CS7IS2 – Artificial Intelligence – Assignment 1

Sibin Babu George 24335327 MSc Computer Science (Future Networked Systems)

1. INTRODUCTION

In this assignment, I explore the implementation and performance analysis of various search and Markov Decision Process (MDP) algorithms for maze solving. I design a custom maze generator that produces mazes of different sizes, and I implement classic search algorithms like Depth-First Search (DFS), Breadth-First Search (BFS), and A*. I also implement MDP-based methods like value iteration and policy iteration. I apply these algorithms to randomly generated mazes of different sizes and compare their performance over 5 iterations. I collect plots as images and results as a csv file which contains performanc evaluation over 5 iterations. I provide a comprehensive report that includes design justifications and visual comparisons.

2. IMPLEMENTATION

A) MAZE GENERATOR

The maze generator which I implemented for this assignment is encapsulated in a single class called Maze. This class manages both the maze's properties and its generation logic. It allows me to generate mazes of varying sizes, which is a core requirement of the assignment. The Maze class constructor initializes the maze by defining the number of cells horizontally and vertically. It then calculates the grid dimensions as (2cells_h + 1) by (2cells_w + 1). The maze is represented as a 2D NumPy array, where a value of 1 indicates a wall and a value of 0 represents a free cell. This setup creates a grid with walls that surround and separate the cells.

Eg of Maze: 5x5 i.e 2*5+1, 2*5+1 = 11x11 maze, 25x25 and 50x50 mazes are shown in below figures.

The maze generation process starts by the generate method, which uses an iterative Depth-First Search (DFS) algorithm using stack. The algorithm starts from cell (0, 0), which is mapped to grid coordinates (1, 1), and marks it as visited. It then selects a random unvisited neighboring cell, removes the wall between the current cell and the neighbor, and continues this process until all cells have been visited. This approach creates a "perfect maze" that has exactly one unique path between any two cells. The final maze is returned as a NumPy array, which provides a clean and efficient representation for further analysis.

To test the implementation, I save the generated maze grid into a CSV file. In figure 4 we can With the structure of maze. With this I can test search and MDP algorithms on custom mazes. By generating perfect mazes with only one solution, it would be a proper baseline metric to evaluate the performance of algorithms like DFS, BFS, A*, value iteration, and policy iteration. The use of a NumPy array is better for computation.

Note:- The maze is created assuming that that start position will be 1,1 and goal state will be n-1,n-1 where n is 2*(maze_cell_size) - 1

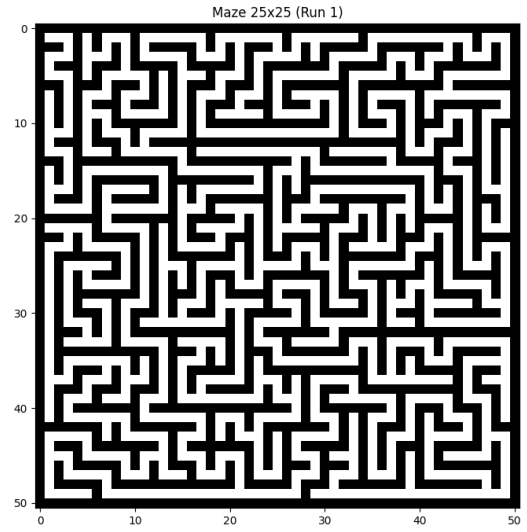


Fig (1)

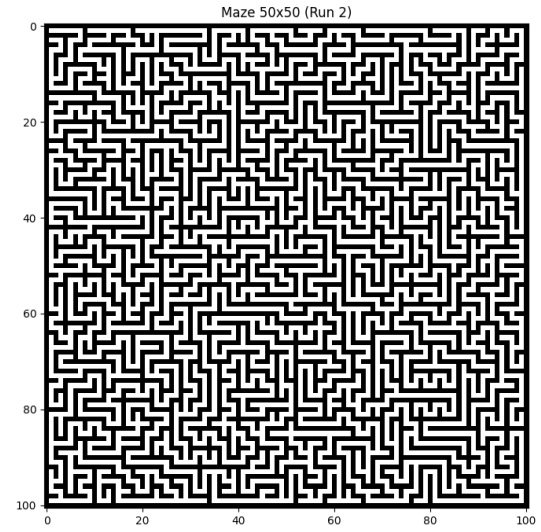


Fig (2)

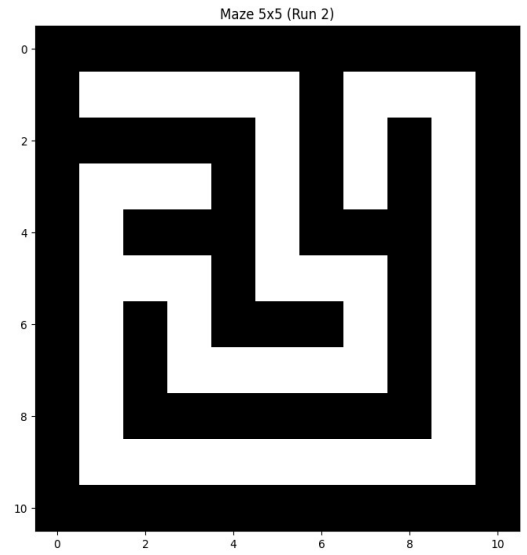


Fig (3)

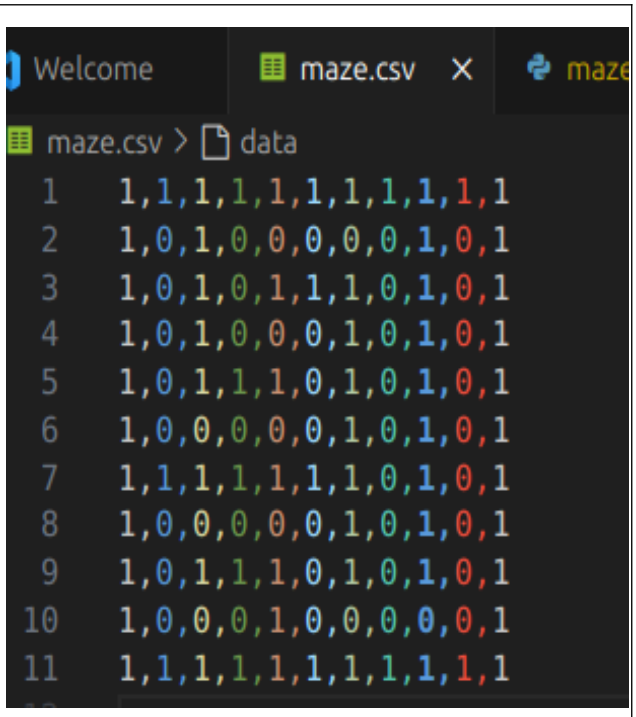


Fig (4) – Example of Maze in CSV

References:-

- https://github.com/Karthick47v2/maze-gen-and-solver/blob/main/src/MazeGenerator.cpp (C++ code)
- https://github.com/AzizZayed/Maze-Generator-and-Solver.git

B) SEARCH ALGORITHMS

DEPTH-FIRST SEARCH (DFS)

Depth-First Search (DFS) is an uninformed search algorithm that focuses on exploring the deepest unexpanded node in the search space. I implemented DFS using a Last-In-First-Out (LIFO) strategy and a stack to manage node exploration. The algorithm begins at a designated starting node and fully explores each branch while checking for the goal state. It then backtracks to explore alternative paths until the goal is found or all reachable nodes are exhausted. DFS works well when the solution is located deeper in the maze, but it may be less effective if the goal is near the starting node. It can also fail to terminate in mazes with infinite depth or loops unless loop detection is added.

In my implementation, the DFS algorithm operates on the maze grid generated by the Maze class, where walls are represented by 1s and free cells by 0s in a 2D NumPy array. Figure 2 illustrates the traversal of an agent from the start node (1, 1) to a target node in a sample maze, showing the path taken. The pseudo-code for my DFS maze-solving algorithm is provided in appendix. The following 2 figures shows the path traced by my DFS implementation for maze 25x25 maze cells ie 51x51 maze and 50x50 maze_cells (101x101 maze). It is important to note that for my results I see that DFS performed the best in terms of compute time when I compare it with other search and mdp algorithms.

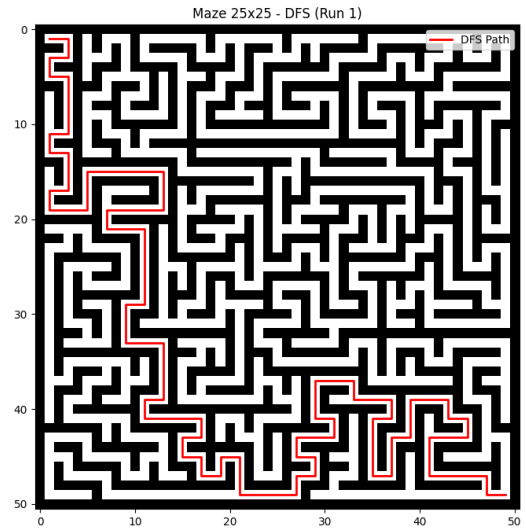


Fig (1)

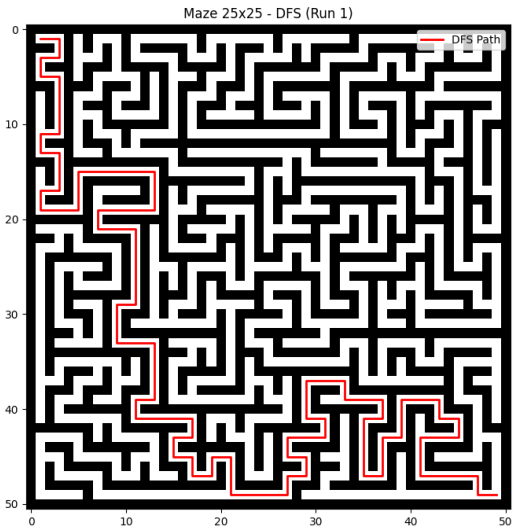


Fig (2)

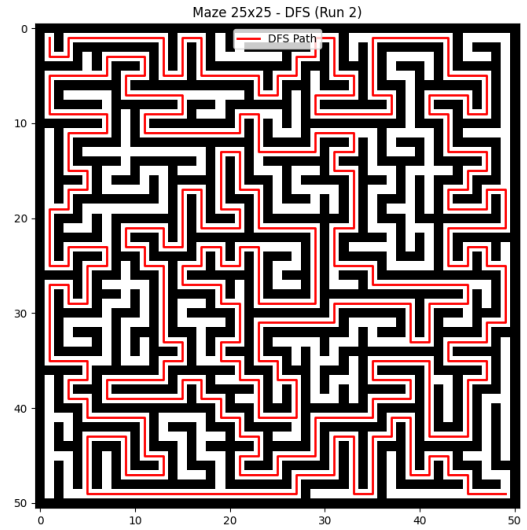


Fig (3)

