action in directions:
                    next_i, next_j = i + di, j + dj

                    # Check if the next position is valid
                    if (0 <= next_i < rows and 0 <= next_j < cols and
maze[next_i][next_j] != '#'):
                        next_state = (next_i, next_j)

                        # Deterministic transition
                        transitions[(state, action)].append((1.0, next_state))
```

```
                    else:
                        # Stay in place if hitting a wall
                        transitions[(state, action)].append((1.0, state))

    return states, actions, transitions, rewards, goal_state
def policy_iteration(states, actions, transitions, rewards, goal_state, gamma=0.9,
epsilon=1e-6):
    """
    Implements Policy Iteration algorithm with sparse representation.
    Only valid states and actions are processed to optimize memory.
    """
    # Initialize policy randomly but only with valid actions
    policy = {}
    for state in states:
        if state != goal_state:
            valid_actions = []
            for action in actions:
                if (state, action) in transitions:
                    for prob, next_state in transitions[(state, action)]:
                        if next_state != state:  # Only add actions that move to new
states
                            valid_actions.append(action)
                            break

            if valid_actions:  # Only assign policy if there are valid actions
                policy[state] = random.choice(valid_actions)

    # Initialize value function
    V = {state: 0 for state in states}
    V[goal_state] = 100  # Initialize goal state with reward

    explored = []
    iteration = 0
    policy_stable = False

    # Improved convergence criteria
    max_iterations = 100  # Limit maximum iterations

    while not policy_stable and iteration < max_iterations:
        iteration += 1
        print(f"Policy Iteration - Iteration {iteration}")

        # Policy Evaluation
        # Iteratively evaluate the current policy until convergence
        for eval_iter in range(100):  # Limit iterations for policy evaluation
            delta = 0
            for state in states:
                explored.append(state)

                if state == goal_state:
                    continue  # Skip goal state

                old_value = V[state]

                if state in policy:
                    action = policy[state]
```

```
                # Calculate new value based on current policy
                new_value = 0
                if (state, action) in transitions:
                    for prob, next_state in transitions[(state, action)]:
                        reward = rewards[(state, action)]
                        new_value += prob * (reward + gamma * V[next_state])
                V[state] = new_value

                delta = max(delta, abs(old_value - V[state]))

        if delta < epsilon:
            print(f"  Policy evaluation converged after {eval_iter+1} iterations")
            break

    # Policy Improvement
    policy_stable = True
    for state in states:
        if state == goal_state:
            continue  # Skip goal state

        old_action = policy.get(state)

        # Find the best action for the current state
        best_value = float('-inf')
        best_action = None

        for action in actions:
            if (state, action) in transitions:
                # Calculate value for this action
                value = 0
                action_leads_to_new_state = False

                for prob, next_state in transitions[(state, action)]:
                    if next_state != state:
                        action_leads_to_new_state = True
                    reward = rewards[(state, action)]
                    value += prob * (reward + gamma * V[next_state])

                # Only consider actions that lead to a new state
                if action_leads_to_new_state and value > best_value:
                    best_value = value
                    best_action = action

        # Only update policy if we found a valid action
        if best_action:
            policy[state] = best_action

            # Check if policy changed
            if old_action != best_action:
                policy_stable = False

    # Print current policy for debugging
    if iteration % 10 == 0:
        print(f"Current memory usage: {calculate_memory_usage(V, policy) /
(1024*1024):.2f} MB")
```

```
    print(f"Policy Iteration completed after {iteration} iterations")

    # Enhanced value function - make sure values to goal are higher
    # This creates a gradient toward the goal to help with path finding
    propagate_goal_values(V, transitions, goal_state, states, gamma)

    return policy, V, [], explored
def propagate_goal_values(V, transitions, goal_state, states, gamma, iterations=10):
    """Propagate high goal values backward to create a value gradient toward the
goal."""
    print("Propagating goal values to enhance value function...")