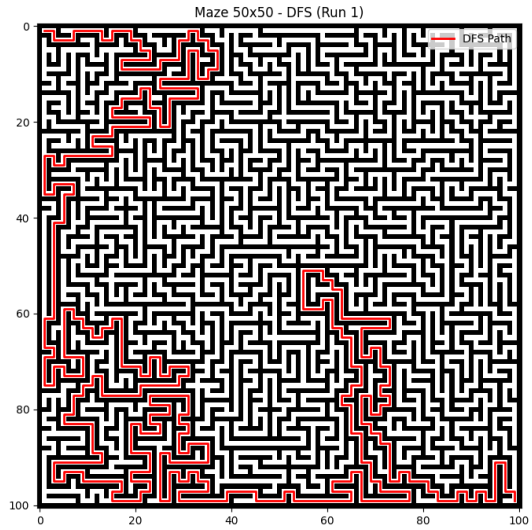


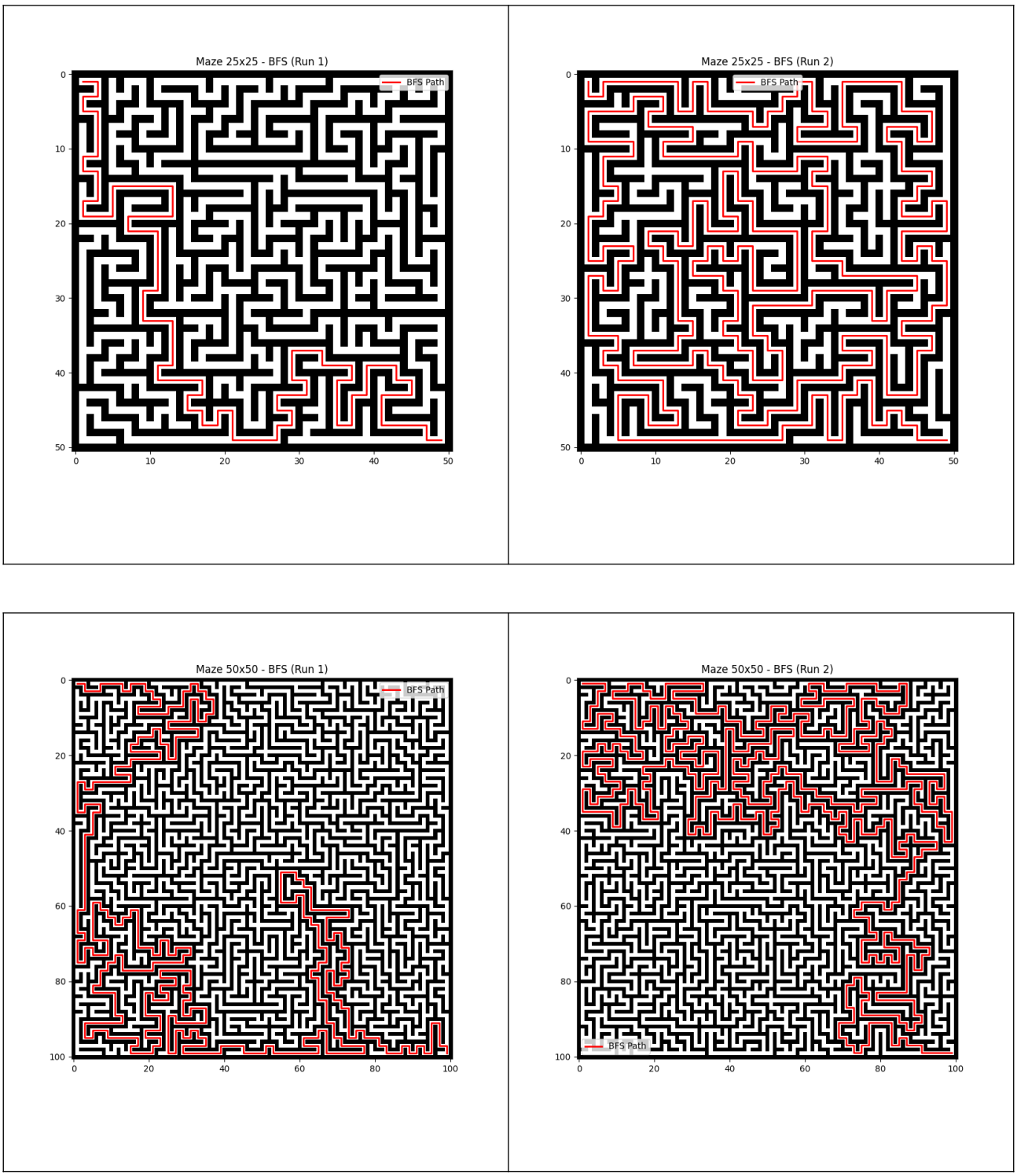
Fig (4)

BREADTH FIRST SEARCH (BFS)

Breadth-First Search (BFS) is an uninformed algorithm that explores the shallowest nodes first. It uses a First-In-First-Out (FIFO) queue, which means it examines all nodes at a given depth before moving deeper. This ensures that BFS is complete and finds the optimal solution in finite mazes with uniform step costs (Ref: Lecture Notes). However, BFS can use a lot of memory and may be less effective when the target is deep in the maze.

My implementation of BFS uses the Maze class's NumPy grid, where walls are represented by 1s and free cells by 0s. Figure 1 and 2 shows an agent moving from the starting node (1, 1) to a target in a sample maze, and it finds the shortest path. The bfs function uses a deque for FIFO operations, a visited set to avoid cycles, and a parent dictionary to reconstruct the shortest path. It also tracks the number of nodes expanded for performance analysis and returns the path and count if the goal is found, or None otherwise.

The following 2 figures shows the path traced by my BFS implementation for maze 25x25 maze cells ie 51x51 maze and 50x50 maze_cells (101x101 maze).



A* SEARCH

A* Search is an informed, weighted graph algorithm combining Uniform Cost Search (UCS) and Greedy Best-First Search. It uses a heuristic function to estimate the distance to the goal, ordering nodes by f(n)=g(n)+h(n), where g(n) is the path cost from the start and h(n) is the heuristic cost to the goal (Ref: Lecture Notes). A* avoids expensive paths and local maxima traps, making it efficient. I chose the Manhattan distance heuristic for its consistency (never overestimating the true distance) and speed, as it limits exploration to top-down and sideways movements, unlike the

diagonal-inclusive Euclidean distance. Figure 4 shows an agent traversing from (1, 1) to a target in a sample maze.

My A star function uses a priority queue (open_set) via heapq, prioritizing nodes by f(n). The manhattan function computes h(n), while g(n) assumes a uniform step cost of 1. It tracks explored nodes with closed_set, updates costs in g_score, and reconstructs the path using came_from. The function returns the path and nodes_expanded if the goal is reached.

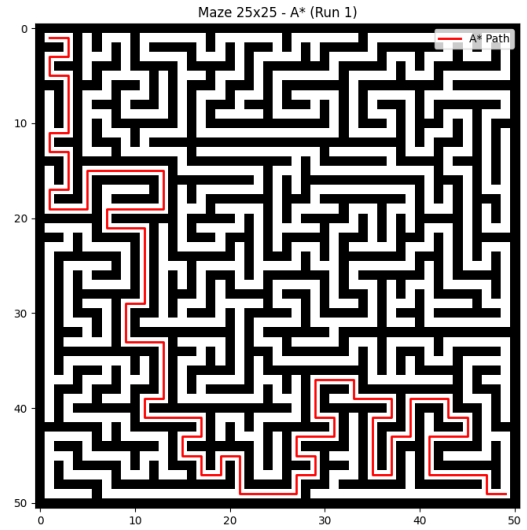


Fig (1)

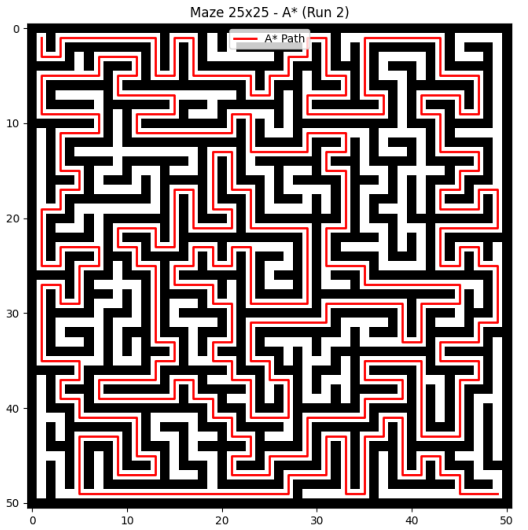


Fig (2)

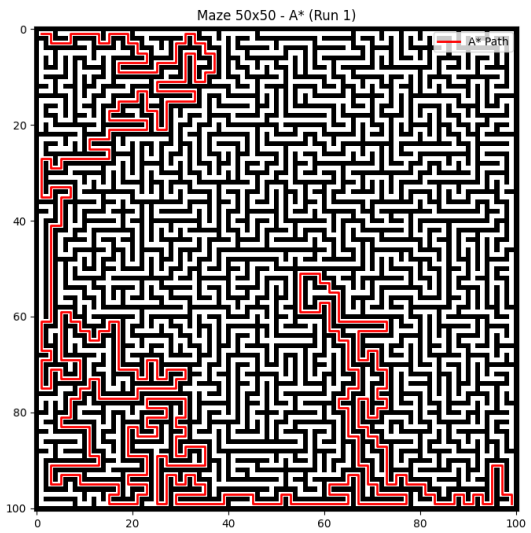


Fig (3)

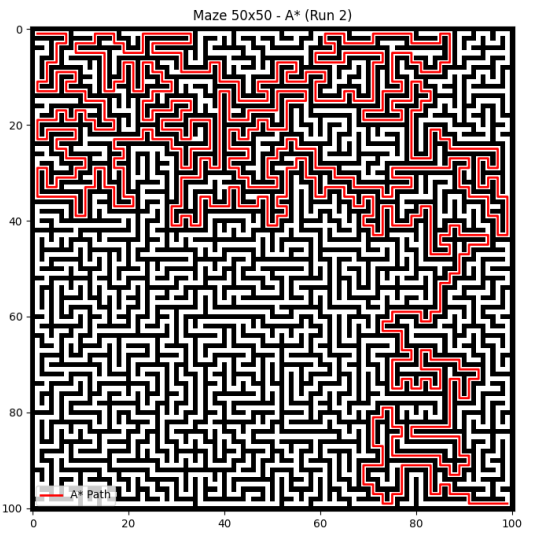


Fig (4)

C) MARKOV DECISION PROCESS ALGORITHMS

A Markov Decision Process (MDP) models decision-making in a random environment where outcomes are uncertain. It has a set of states, such as positions in a maze, and actions you can take in each state, like moving up or down. It also includes a transition model that shows the chance of moving to a new state after an action, and rewards you receive for being in certain states. The goal of an MDP is to make choices that maximize your total reward over time. These choices form a policy, which is a plan that tells you what action to take in each state. The best policy is the one that gives the highest expected payoff, where the payoff is the sum of rewards that are slightly reduced over time.

I have used two methods to solve an MDP and find this optimal policy: Value Iteration and Policy Iteration.

VALUE ITERATION

Value Iteration is a dynamic programming algorithm that finds the best policy for a Markov Decision Process (MDP). It works by repeatedly updating the utility of each state using the Bellman optimality equation:

$$U(S)=R(S)+\gamma \max_{s'} \sum P(s' \mid S,a)U(s')$$

This equation calculates the expected utility of a state S by considering its immediate reward R(S) and the highest possible future reward. The algorithm starts with random utility values and improves them step by step. It stops when the difference in utility values between iterations becomes smaller than a chosen threshold θ, meaning the values have settled. Once the utilities stabilize, the best policy is created by selecting the action that gives the highest expected reward in each state.

Value Iteration is a reliable method because it always finds the best policy, and it works well for problems with discrete state spaces. However, it needs full knowledge of the environment, including all transition probabilities and rewards, which is not always possible. The algorithm can also be slow for large environments like a 50×50 maze, since it has to update every state multiple times.

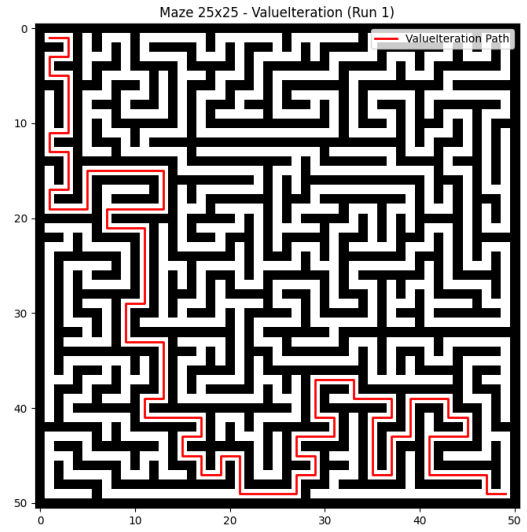


Fig (1)

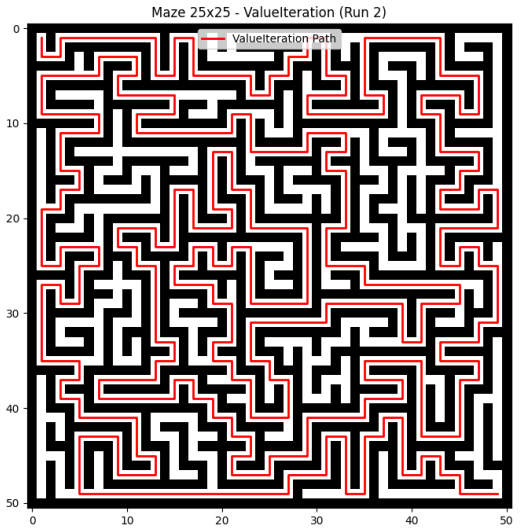


Fig (2)

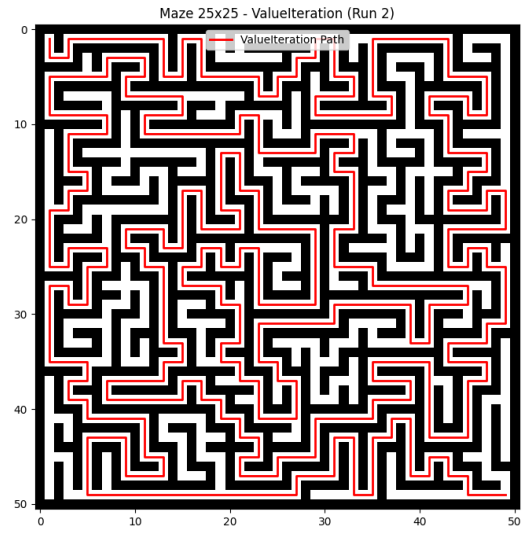


Fig (3)

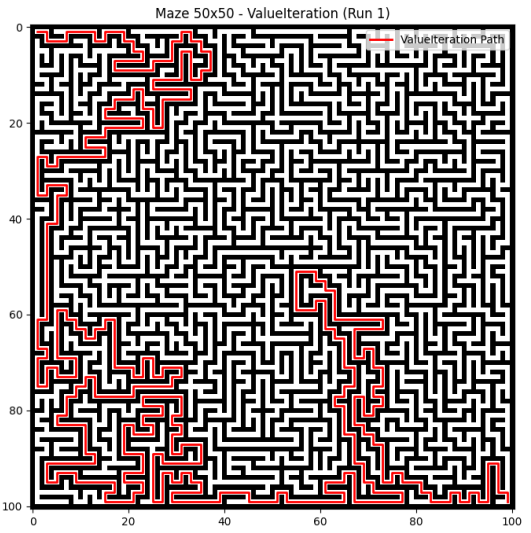


Fig (4)

POLICY ITERATION

Policy Iteration is a method that finds the best policy directly instead of refining utilities step by step. It has two processes: Policy Evaluation, where the algorithm calculates the utility of each state for a given policy, and Policy Improvement, where the policy is updated based on these utilities.

The utility function follows this rule:

$U(S)=R(S)+\gamma\sum_{s'}P(s' \mid S,\pi(S))U(s')$

Once the utility values settle, the policy is updated using:

$\pi(S)=\text{argmax}_a\sum_{s'}P(s' \mid S,a)U(s')$

This cycle repeats until the policy stops changing, meaning it has reached the best possible strategy.

Policy Iteration usually takes fewer steps to converge than Value Iteration, especially when there are only a few possible policies. Whereas, the policy evaluation step can be but slow as it needs to solve a system of equations or run multiple iterations. In my tests, Policy Iteration took longer execution time than Value Iteration in large mazes (50×50) because of repeated evaluations.

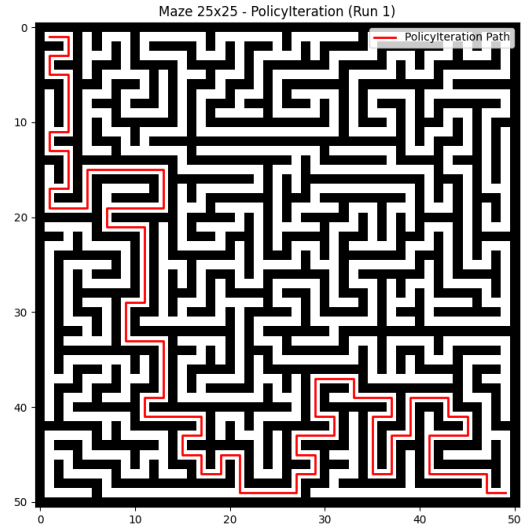


Fig (1)

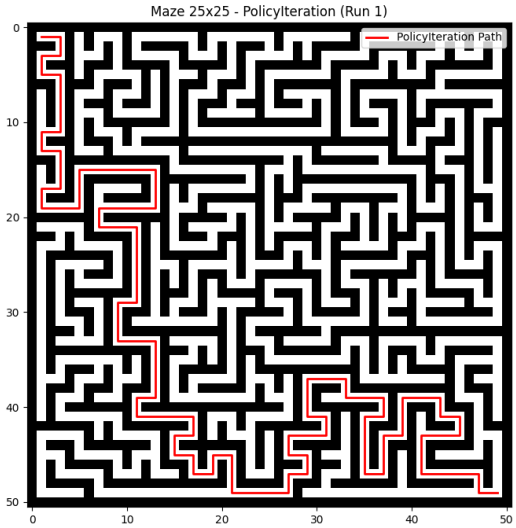


Fig (2)

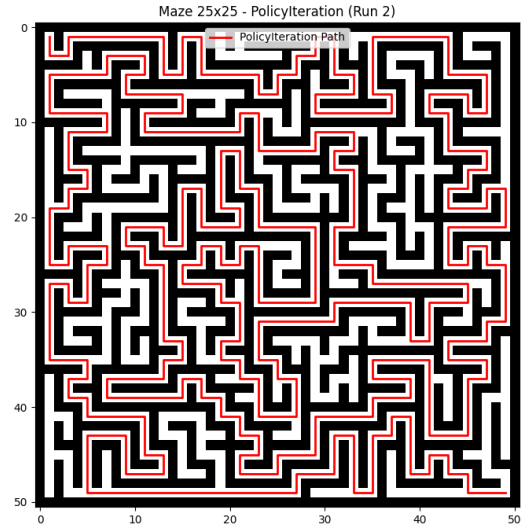


Fig (3)

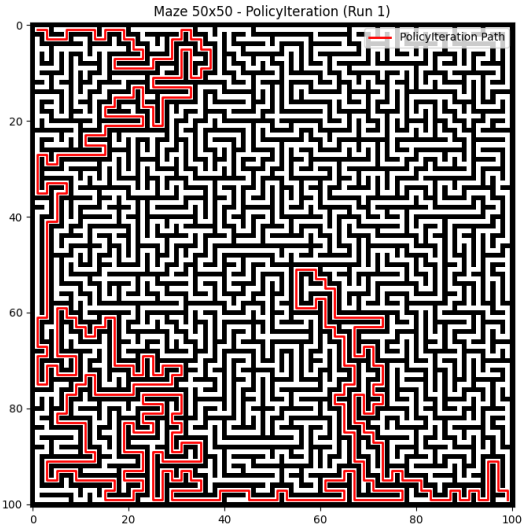


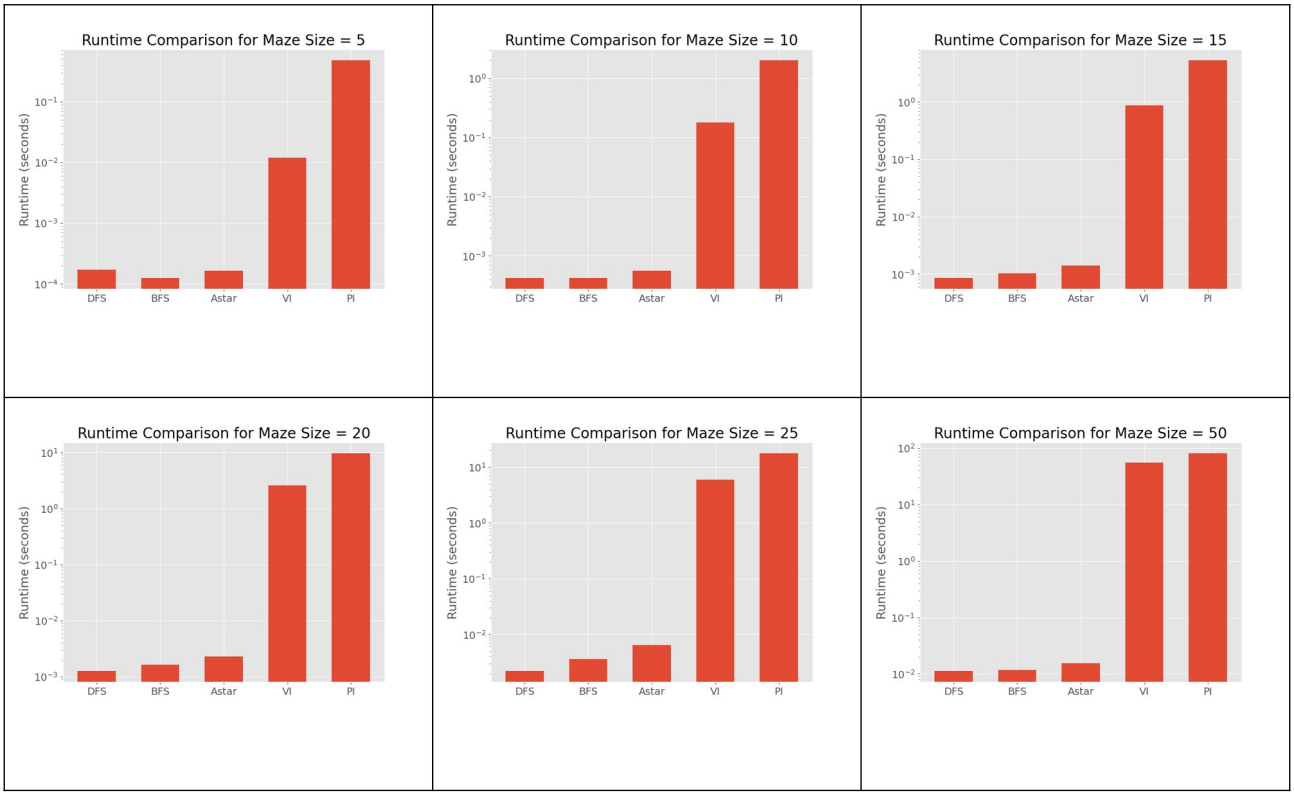
Fig (4)

For both value and policy iterations, I implemented **deterministic transitions** rather than stochastic ones, as stochastic transitions significantly increase computational complexity by requiring probability-weighted sums over multiple possible outcomes. This is because, the uncertainty in stochastic transitions would have introduced variance in results by increasing complexity, making it difficult to fairly compare different algorithms. Keeping transitions deterministic ensured stability and consistency in analyzing convergence behavior.

3) RESULTS AND EVALUATION

The Evaluation is performed on Maze Sizes of 10,15,20,25,50 and each maze size is evaluated for 5 iterations. This would give use good comparisoon for results. In the plots I have taken average of the five iterations for that particular maze size.