    # Map from state to actions that can be taken from it
    state_to_actions = defaultdict(list)
    for (state, action) in transitions:
        state_to_actions[state].append(action)

    # Map from state to states that can reach it (reverse transitions)
    reverse_transitions = defaultdict(list)
    for (state, action) in transitions:
        for _, next_state in transitions[(state, action)]:
            if next_state != state:  # Only consider actual moves
                reverse_transitions[next_state].append(state)

    # Start from goal and work backwards
    current_states = {goal_state}
    visited = set()

    for _ in range(iterations):
        next_states = set()
        for state in current_states:
            visited.add(state)
            for prev_state in reverse_transitions[state]:
                if prev_state not in visited:
                    # Update value based on best neighbor
                    best_val = float('-inf')
                    for action in state_to_actions[prev_state]:
                        for _, next_s in transitions[(prev_state, action)]:
                            if next_s != prev_state:  # Only consider actual moves
                                val = -1 + gamma * V[next_s]  # Simple reward of -1 for
each step
                                best_val = max(best_val, val)

                    if best_val > V[prev_state]:
                        V[prev_state] = best_val

                    next_states.add(prev_state)

        if not next_states:
            break

        current_states = next_states

    print("Value propagation complete")
def extract_path(start, goal, policy, maze, value_function):
```

```
    """Extracts a path from start to goal using the policy with enhanced path
finding."""
    print("Extracting path from start to goal...")

    path = [start]
    current = start
    directions = {'UP': (-1, 0), 'DOWN': (1, 0), 'LEFT': (0, -1), 'RIGHT': (0, 1)}

    # Maximum steps as a safety measure
    max_steps = len(maze) * len(maze[0]) * 2
    steps = 0

    # Set to track visited states to avoid cycles
    visited = {start}

    while current != goal and steps < max_steps:
        next_state = None

        # First, try to use the policy if available
        if current in policy:
            action = policy[current]
            di, dj = directions[action]
            ni, nj = current[0] + di, current[1] + dj

            if (0 <= ni < len(maze) and 0 <= nj < len(maze[0]) and
                maze[ni][nj] != '#' and (ni, nj) not in visited):
                next_state = (ni, nj)

        # If policy doesn't work, use value function to guide us
        if next_state is None:
            best_value = float('-inf')
            best_next = None

            for action, (di, dj) in directions.items():
                ni, nj = current[0] + di, current[1] + dj

                if (0 <= ni < len(maze) and 0 <= nj < len(maze[0]) and
                    maze[ni][nj] != '#' and (ni, nj) not in visited):
                    next_val = value_function.get((ni, nj), float('-inf'))

                    if next_val > best_value:
                        best_value = next_val
                        best_next = (ni, nj)

            next_state = best_next

        # If we still don't have a next state, try to find any valid move
        if next_state is None:
            for di, dj in directions.values():
                ni, nj = current[0] + di, current[1] + dj

                if (0 <= ni < len(maze) and 0 <= nj < len(maze[0]) and
                    maze[ni][nj] != '#' and (ni, nj) not in visited):
                    next_state = (ni, nj)
                    break
```

```
        # If we have a valid next state, move there
        if next_state:
            current = next_state
            path.append(current)
            visited.add(current)
        else:
            # If we're stuck, try backtracking along the path
            if len(path) > 1:
                path.pop()  # Remove current
                current = path[-1]  # Go back to previous
                print(f"Backtracking to {current}")
            else:
                print("Cannot find path to goal")
                break

        steps += 1

        # If we're getting close to the maximum steps, print debug info
        if steps >= max_steps - 10:
            print(f"Warning: Approaching maximum steps. Current position: {current}, 
Goal: {goal}")

    if current == goal:
        print(f"Path found with {len(path)} steps")
    else:
        print("Failed to reach goal within step limit")

    return path
def calculate_memory_usage(value_function, policy):
    """Calculates memory usage for the value function and policy dictionaries."""
    memory_usage = sys.getsizeof(value_function) + sys.getsizeof(policy)

    # Add memory for dictionary entries
    for state, value in value_function.items():
        memory_usage += sys.getsizeof(state) + sys.getsizeof(value)

    for state, action in policy.items():
        memory_usage += sys.getsizeof(state) + sys.getsizeof(action)

    return memory_usage
def save_metrics_to_csv(metrics, filename="metrics.csv"):
    """Saves the metrics to a CSV file with the specified format."""
    # Check if the file exists to write the header
    try:
        with open(filename, 'r') as file:
            pass  # File exists, no need to write header
    except FileNotFoundError:
        # File doesn't exist, write the header
        with open(filename, mode='w', newline='') as file:
            writer = csv.writer(file)
            writer.writerow(["Algorithm", "Search Type", "Path Length", "Cells 
Explored", "Memory (KB)", "Time (Seconds)"])
    # Append the metrics to the CSV file
    with open(filename, mode='a', newline='') as file:
        writer = csv.writer(file)
        writer.writerow(metrics)
```