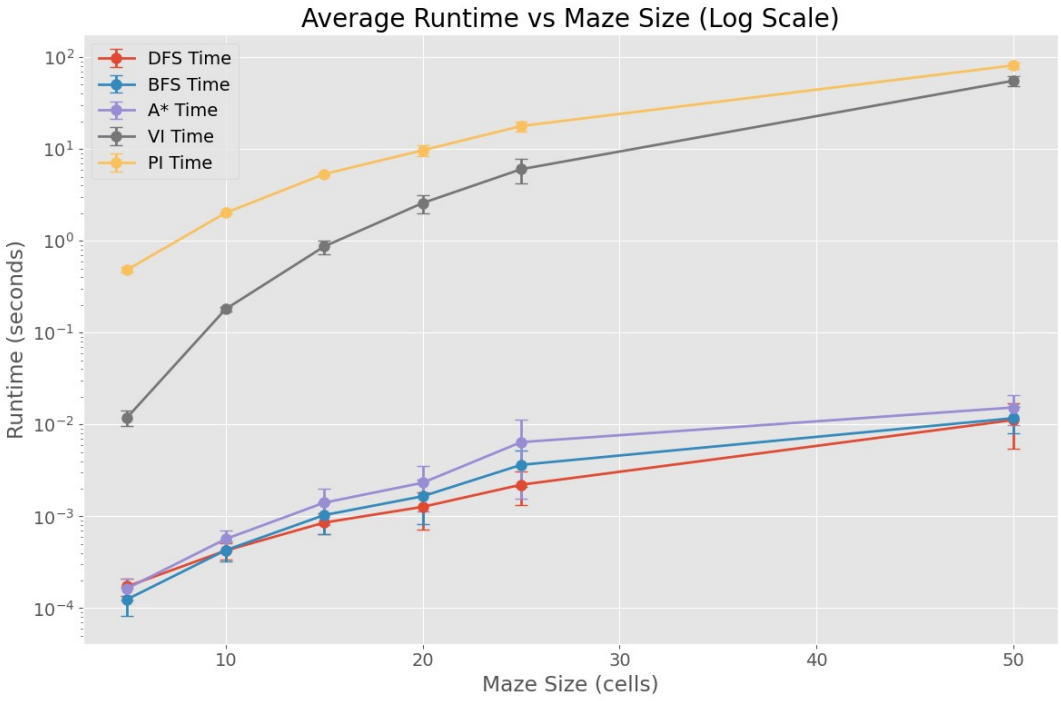
AGGREGATED RUNTIME PLOT



When I tested a 5×5 maze, DFS, BFS, and A* finished almost instantly in under 10^{-2} seconds, while VI and PI took slightly longer but still remained under 10^{-1} seconds. For a 10×10 maze, DFS and BFS stayed very fast, completing in less than 10^{-2} seconds, while A* was a bit slower, taking between 10^{-2} and 10^{-1} seconds. At this size, VI and PI started to show noticeable overhead, nearing

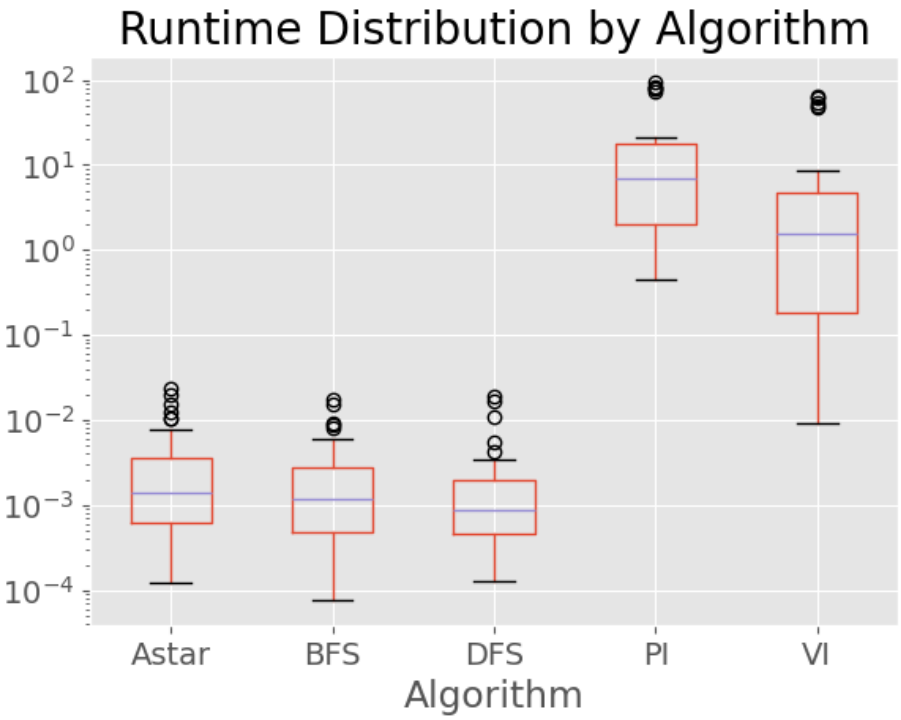
10^{-1} seconds. In a 15×15 maze, the search algorithms remained efficient, all completing in under 1 second, but VI took nearly 1 second, and PI exceeded that slightly. As I increased the maze size to 20×20, DFS, BFS, and A* still completed within 10^{-2} to 10^{-1} seconds, while VI and PI took about 1 second or more. At 25×25, DFS and BFS remained under 10^{-1} seconds, while A* performed similarly, but VI and PI required around 10 seconds. Finally, in a 50×50 maze, DFS, BFS, and A* finished within a few seconds, whereas VI and PI slowed down drastically, taking tens or even over 100 seconds to complete.

AVERAGE RUNTIME



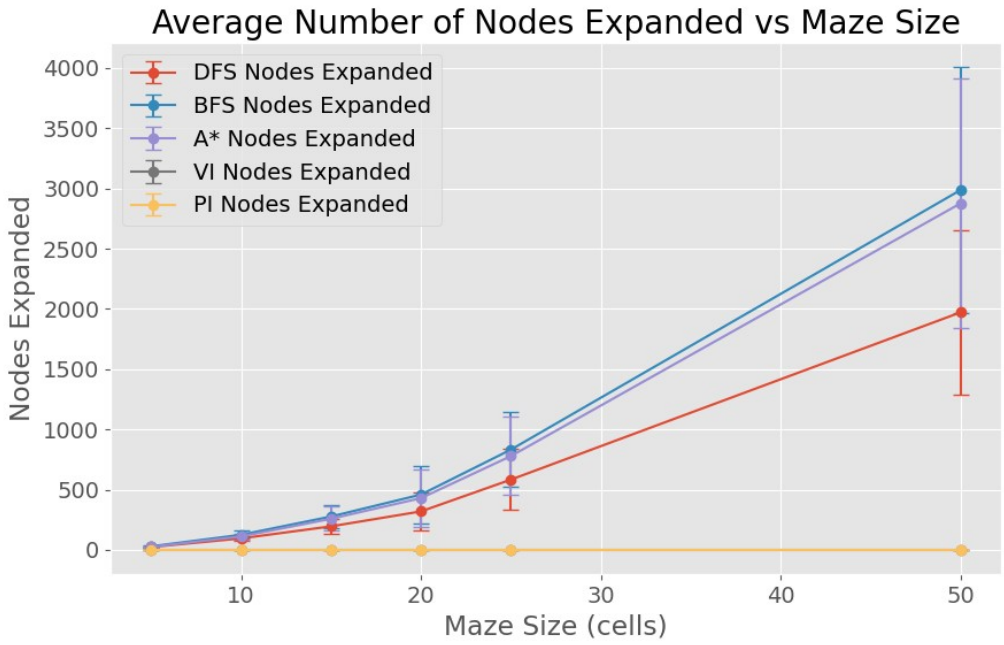
In this line chart, I plotted the average runtime on a log scale (y-axis) against maze size (x-axis, cells per side) for DFS, BFS, A*, Value Iteration (VI), and Policy Iteration (PI). I observed that DFS, BFS, and A* remained under a few seconds even at 50×50, while VI and PI showed a sharp increase in runtime, reaching tens or even hundreds of seconds for larger mazes. This clearly highlights the higher computational overhead of MDP-based methods compared to graph-based searches.

RUNTIME DISTRIBUTION BY ALGORITHM



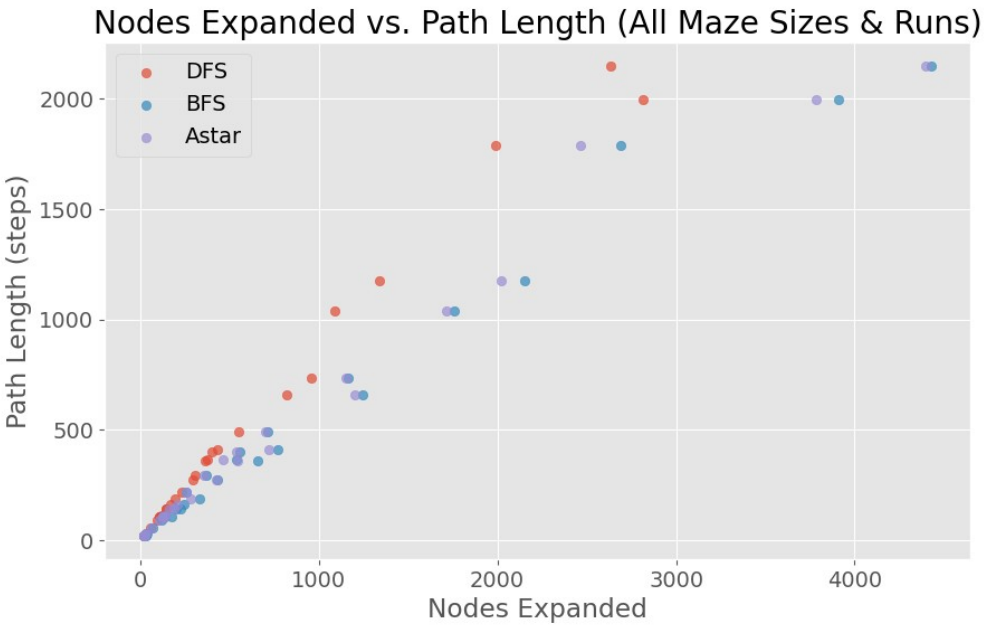
This box plot compares the runtime distributions of each algorithm across multiple runs, with the y-axis in log scale (seconds). DFS, BFS, and A* cluster toward shorter runtimes, though each has outliers reflecting occasional slower runs. VI and PI exhibit higher runtimes overall, as well as broader boxes, indicating greater variability. The box's midline shows the median runtime, while whiskers and dots represent the full spread of values (including outliers).

AVERAGE NODE EXPANSION



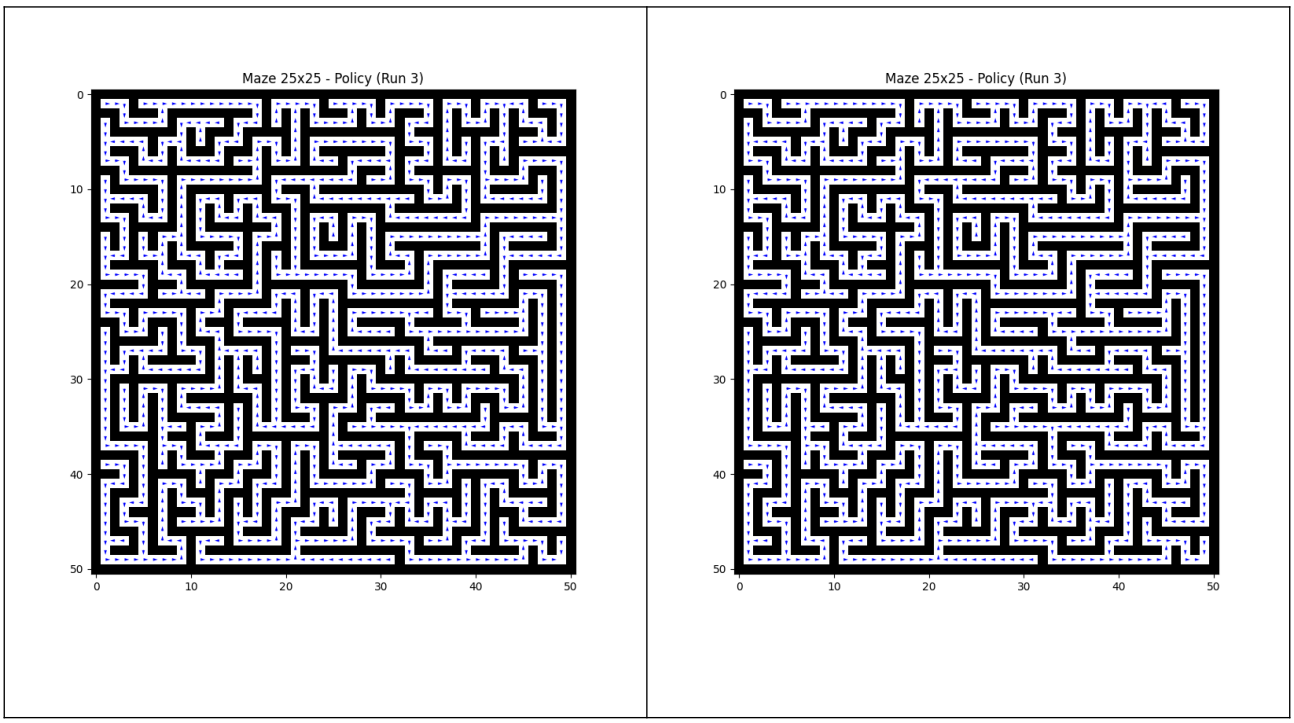
DFS and BFS start with relatively low node expansions at smaller maze sizes, although BFS typically expands more nodes than DFS because it systematically searches all neighbors to ensure the shortest path. A* usually falls between BFS and DFS, as its heuristic reduces unnecessary expansions compared to BFS, though it can still expand more nodes than DFS if the maze is complex. VI and PI expand the most nodes, especially when the maze is 25×25 or larger, because they evaluate or improve policies for the entire state space rather than finding just a single path. Overall, as maze size increases, the difference between graph-based searches (DFS, BFS, A*) and MDP methods (VI, PI) becomes more pronounced, reflecting the heavy overhead of policy-based approaches in large state spaces.

Trade-Off Between Search Effort And Path Quality



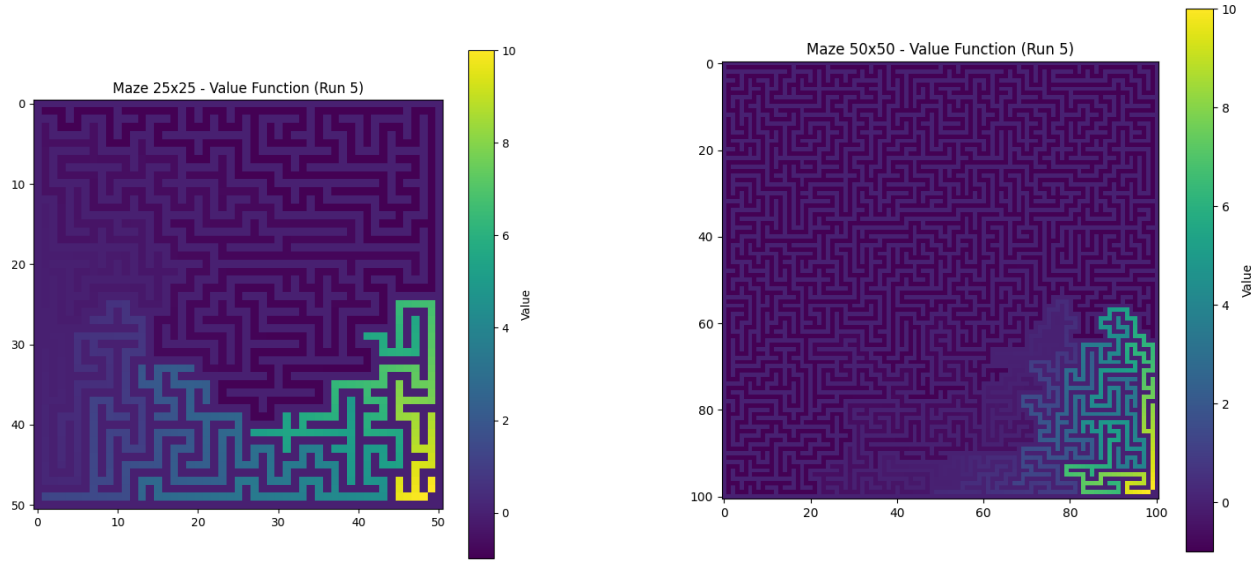
BFS expands many nodes in the maze, which helps it reliably produce the shortest paths, while DFS explores fewer nodes but may result in longer, less direct routes. A* strikes a balance by using a heuristic that reduces unnecessary expansions compared to BFS yet still finds relatively short paths. Overall, there is a clear trade-off: algorithms that explore more nodes, like BFS, achieve optimal paths, whereas those that expand fewer nodes, like DFS, risk suboptimal outcomes, and A* combines both efficiency and path quality.

OPTIMAL POLICY



The above figures show optimal policy that has been found. All the safe states has a direction assigned to it which describes the best action at that particular cell.

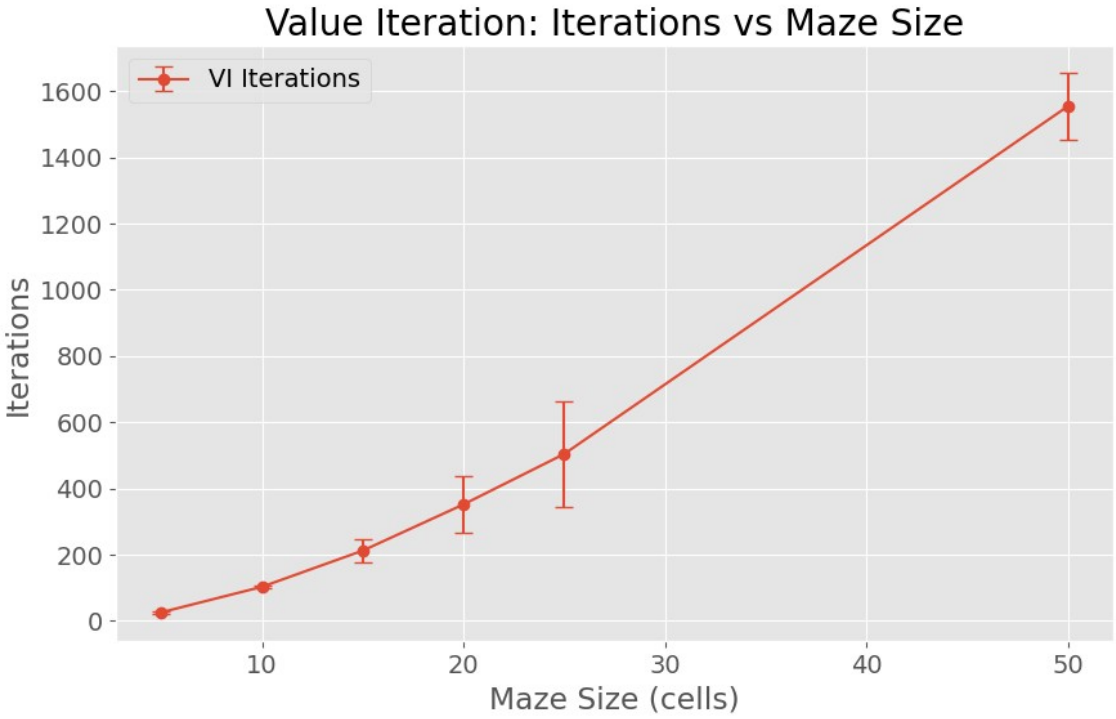
VALUE FUNCTION



This heatmap displays how beneficial each cell is for reaching the goal in a 25×25 maze. Warmer colors (yellow/green) appear in the bottom-right, indicating higher-value states that lie closer to or on an optimal path to the goal. Cooler colors (purple) represent lower-value states, typically farther from the goal. For the 25x25 and 50x50 maze the bottom-right region has warmer hues, reflecting

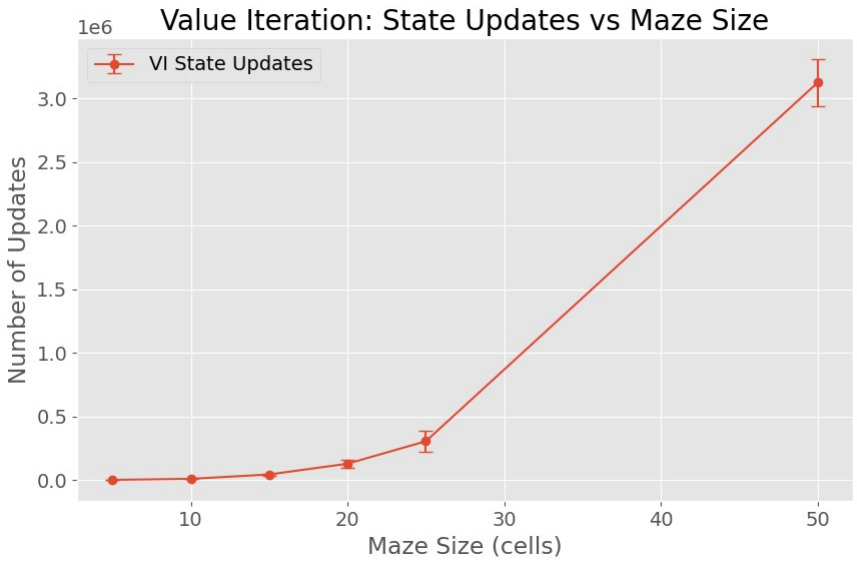
areas near the goal where the expected reward is highest. Most of the maze remains in cooler (purple) shades, signaling lower-value states that are more distant or less direct routes to the goal. The scale from 0 to 10 on the right corresponds to the reward structure

Vi ITERATIONS



This plot shows how many iterations Value Iteration requires to converge for mazes of increasing size. The x-axis is the maze size (in cells per side), and the y-axis is the average number of iterations. As maze size grows (from 5×5 to 50×50), the required iterations rise significantly, reflecting the larger state space and increased computational effort.

Vi STATE UPDATES



This plot shows the number of state updates performed by Value Iteration as the maze size increases from 5×5 to 50×50. Each point represents the mean number of updates across multiple runs. The steep rise at larger maze sizes reflects the growing complexity of the state space, requiring significantly more updates to converge on an optimal value function.

EXPLANATION & DESIGN JUSTIFICATIONS

I experimented with different parameter values and found that setting the discount factor to **γ = 0.99** and the convergence threshold to **θ = 1e-6** produced the most reliable results. A value **γ** = 0.9 worked well for small mazes (up to 20×20), but it failed to converge for larger mazes (25×25 and 50×50) because it undervalued long-term rewards. The high γ makes sure that even distant rewards, like reaching the goal, are considered as the maze size increases, but it also requires more iterations to converge. The tight threshold guarantees a finely tuned value function, yet it increases computation time, especially for very large mazes (such as 100×100) where the state space grows exponentially and convergence becomes impractical.

I also used the Manhattan distance heuristic in A* because it is admissible on a grid and helps reduce unnecessary node expansions while still finding short paths. I compared the algorithms using runtime, path length, nodes expanded, and MDP-specific metrics (iterations and updates) to measure both efficiency and solution quality. Overall, these design choices balance accuracy and performance, ensuring optimal policies in moderately sized mazes, even though MDP methods become computationally expensive and may not converge in extremely large mazes.