```
def visualize_search(maze, path, explored, value_function=None):
    """Creates a Tkinter GUI to visualize the maze solving process with a size-limited
window."""
    global total_time_taken
    window = tk.Tk()
    window.title("Policy Iteration Maze Solver")
    # Get screen dimensions
    screen_width = window.winfo_screenwidth()
    screen_height = window.winfo_screenheight()

    # Calculate maze dimensions
    maze_height = len(maze)
    maze_width = len(maze[0])

    # Adjust cell size for large mazes
    global cell_size
    original_cell_size = cell_size

    # Calculate maximum window size (70% of screen)
    max_width = int(screen_width * 0.7)
    max_height = int(screen_height * 0.7)

    # Calculate what cell size would fit in max dimensions
    width_cell_size = max_width // maze_width
    height_cell_size = max_height // maze_height

    # Take the smaller of the two to ensure it fits both dimensions
    if maze_width > 50 or maze_height > 50:
        cell_size = min(width_cell_size, height_cell_size, original_cell_size)
        print(f"Adjusted cell size to {cell_size} for large maze")

    # Calculate canvas dimensions
    canvas_width = maze_width * cell_size
    canvas_height = maze_height * cell_size
    # Ensure window fits on screen
    total_height = canvas_height + 150  # Add space for info panel

    # Set window size and position it centered
    window.geometry(f"{canvas_width}x{total_height}")
    window.update_idletasks()  # Update to get actual window size

    # Center the window
    x_position = (screen_width - window.winfo_width()) // 2
    y_position = (screen_height - window.winfo_height()) // 2
    window.geometry(f"+{x_position}+{y_position}")
    # Create canvas for maze with scrollbars for very large mazes
    frame = tk.Frame(window)
    frame.pack(fill="both", expand=True)

    # Add scrollbars if the maze is large
    if canvas_width > max_width or canvas_height > max_height:
        # Create scrollbars
        h_scrollbar = tk.Scrollbar(frame, orient="horizontal")
        v_scrollbar = tk.Scrollbar(frame, orient="vertical")

        # Position scrollbars
```

```
        h_scrollbar.pack(side="bottom", fill="x")
        v_scrollbar.pack(side="right", fill="y")

        # Create canvas with scrollbars
        canvas = tk.Canvas(frame, width=min(canvas_width, max_width),
                           height=min(canvas_height, max_height - 150),
                           bg="white",
                           xscrollcommand=h_scrollbar.set,
                           yscrollcommand=v_scrollbar.set)

        # Configure scrollbars
        h_scrollbar.config(command=canvas.xview)
        v_scrollbar.config(command=canvas.yview)

        # Configure canvas scroll region
        canvas.config(scrollregion=(0, 0, canvas_width, canvas_height))
    else:
        # Create canvas without scrollbars for smaller mazes
        canvas = tk.Canvas(frame, width=canvas_width, height=canvas_height, bg="white")

    canvas.pack(side="left", fill="both", expand=True)
    # Create info panel below the canvas
    info_panel = tk.Frame(window, height=150, bg="white")
    info_panel.pack(fill="x")
    # Title label
    title_label = tk.Label(info_panel, text="Policy Iteration Maze Solver",
font=("Arial", 16, "bold"), bg="white")
    title_label.pack()
    # Create a frame for the legend
    legend_frame = tk.Frame(info_panel, bg="white")
    legend_frame.pack()
    # Create legend items
    legend_explored = tk.Canvas(legend_frame, width=20, height=20, bg="gray")
    legend_explored.grid(row=0, column=0, padx=5)
    legend_label_explored = tk.Label(legend_frame, text="Explored Nodes",
font=("Arial", 12), bg="white")
    legend_label_explored.grid(row=0, column=1)
    legend_path = tk.Canvas(legend_frame, width=20, height=20, bg="orange")
    legend_path.grid(row=0, column=2, padx=5)
    legend_label_path = tk.Label(legend_frame, text="Optimal Path", font=("Arial", 12),
bg="white")
    legend_label_path.grid(row=0, column=3)
    legend_start = tk.Canvas(legend_frame, width=20, height=20, bg="green")
    legend_start.grid(row=0, column=4, padx=5)
    legend_label_start = tk.Label(legend_frame, text="Start Position", font=("Arial",
12), bg="white")
    legend_label_start.grid(row=0, column=5)
    legend_goal = tk.Canvas(legend_frame, width=20, height=20, bg="red")
    legend_goal.grid(row=0, column=6, padx=5)
    legend_label_goal = tk.Label(legend_frame, text="Goal Position", font=("Arial",
12), bg="white")
    legend_label_goal.grid(row=0, column=7)
    # Metrics Label (Initially Empty)
    metrics_label = tk.Label(info_panel, text="Metrics", font=("Arial", 12, "bold"),
fg="#2E86C1", bg="white")
    metrics_label.pack()
```