4) CONCLUSION

Comparison Among Search Algorithms (DFS, BFS, A*)

DFS and BFS run quickly on small mazes, and A* is slightly slower because it uses a heuristic; however, A* still scales well. BFS always finds the shortest path in an unweighted grid, but DFS sometimes produces longer routes. Additionally, BFS expands more nodes than DFS since it checks every neighbor at each level, while A* usually falls between the two because its heuristic helps reduce unnecessary expansions.

Comparison Among MDP Algorithms (Value Iteration vs. Policy Iteration)

Policy Iteration (PI) tends to converge in fewer iterations than Value Iteration (VI), but each iteration in PI is computationally heavy. VI updates each state directly and may require many iterations, which makes it slower in some cases. Both VI and PI require many state updates in larger mazes, and sometimes PI updates more states because it re-evaluates the entire policy each time. Overall, both methods show a rapid increase in runtime as the maze grows, and for me PI ended up with a higher runtime when compared to VI.

Comparison Between Search and MDP Approaches

DFS, BFS, and A* efficiently finds a single path with lower runtime and fewer node expansions, especially in larger mazes. Whereas, VI and PI becomes much slower because they compute an

optimal policy for every state rather than just one path. BFS guarantees the shortest path, and A* usually finds a path that is as short as BFS if its heuristic is good.

In short if a quick single path is needed, graph-based searches are best. However, if an optimal policy for all states is required, then VI or PI is preferred despite their higher computational cost.

APENDIX

main.py

```
from experiments import run_experiments, plot_aggregated_results
import metrics

def main():
# Define maze sizes and number of runs.
maze_sizes = [5, 10, 15, 20, 25, 50]
num_runs = 5

# Run experiments, returning both aggregated and detailed results
aggregated_results, detailed_results = run_experiments(maze_sizes, num_runs=num_runs)

# Basic aggregated plots (runtime, path length)
plot_aggregated_results(aggregated_results)


# Nodes Expanded vs. Maze Size
metrics.plot_nodes_expanded(aggregated_results)

# Value Iteration Stats (iterations, state updates)
metrics.plot_vi_stats(aggregated_results)

#Path Length vs. Runtime (Scatter)
metrics.plot_path_length_vs_time(detailed_results)

#Runtime Distribution (Box Plot)
metrics.plot_runtime_boxplots(detailed_results)

# Side-by-Side Bar Charts by Maze Size (for runtime)
metrics.plot_bar_charts_by_size(aggregated_results)

#Nodes Expanded vs. Path Length
metrics.plot_nodes_expanded_vs_path_length(detailed_results)

print("Algorithms Evaluated, Check Plots in Results Folder.")

if __name__ == "__main__":
main()
```

experiments.py

```
import os
import csv
import time
import numpy as np
import matplotlib.pyplot as plt

from maze_generator import Maze
from search_algorithms import dfs, bfs, astar
from mdp_algorithms import value_iteration, policy_iteration


RESULTS_DIR = "results"
os.makedirs(RESULTS_DIR, exist_ok=True)
IMAGE_DIR = os.path.join(RESULTS_DIR, "images")
os.makedirs(IMAGE_DIR, exist_ok=True)

def visualize_path(maze_grid, path, algorithm_name, maze_size, run_id):
plt.figure(figsize=(8, 8))
plt.imshow(maze_grid, cmap="binary")
if path:
rows = [p[0] for p in path]
cols = [p[1] for p in path]
plt.plot(cols, rows, color="red", linewidth=2, label=f"{algorithm_name} Path")
plt.title(f"Maze {maze_size}x{maze_size} - {algorithm_name} (Run {run_id})")
plt.legend()
filename = os.path.join(IMAGE_DIR,
f"{algorithm_name.lower()}_maze{maze_size}_run{run_id}.png")
plt.savefig(filename)
plt.close()

def visualize_value(maze_grid, V, maze_size, run_id):
value_array = np.full(maze_grid.shape, np.nan)
for (r, c), val in np.ndenumerate(V):
value_array[r, c] = val
plt.figure(figsize=(8, 8))
cmap = plt.cm.viridis.copy()
cmap.set_bad(color="black")
plt.imshow(value_array, cmap=cmap)
plt.colorbar(label="Value")
plt.title(f"Maze {maze_size}x{maze_size} - Value Function (Run {run_id})")
filename = os.path.join(IMAGE_DIR, f"value_maze{maze_size}_run{run_id}.png")
plt.savefig(filename)
plt.close()
```

```
def visualize_policy(maze_grid, policy, maze_size, run_id):
X, Y, U, V_dir = [], [], [], []
for (r, c), action in np.ndenumerate(policy):
if action == -1 or maze_grid[r, c] != 0:
continue
X.append(c)
Y.append(r)
if action == 0:
U.append(-0.5)
V_dir.append(0)
elif action == 1:
U.append(0.5)
V_dir.append(0)
elif action == 2:
U.append(0)
V_dir.append(-0.5)
elif action == 3:
U.append(0)
V_dir.append(0.5)
if not U:
print("No valid actions found for policy visualization.")
return
plt.figure(figsize=(8, 8))
plt.imshow(maze_grid, cmap="binary")
plt.quiver(X, Y, U, V_dir, color="blue", scale=1, scale_units="xy", angles="xy")
plt.title(f"Maze {maze_size}x{maze_size} - Policy (Run {run_id})")
filename = os.path.join(IMAGE_DIR, f"policy_maze{maze_size}_run{run_id}.png")
plt.savefig(filename)
plt.close()

def visualize_maze(maze_grid, maze_size, run_id):
plt.figure(figsize=(8, 8))
plt.imshow(maze_grid, cmap="binary")
plt.title(f"Maze {maze_size}x{maze_size} (Run {run_id})")
filename = os.path.join(IMAGE_DIR, f"maze_{maze_size}_run{run_id}.png")
plt.savefig(filename)
plt.close()

def run_single_experiment(maze_cells, run_id):
maze_obj = Maze(maze_cells, maze_cells)
maze_grid = maze_obj.generate()

visualize_maze(maze_grid, maze_cells, run_id)

start = (1, 1)
goal = (maze_grid.shape[0] - 2, maze_grid.shape[1] - 2)
metrics = {"maze_cells": maze_cells}

# DFS
t0 = time.time()
```

```
path_dfs, nodes_dfs = dfs(maze_grid, start, goal)
dfs_time = time.time() - t0
metrics["DFS_time"] = dfs_time
metrics["DFS_path_length"] = len(path_dfs) if path_dfs else None
metrics["DFS_nodes_expanded"] = nodes_dfs

# BFS
t0 = time.time()
path_bfs, nodes_bfs = bfs(maze_grid, start, goal)
bfs_time = time.time() - t0
metrics["BFS_time"] = bfs_time
metrics["BFS_path_length"] = len(path_bfs) if path_bfs else None
metrics["BFS_nodes_expanded"] = nodes_bfs

# A*
t0 = time.time()
path_astar, nodes_astar = astar(maze_grid, start, goal)
astar_time = time.time() - t0
metrics["Astar_time"] = astar_time
metrics["Astar_path_length"] = len(path_astar) if path_astar else None
metrics["Astar_nodes_expanded"] = nodes_astar

# MDP Value Iteration
t0 = time.time()
V_vi, policy_vi, iters_vi, state_updates_vi, path_vi = value_iteration(maze_grid, start, goal, 0.99)
vi_time = time.time() - t0
metrics["VI_time"] = vi_time
metrics["VI_path_length"] = len(path_vi) if path_vi else None
metrics["VI_iterations"] = iters_vi
metrics["VI_state_updates"] = state_updates_vi

# MDP Policy Iteration
t0 = time.time()
V_pi, policy_pi, iters_pi, state_updates_pi, path_pi = policy_iteration(maze_grid, start, goal, 0.99)
pi_time = time.time() - t0
metrics["PI_time"] = pi_time
metrics["PI_path_length"] = len(path_pi) if path_pi else None
metrics["PI_iterations"] = iters_pi
metrics["PI_state_updates"] = state_updates_pi

return (metrics, maze_grid, path_dfs, path_bfs, path_astar,
path_vi, path_pi, V_vi, policy_vi, V_pi, policy_pi)

def run_experiments(maze_sizes, num_runs=5):
all_results = []
detailed_results = []

for size in maze_sizes:
metrics_list = []
print(f"Running experiments for maze size: {size}x{size}")
```

```
for run in range(1, num_runs + 1):
(metrics, maze_grid, path_dfs, path_bfs, path_astar,
path_vi, path_pi, V_vi, policy_vi, V_pi, policy_pi) = run_single_experiment(size, run)

metrics["run"] = run
metrics_list.append(metrics)
detailed_results.append(metrics)

visualize_path(maze_grid, path_dfs, "DFS", size, run)
visualize_path(maze_grid, path_bfs, "BFS", size, run)
visualize_path(maze_grid, path_astar, "A*", size, run)
visualize_path(maze_grid, path_vi, "ValueIteration", size, run)
visualize_path(maze_grid, path_pi, "PolicyIteration", size, run)
visualize_value(maze_grid, V_vi, size, run)
visualize_policy(maze_grid, policy_pi, size, run)

# Aggregate metrics for this maze size.
agg = {"maze_cells": size}
keys = [k for k in metrics_list[0].keys() if k not in ["maze_cells", "run"]]
for key in keys:
values = [m[key] for m in metrics_list if m[key] is not None]
if values:
agg[f"{key}_mean"] = np.mean(values)
agg[f"{key}_std"] = np.std(values)
else:
agg[f"{key}_mean"] = None
agg[f"{key}_std"] = None
all_results.append(agg)

# Save detailed results to CSV inside the results folder.
csv_file = os.path.join(RESULTS_DIR, "maze_experiment_data_detailed.csv")
with open(csv_file, "w", newline="") as f:
writer = csv.DictWriter(f, fieldnames=detailed_results[0].keys())
writer.writeheader()
for row in detailed_results:
writer.writerow(row)
print(f"Detailed experiment data saved to {csv_file}")

return all_results, detailed_results

def plot_aggregated_results(aggregated_results):
plt.style.use('ggplot')
plt.rcParams.update({'font.size': 14})

sizes = [res["maze_cells"] for res in aggregated_results]

algo_mapping = {
"DFS": "DFS",
"BFS": "BFS",
"A*": "Astar",
```

```
"VI": "VI",
"PI": "PI"
}

fig, ax = plt.subplots(figsize=(12, 8))
for display_name, key_name in algo_mapping.items():
times = [res.get(f"{key_name}_time_mean", 0) for res in aggregated_results]
stds = [res.get(f"{key_name}_time_std", 0) for res in aggregated_results]
ax.errorbar(
sizes, times, yerr=stds,
marker='o', markersize=8,
linewidth=2, capsize=5,
label=f"{display_name} Time"
)

ax.set_xlabel("Maze Size (cells)")
ax.set_ylabel("Runtime (seconds)")
ax.set_title("Average Runtime vs Maze Size (Log Scale)")
ax.set_yscale('log')
ax.legend()
ax.grid(True)
plt.tight_layout()
filename = os.path.join(IMAGE_DIR, "aggregated_runtime.png")
plt.savefig(filename)
plt.close()


if __name__ == "__main__":
maze_sizes = [10, 15, 20, 25, 50]
num_runs = 5
aggregated_results, detailed_results = run_experiments(maze_sizes, num_runs=num_runs)
plot_aggregated_results(aggregated_results)
csv_agg_file = os.path.join(RESULTS_DIR, "maze_experiment_data_aggregated.csv")
with open(csv_agg_file, "w", newline="") as f:
fieldnames = aggregated_results[0].keys()
writer = csv.DictWriter(f, fieldnames=fieldnames)
writer.writeheader()
for row in aggregated_results:
writer.writerow(row)
print(f"Aggregated experiment data saved to {csv_agg_file}")
```

maze_genereate.py

```
import numpy as np
import random

class Maze:
```

```
def __init__(self, cells_w, cells_h):
self.cells_w = cells_w
self.cells_h = cells_h
self.grid_w = 2 * cells_w + 1
self.grid_h = 2 * cells_h + 1
self.grid = np.ones((self.grid_h, self.grid_w), dtype=int)

def generate(self):
visited = [[False for _ in range(self.cells_w)] for _ in range(self.cells_h)]
stack = [(0, 0)]
visited[0][0] = True
self.grid[1, 1] = 0

while stack:
x, y = stack[-1]
neighbors = []
for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
nx, ny = x + dx, y + dy
if 0 <= nx < self.cells_w and 0 <= ny < self.cells_h and not visited[ny][nx]:
neighbors.append((nx, ny, dx, dy))
if neighbors:
nx, ny, dx, dy = random.choice(neighbors)
visited[ny][nx] = True
# Remove wall between current cell and neighbor:
grid_x, grid_y = 2 * x + 1, 2 * y + 1
self.grid[grid_y + dy, grid_x + dx] = 0
# Mark neighbor cell
self.grid[2 * ny + 1, 2 * nx + 1] = 0
stack.append((nx, ny))
else:
stack.pop()
return self.grid

if __name__ == "__main__":
# Simple test of maze generation.
maze = Maze(5, 5)
grid = maze.generate()

# Save grid to CSV file
np.savetxt('maze_test.csv', grid, delimiter=',', fmt='%d')
print(grid)
```

mdp_algorithms.py

```
import numpy as np
import pandas as pd
```

```
def print_policy(policy, shape):
actions = ['←', '→', '↑', '↓']
policy_grid = np.array([[actions[a] if a != -1 else '#' for a in row] for row in policy])
print("\nPolicy:")
for row in policy_grid:
print(" ".join(row))


def reconstruct_path(policy, start, goal):
actions = [(0, -1), (0, 1), (-1, 0), (1, 0)] # Left, Right, Up, Down
path = []
current = start
visited = set()
while current != goal and current not in visited:
path.append(current)
visited.add(current)
action = policy[current[0], current[1]]
if action == -1:
break
dr, dc = actions[action]
current = (current[0] + dr, current[1] + dc)
if current == goal:
path.append(goal)
return path


def value_iteration(grid, start, goal, gamma, theta=1e-6, max_iterations=10000):
rows, cols = grid.shape
values = np.zeros((rows, cols), dtype=np.float64)
policy = np.full((rows, cols), -1, dtype=np.int8)
actions = [(0, -1), (0, 1), (-1, 0), (1, 0)] # Left, Right, Up, Down
state_updates = 0

for iteration in range(max_iterations):
delta = 0
new_values = values.copy()

for r in range(rows):
for c in range(cols):
if grid[r, c] == 1 or (r, c) == goal:
continue

best_value = float('-inf')
best_action = -1

for a, (dr, dc) in enumerate(actions):
nr, nc = r + dr, c + dc
if 0 <= nr < rows and 0 <= nc < cols and grid[nr, nc] != 1:
```

```
next_state = (nr, nc)
if next_state == goal:
reward = 10 # Reaching the goal
else:
reward = -0.01 # Moving to a free space
else:
next_state = (r, c) # Stay put if hitting wall or out of bounds
reward = -10 # Hitting a wall or out of bounds

v = reward + gamma * values[next_state[0]][next_state[1]]
if v > best_value:
best_value = v
best_action = a

if best_action != -1: # Only update if a valid action exists
if abs(new_values[r, c] - best_value) > theta:
state_updates += 1
new_values[r, c] = best_value
policy[r, c] = best_action
delta = max(delta, abs(new_values[r, c] - values[r, c]))

values = new_values
if delta < theta:
break
else:
print(f"Value Iteration stopped after max iterations ({max_iterations})")

path = reconstruct_path(policy, start, goal)
return values, policy, iteration + 1, state_updates, path


def policy_iteration(grid, start, goal, gamma, theta=1e-6, max_policy_iterations=1000,
max_evaluation_iterations=10000):
rows, cols = grid.shape
values = np.zeros((rows, cols), dtype=np.float64)
policy = np.full((rows, cols), -1, dtype=np.int8)
actions = [(0, -1), (0, 1), (-1, 0), (1, 0)] # Left, Right, Up, Down
state_updates = 0 # Initialize state updates

# Initialize policy for open states to a default action (action 0: Left)
for r in range(rows):
for c in range(cols):
if grid[r, c] == 0 and (r, c) != goal:
policy[r, c] = 0

policy_iteration_count = 0
while policy_iteration_count < max_policy_iterations:
# **Policy Evaluation**
evaluation_iteration = 0
while evaluation_iteration < max_evaluation_iterations:
```

```
delta = 0
for r in range(rows):
for c in range(cols):
if grid[r, c] == 1 or (r, c) == goal: # Skip walls and goal
continue
action = policy[r, c]
dr, dc = actions[action]
nr, nc = r + dr, c + dc
if 0 <= nr < rows and 0 <= nc < cols and grid[nr, nc] != 1:
next_state = (nr, nc)
if next_state == goal:
reward = 10 # Reaching the goal (adjusted to match value iteration)
else:
reward = -0.01 # Moving to a free space
else:
next_state = (r, c) # Stay put if hitting wall or out of bounds
reward = -10 # Hitting a wall or out of bounds
new_v = reward + gamma * values[next_state[0]][next_state[1]]
if abs(new_v - values[r, c]) > theta: # If state value changes significantly
state_updates += 1
delta = max(delta, abs(new_v - values[r, c]))
values[r, c] = new_v
evaluation_iteration += 1
if delta < theta:
break
if evaluation_iteration == max_evaluation_iterations:
print(f"Policy evaluation did not converge after {max_evaluation_iterations} iterations")

# **Policy Improvement**
policy_stable = True
for r in range(rows):
for c in range(cols):
if grid[r, c] == 1 or (r, c) == goal: # Skip walls and goal
continue
old_action = policy[r, c]
best_action = None
best_value = float('-inf')
for a, (dr, dc) in enumerate(actions):
nr, nc = r + dr, c + dc
if 0 <= nr < rows and 0 <= nc < cols and grid[nr, nc] != 1:
next_state = (nr, nc)
if next_state == goal:
reward = 10 # Reaching the goal (adjusted to match value iteration)
else:
reward = -0.01 # Moving to a free space
else:
next_state = (r, c) # Stay put if hitting wall or out of bounds
reward = -10 # Hitting a wall or out of bounds
v = reward + gamma * values[next_state[0]][next_state[1]]
if v > best_value:
```

```
best_value = v
best_action = a
if best_action is not None and best_action != old_action:
policy[r, c] = best_action
policy_stable = False

policy_iteration_count += 1
if policy_stable:
break
else:
print(f"Policy Iteration did not converge after {max_policy_iterations} iterations")

path = reconstruct_path(policy, start, goal)
return values, policy, policy_iteration_count, state_updates, path


def load_grid_from_csv(filename):
"""Load a maze grid from a CSV file."""
return np.array(pd.read_csv(filename, header=None), dtype=np.int8)


def main():
filename = input("Enter CSV filename: ")
grid = load_grid_from_csv(filename)
print(f"Grid shape: {grid.shape}")

start_input = input("Enter start state as 'row,col' (default: 1,1): ")
goal_input = input(f"Enter goal state as 'row,col' (default: {grid.shape[0]-2},{grid.shape[1]-2}): ")

if start_input:
start_row, start_col = map(int, start_input.split(','))
start_state = (start_row, start_col)
else:
start_state = (1, 1)

if goal_input:
goal_row, goal_col = map(int, goal_input.split(','))
goal_state = (goal_row, goal_col)
else:
goal_state = (grid.shape[0]-2, grid.shape[1]-2)

# Get gamma value
gamma_input = input("Enter gamma value (default: 0.9): ")
gamma = float(gamma_input) if gamma_input else 0.99

# Run algorithms
V_vi, policy_vi, iters_vi, state_updates_vi, path_vi = value_iteration(grid, start_state, goal_state,
gamma)
V_pi, policy_pi, iters_pi, state_updates_pi, path_pi = policy_iteration(grid, start_state, goal_state,
gamma)
```

```
# Print results
print("\n=== VALUE ITERATION ===")
print_policy(policy_vi, grid.shape)
print(f"Iterations: {iters_vi}")
print(f"State updates: {state_updates_vi}")
print(f"Path: {path_vi}")

print("\n=== POLICY ITERATION ===")
print_policy(policy_pi, grid.shape)
print(f"Iterations: {iters_pi}")
print(f"State updates: {state_updates_pi}")
print(f"Path: {path_pi}")

if __name__ == "__main__":
main()
```

search_algorithms.py

```
from collections import deque
import heapq

def get_neighbors(maze_grid, pos):
directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
neighbors = []
r, c = pos
rows, cols = maze_grid.shape
for dr, dc in directions:
nr, nc = r + dr, c + dc
if 0 <= nr < rows and 0 <= nc < cols and maze_grid[nr, nc] == 0:
neighbors.append((nr, nc))
return neighbors

def dfs(maze_grid, start, goal):
stack = [start]
visited = set()
parent = {}
nodes_expanded = 0

while stack:
current = stack.pop()
nodes_expanded += 1
if current == goal:
break
if current in visited:
continue
visited.add(current)
for neighbor in get_neighbors(maze_grid, current):
```

```
if neighbor not in visited:
stack.append(neighbor)
if neighbor not in parent:
parent[neighbor] = current

if current != goal:
return None, nodes_expanded

path = []
node = goal
while node != start:
path.append(node)
node = parent[node]
path.append(start)
path.reverse()
return path, nodes_expanded

def bfs(maze_grid, start, goal):
queue = deque([start])
visited = {start}
parent = {}
nodes_expanded = 0

while queue:
current = queue.popleft()
nodes_expanded += 1
if current == goal:
break
for neighbor in get_neighbors(maze_grid, current):
if neighbor not in visited:
visited.add(neighbor)
parent[neighbor] = current
queue.append(neighbor)

if current != goal:
return None, nodes_expanded

path = []
node = goal
while node != start:
path.append(node)
node = parent[node]
path.append(start)
path.reverse()
return path, nodes_expanded

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

def astar(maze_grid, start, goal):
```

```
import heapq

open_set = []
# The heap will store tuples of (f_score, g_score, node)
heapq.heappush(open_set, (manhattan(start, goal), 0, start))
came_from = {}
g_score = {start: 0}
nodes_expanded = 0
closed_set = set()

while open_set:
f, current_g, current = heapq.heappop(open_set)
nodes_expanded += 1

if current == goal:
path = []
node = current
while node in came_from:
path.append(node)
node = came_from[node]
path.append(start)
path.reverse()
return path, nodes_expanded

if current in closed_set:
continue
closed_set.add(current)

for neighbor in get_neighbors(maze_grid, current):
tentative_g = current_g + 1
if tentative_g < g_score.get(neighbor, float('inf')):
g_score[neighbor] = tentative_g
priority = tentative_g + manhattan(neighbor, goal)
heapq.heappush(open_set, (priority, tentative_g, neighbor))
came_from[neighbor] = current

return None, nodes_expanded
```