```
    def draw_cell(x, y, color):
        """Draws a single cell on the canvas."""
        x1, y1 = y * cell_size, x * cell_size
        x2, y2 = x1 + cell_size, y1 + cell_size
        canvas.create_rectangle(x1, y1, x2, y2, fill=color, outline="black")

        # Display value if available and cell size is big enough
        if value_function and (x, y) in value_function and cell_size >= 20:
            value = value_function[(x, y)]
            if abs(value) < 1000:  # Only show reasonable values
                canvas.create_text((x1 + x2) / 2, (y1 + y2) / 2,
                                  text=f"{value:.1f}", font=("Arial", 8))
    # Draw the initial maze layout
    for i in range(len(maze)):
        for j in range(len(maze[0])):
            color = "white"
            if maze[i][j] == '#':
                color = "black"
            elif maze[i][j] == 'S':
                color = "green"
            elif maze[i][j] == 'G':
                color = "red"
            draw_cell(i, j, color)
    # Use unique states for visualization
    unique_explored = list(dict.fromkeys(explored))
    explored_idx = 0
    def animate_explored():
        """Animates the explored cells."""
        nonlocal explored_idx
        if explored_idx < len(unique_explored):
            x, y = unique_explored[explored_idx]
            if maze[x][y] not in ['S', 'G']:  # Don't color start/goal
                draw_cell(x, y, "gray")
            explored_idx += 1
            window.after(5, animate_explored)  # Speed up for large mazes
    path_idx = 0
    def animate_path():
        """Animates the optimal path."""
        nonlocal path_idx
        if path_idx < len(path):
            x, y = path[path_idx]
            if maze[x][y] not in ['S', 'G']:  # Don't color start/goal
                draw_cell(x, y, "orange")
            path_idx += 1
            window.after(50, animate_path)  # Speed up for large mazes
        else:
            # Final time calculation
            elapsed_time = time.time() - start_time
            global total_time_taken
            total_time_taken = elapsed_time
            # Save metrics
            memory_usage = calculate_memory_usage(value_function or {}, {})
            # Update metrics in info panel
            metrics_text = f"📊 Metrics:\nCells Explored: {len(unique_explored)}\nPath
Length: {len(path)}\nMemory Usage: {memory_usage / 1024:.2f} KB\nExecution Time:
{total_time_taken:.2f} sec"
```

```
            metrics_label.config(text=metrics_text, font=("Arial", 12), fg="black")
            # Save metrics to CSV file
            save_metrics_to_csv(["Policy Iteration", "MDP", len(path),
len(unique_explored), memory_usage / 1024, total_time_taken])
    start_time = time.time()
    animate_explored()
    def start_path_animation():
        """Starts the path animation after the exploration animation is done."""
        animation_delay = 5 * len(unique_explored)  # Speed up for large mazes
        window.after(animation_delay, animate_path)
    window.after(5 * len(unique_explored), start_path_animation)  # Speed up for large
mazes
    window.mainloop()

# Main execution
if __name__ == "__main__":
    import argparse

    parser = argparse.ArgumentParser(description='Run Policy Iteration on a maze.')
    parser.add_argument('--maze', type=str, default='maze.txt', help='The maze file to
use')
    parser.add_argument('--gamma', type=float, default=0.95, help='Discount factor
(default: 0.95)')
    parser.add_argument('--epsilon', type=float, default=1e-4, help='Convergence
threshold (default: 1e-4)')

    args = parser.parse_args()

    print("Running Policy Iteration...")
    maze = load_maze(args.maze)
    start, goal = find_start_goal(maze)

    if start is None or goal is None:
        print("Error: Start or Goal not found in the maze!")
    else:
        # Create MDP from maze
        states, actions, transitions, rewards, goal_state = create_mdp_from_maze(maze,
goal)

        # Run Policy Iteration with specified parameters
        start_time = time.time()
        policy, value_function, _, explored = policy_iteration(
            states, actions, transitions, rewards, goal_state,
            gamma=args.gamma, epsilon=args.epsilon
        )

        # Extract path using policy and value function
        path = extract_path(start, goal, policy, maze, value_function)

        # Visualize
        visualize_search(maze, path, explored, value_function)
```