metrics.py

```
""import os
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

# Use the same RESULTS_DIR and IMAGE_DIR as in experiments.py.
RESULTS_DIR = "results"
```

```
IMAGE_DIR = os.path.join(RESULTS_DIR, "images")
os.makedirs(IMAGE_DIR, exist_ok=True)

def plot_nodes_expanded(aggregated_results):
algo_mapping = {
"DFS": "DFS",
"BFS": "BFS",
"A*": "Astar",
"VI": "VI",
"PI": "PI"
}
sizes = [res["maze_cells"] for res in aggregated_results]

plt.figure(figsize=(10, 6))
for display_name, key_name in algo_mapping.items():
means = [res.get(f"{key_name}_nodes_expanded_mean", 0) for res in aggregated_results]
stds = [res.get(f"{key_name}_nodes_expanded_std", 0) for res in aggregated_results]
plt.errorbar(sizes, means, yerr=stds, marker='o', capsize=5,
label=f"{display_name} Nodes Expanded")

plt.xlabel("Maze Size (cells)")
plt.ylabel("Nodes Expanded")
plt.title("Average Number of Nodes Expanded vs Maze Size")
plt.legend()
plt.grid(True)
filename = os.path.join(IMAGE_DIR, "aggregated_nodes_expanded.png")
plt.savefig(filename)
plt.close()

def plot_vi_stats(aggregated_results):
sizes = [res["maze_cells"] for res in aggregated_results]

# VI Iterations
vi_iters_mean = [res.get("VI_iterations_mean", 0) for res in aggregated_results]
vi_iters_std = [res.get("VI_iterations_std", 0) for res in aggregated_results]

plt.figure(figsize=(10,6))
plt.errorbar(sizes, vi_iters_mean, yerr=vi_iters_std, marker='o', capsize=5,
label="VI Iterations")
plt.xlabel("Maze Size (cells)")
plt.ylabel("Iterations")
plt.title("Value Iteration: Iterations vs Maze Size")
plt.legend()
plt.grid(True)
filename = os.path.join(IMAGE_DIR, "vi_iterations.png")
plt.savefig(filename)
plt.close()

# VI State Updates
vi_updates_mean = [res.get("VI_state_updates_mean", 0) for res in aggregated_results]
```

```
vi_updates_std = [res.get("VI_state_updates_std", 0) for res in aggregated_results]

plt.figure(figsize=(10,6))
plt.errorbar(sizes, vi_updates_mean, yerr=vi_updates_std, marker='o', capsize=5,
label="VI State Updates")
plt.xlabel("Maze Size (cells)")
plt.ylabel("Number of Updates")
plt.title("Value Iteration: State Updates vs Maze Size")
plt.legend()
plt.grid(True)
filename = os.path.join(IMAGE_DIR, "vi_state_updates.png")
plt.savefig(filename)
plt.close()

def plot_pi_stats(aggregated_results):
sizes = [res["maze_cells"] for res in aggregated_results]

# PI Iterations
pi_iters_mean = [res.get("PI_iterations_mean", 0) for res in aggregated_results]
pi_iters_std = [res.get("PI_iterations_std", 0) for res in aggregated_results]

plt.figure(figsize=(10,6))
plt.errorbar(sizes, pi_iters_mean, yerr=pi_iters_std, marker='o', capsize=5,
label="PI Iterations")
plt.xlabel("Maze Size (cells)")
plt.ylabel("Iterations")
plt.title("Policy Iteration: Iterations vs Maze Size")
plt.legend()
plt.grid(True)
filename = os.path.join(IMAGE_DIR, "pi_iterations.png")
plt.savefig(filename)
plt.close()

# PI State Updates
pi_updates_mean = [res.get("PI_state_updates_mean", 0) for res in aggregated_results]
pi_updates_std = [res.get("PI_state_updates_std", 0) for res in aggregated_results]

plt.figure(figsize=(10,6))
plt.errorbar(sizes, pi_updates_mean, yerr=pi_updates_std, marker='o', capsize=5,
label="PI State Updates")
plt.xlabel("Maze Size (cells)")
plt.ylabel("State Updates")
plt.title("Policy Iteration: State Updates vs Maze Size")
plt.legend()
plt.grid(True)
filename = os.path.join(IMAGE_DIR, "pi_state_updates.png")
plt.savefig(filename)
plt.close()

def plot_path_length_vs_time(detailed_results):
```

```
algo_keys = ["DFS", "BFS", "Astar", "VI", "PI"]

plt.figure(figsize=(10, 6))
for algo in algo_keys:
path_lengths = []
times = []
for row in detailed_results:
length_key = f"{algo}_path_length"
time_key = f"{algo}_time"
if row.get(length_key) is not None and row.get(time_key) is not None:
path_lengths.append(row[length_key])
times.append(row[time_key])
if path_lengths and times:
plt.scatter(path_lengths, times, alpha=0.7, label=algo)
plt.xlabel("Path Length (steps)")
plt.ylabel("Runtime (seconds)")
plt.title("Path Length vs. Runtime (All Maze Sizes & Runs)")
plt.legend()
plt.grid(True)
filename = os.path.join(IMAGE_DIR, "path_length_vs_runtime.png")
plt.savefig(filename)
plt.close()

def plot_runtime_boxplots(detailed_results):
algo_keys = ["DFS", "BFS", "Astar", "VI", "PI"]
time_data = []
for row in detailed_results:
for algo in algo_keys:
time_key = f"{algo}_time"
if row.get(time_key) is not None:
time_data.append({
"algorithm": algo,
"time": row[time_key]
})
if not time_data:
print("No time data found for boxplots. Skipping.")
return
df_time = pd.DataFrame(time_data)
plt.figure(figsize=(10, 6))
df_time.boxplot(column="time", by="algorithm", grid=True)
plt.yscale('log')
plt.title("Runtime Distribution by Algorithm")
plt.suptitle("")
plt.xlabel("Algorithm")
plt.ylabel("Runtime (seconds)")
filename = os.path.join(IMAGE_DIR, "runtime_boxplot.png")
plt.savefig(filename)
plt.close()

def plot_bar_charts_by_size(aggregated_results):
```

```
algo_keys = ["DFS", "BFS", "Astar", "VI", "PI"]
for res in aggregated_results:
size = res["maze_cells"]
means = [res.get(f"{algo}_time_mean", 0) for algo in algo_keys]
plt.figure(figsize=(8,6))
x_positions = np.arange(len(algo_keys))
plt.bar(x_positions, means, width=0.6, align='center')
plt.xticks(x_positions, algo_keys)
plt.yscale('log')
plt.ylabel("Runtime (seconds)")
plt.title(f"Runtime Comparison for Maze Size = {size}")
plt.grid(True, axis='y')
filename = os.path.join(IMAGE_DIR, f"bar_runtime_size_{size}.png")
plt.savefig(filename)
plt.close()

def plot_nodes_expanded_vs_path_length(detailed_results):
algo_keys = ["DFS", "BFS", "Astar", "VI", "PI"]
plt.figure(figsize=(10, 6))
for algo in algo_keys:
expanded_key = f"{algo}_nodes_expanded"
length_key = f"{algo}_path_length"
xs = []
ys = []
for row in detailed_results:
if row.get(expanded_key) is not None and row.get(length_key) is not None:
xs.append(row[expanded_key])
ys.append(row[length_key])
if xs and ys:
plt.scatter(xs, ys, alpha=0.7, label=algo)
plt.xlabel("Nodes Expanded")
plt.ylabel("Path Length (steps)")
plt.title("Nodes Expanded vs. Path Length (All Maze Sizes & Runs)")
plt.legend()
plt.grid(True)
filename = os.path.join(IMAGE_DIR, "nodes_expanded_vs_path_length.png")
plt.savefig(filename)
plt.close()


plt.xlabel("Maze Size (cells)")
plt.ylabel("Iterations")
plt.title("Policy Iteration: Iterations vs Maze Size")
plt.legend()
plt.grid(True)
filename = os.path.join(IMAGE_DIR, "pi_iterations.png")
plt.savefig(filename)
plt.close()

# PI State Updates
```

```
pi_updates_mean = [res.get("PI_state_updates_mean", 0) for res in aggregated_results]
pi_updates_std = [res.get("PI_state_updates_std", 0) for res in aggregated_results]

plt.figure(figsize=(10,6))
plt.errorbar(sizes, pi_updates_mean, yerr=pi_updates_std, marker='o', capsize=5,
label="PI State Updates")
plt.xlabel("Maze Size (cells)")
plt.ylabel("State Updates")
plt.title("Policy Iteration: State Updates vs Maze Size")
plt.legend()
plt.grid(True)
filename = os.path.join(IMAGE_DIR, "pi_state_updates.png")
plt.savefig(filename)
plt.close()

def plot_path_length_vs_time(detailed_results):
algo_keys = ["DFS", "BFS", "Astar", "VI", "PI"]

plt.figure(figsize=(10, 6))
for algo in algo_keys:
path_lengths = []
times = []
for row in detailed_results:
length_key = f"{algo}_path_length"
time_key = f"{algo}_time"
if row.get(length_key) is not None and row.get(time_key) is not None:
path_lengths.append(row[length_key])
times.append(row[time_key])
if path_lengths and times:
plt.scatter(path_lengths, times, alpha=0.7, label=algo)
plt.xlabel("Path Length (steps)")
plt.ylabel("Runtime (seconds)")
plt.title("Path Length vs. Runtime (All Maze Sizes & Runs)")
plt.legend()
plt.grid(True)
filename = os.path.join(IMAGE_DIR, "path_length_vs_runtime.png")
plt.savefig(filename)
plt.close()

def plot_runtime_boxplots(detailed_results):
algo_keys = ["DFS", "BFS", "Astar", "VI", "PI"]
time_data = []
for row in detailed_results:
for algo in algo_keys:
time_key = f"{algo}_time"
if row.get(time_key) is not None:
time_data.append({
"algorithm": algo,
"time": row[time_key]
})
```

```
if not time_data:
print("No time data found for boxplots. Skipping.")
return
df_time = pd.DataFrame(time_data)
plt.figure(figsize=(10, 6))
df_time.boxplot(column="time", by="algorithm", grid=True)
plt.yscale('log')
plt.title("Runtime Distribution by Algorithm")
plt.suptitle("")
plt.xlabel("Algorithm")
plt.ylabel("Runtime (seconds)")
filename = os.path.join(IMAGE_DIR, "runtime_boxplot.png")
plt.savefig(filename)
plt.close()

def plot_bar_charts_by_size(aggregated_results):
algo_keys = ["DFS", "BFS", "Astar", "VI", "PI"]
for res in aggregated_results:
size = res["maze_cells"]
means = [res.get(f"{algo}_time_mean", 0) for algo in algo_keys]
plt.figure(figsize=(8,6))
x_positions = np.arange(len(algo_keys))
plt.bar(x_positions, means, width=0.6, align='center')
plt.xticks(x_positions, algo_keys)
plt.yscale('log')
plt.ylabel("Runtime (seconds)")
plt.title(f"Runtime Comparison for Maze Size = {size}")
plt.grid(True, axis='y')
filename = os.path.join(IMAGE_DIR, f"bar_runtime_size_{size}.png")
plt.savefig(filename)
plt.close()

def plot_nodes_expanded_vs_path_length(detailed_results):
algo_keys = ["DFS", "BFS", "Astar", "VI", "PI"]
plt.figure(figsize=(10, 6))
for algo in algo_keys:
expanded_key = f"{algo}_nodes_expanded"
length_key = f"{algo}_path_length"
xs = []
ys = []
for row in detailed_results:
if row.get(expanded_key) is not None and row.get(length_key) is not None:
xs.append(row[expanded_key])
ys.append(row[length_key])
if xs and ys:
plt.scatter(xs, ys, alpha=0.7, label=algo)
plt.xlabel("Nodes Expanded")
plt.ylabel("Path Length (steps)")
plt.title("Nodes Expanded vs. Path Length (All Maze Sizes & Runs)")
plt.legend()
```

```
plt.grid(True)
filename = os.path.join(IMAGE_DIR, "nodes_expanded_vs_path_length.png")
plt.savefig(filename)
plt.close()
```