MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

Artificial Intelligence (CSE 2225) MINI PROJECT REPORT ON

AI-Based Maze Solver for Complex Labyrinth Exploration

SUBMITTED TO

Department of Computer Science & Engineering

by

ANISH SETYA , Reg No :-220962005 , Roll No :-04 , Semester :- 4
JANAV BHASIN , Reg No :-220962049 , Roll No :-15,Semester:- 4
YASH ROHAN , Reg No :-220962072 , Roll No :-21 , Semester :- 4
AKSHAY SAXENA , Reg No :-220962009 , Roll No :-07 , Semester :- 4

Name & Signature of Evaluator

(Jan 2024 - May 2024)

Table of Contents

CHAPTER-1: INTRODUCTION

1.1 INTRODUCTION

Exploring tricky mazes and labyrinths can be really hard. But now, thanks to smart computer programs such as AI-based Maze Solver, things are getting easier. These programs use fancy math and learning tricks to find the best way through mazes. In this report, we`ll learn about how these AI systems work and how they can help in lots of different areas. These smart Maze Solver programs are like super-smart puzzle solvers. They use special tricks to figure out the best path, making maze exploration a breeze.

1.2 PROBLEM STATEMENT

The project aims to develop an AI-based solution for navigating through a complex maze, incorporating various challenges and obstacles. The maze consists of multiple pathways, dead ends, traps, and distractions, making it crucial to devise an effective strategy to explore and navigate through the labyrinth efficiently**.**

1.3 OBJECTIVE

- Develop an AI-powered maze-solving algorithm capable of efficiently navigating through the maze to reach the exit.

- Create a user-friendly interface allowing users to input maze configurations, initiate the solving process, and visualize the progress of the AI agent.

- Implement graphical representations of the maze environment, including the positions of the AI agent, obstacles, and the exit, to enhance user engagement and comprehension.

CHAPTER-2: LITERATURE REVIEW

2.1 UCS-Uniform Cost Search

Uniform Cost Search (UCS) is a graph traversal algorithm used in maze navigation. It systematically explores all possible paths from a starting point to a goal, prioritizing paths with lower cumulative costs. UCS selects the path with the minimum cost encountered thus far, making it effective for finding the optimal solution in scenarios with varying edge traversal costs.

2.2 A* Algorithm

A* is a search algorithm like UCS, but it's more efficient. It looks at both the cost of getting to a point and an estimate of how much further it needs to go. This estimate helps it focus on the most promising paths. A* is like planning a road trip, considering both the distance to your destination and how fast you can travel. It picks the path that seems shortest overall.

2.3 BFS-Best First Search

Breadth-First Search (BFS) is a simple search algorithm that explores a graph level by level. It starts from the root node and visits all the neighboring nodes first, before moving on to the next level. BFS is like exploring a maze by checking each room adjacent to your current location before moving forward. It ensures that closer nodes are visited before deeper ones, making it effective for finding the shortest path in unweighted graphs**.**

2.4 DFS-Depth First Search

Depth-First Search (DFS) is another search algorithm, but it explores as far as possible along each branch before backtracking. It starts from the root node and goes as deep as it can, then backtracks when it reaches a dead end. DFS is like exploring a maze by taking one path until you can't go any further, then going back to try another. It's often used to search through large, complex graphs or mazes.

2.5 Best First Search

Best-First Search (BFS) is an algorithm that explores paths based on an evaluation function. It selects the most promising node to expand next, based on a heuristic that estimates the cost to reach the goal from that node. BFS is like navigating a maze by always choosing the path that seems closest to the destination. It's useful for efficiently finding solutions in large search spaces, prioritizing exploration towards potentially optimal paths.

2.6 Reinforcement Learning

The MazeSolver model serves as a learning platform for RL agents seeking to master maze navigation. Its reward mechanism acts as a guiding light, steering agents towards favorable actions while deterring unfavorable ones. When agents encounter obstacles, they receive negative rewards, nudging them to navigate cautiously. Likewise, revisiting previous locations incurs penalties, encouraging exploration of uncharted territory. However, upon reaching the target, agents receive positive reinforcement, reinforcing successful navigation strategies. This blend of rewards shapes the agent's decision-making, fostering adaptive behaviors conducive to efficient maze traversal.

At the heart of this model lies a dynamic interplay between actions, rewards, and observations. Agents leverage their current state to select actions, subsequently receiving feedback through rewards and observations. With each interaction, agents learn from this feedback, refining their strategies to maximize cumulative rewards. Through iterative exploration of various paths and associated rewards, agents incrementally enhance their decision-making prowess, honing in on optimal maze-solving strategies. This iterative learning cycle, driven by the nuanced interplay of actions and rewards, constitutes the essence of the RL process within the MazeSolver model, empowering agents to navigate mazes with increasing proficiency through experiential learning and adaptive decision-making.

CHAPTER-3: METHODOLOGY

3.1 UCS, A*, Best first Search, Depth First Search, Breadth First Search

```
import numpy as np
import cv2

import time
from queue import PriorityQueue

class xyz:
    def _init_(self, maze):
        self.maze = maze
        self.height, self.width = maze.shape

    def a_star(self, start, end):
        visited = np.zeros_like(self.maze, dtype=bool)
        pq = PriorityQueue()
        pq.put((0, start))
        visited[start] = True
        parent = {}

        while not pq.empty():
            _, current = pq.get()

            if current == end:
                return True, parent

            directions = [(1, 0), (0, 1), (-1, 0), (0, -1)]  # down, right, up, left
            for dr, dc in directions:
                new_row, new_col = current[0] + dr, current[1] + dc
                if 0 <= new_row < self.height and 0 <= new_col < self.width and
self.maze[new_row][new_col] == 1 and not visited[new_row][new_col]:
                    visited[new_row][new_col] = True
                    pq.put((self.heuristic((new_row, new_col), end) + 1, (new_row, new_col)))
                    parent[(new_row, new_col)] = current

        return False, None

    def bfs(self, start, end):
        visited = np.zeros_like(self.maze, dtype=bool)
        queue = []
        queue.append(start)
        visited[start] = True
        parent = {}
        while queue:
            current = queue.pop(0)

            if current == end:
                return True, parent

            directions = [(1, 0), (0, 1), (-1, 0), (0, -1)]  # down, right, up, left
            for dr, dc in directions:
                new_row, new_col = current[0] + dr, current[1] + dc
                if 0 <= new_row < self.height and 0 <= new_col < self.width and
self.maze[new_row][new_col] == 1 and not visited[new_row][new_col]:
                    visited[new_row][new_col] = True
                    queue.append((new_row, new_col))
                    parent[(new_row, new_col)] = current

        return False, None

    def render(self, mode='human', delay=0.1):
        canvas = np.zeros((self.height, self.width, 3), dtype=np.uint8)
        canvas[:, :, 2] = self.maze * 255
        canvas = cv2.resize(canvas, (120, 120))
        cv2.namedWindow('Maze', cv2.WINDOW_NORMAL)
        cv2.resizeWindow('Maze', 1600, 600)
        cv2.imshow('Maze', canvas)
        cv2.waitKey(100)
        for _ in range(int(10 / delay)):
            time.sleep(delay)

    def ucs(self, start, end):
        visited = np.zeros_like(self.maze, dtype=bool)
        pq = PriorityQueue()
        pq.put((0, start))
        visited[start] = True
        parent = {}

        while not pq.empty():
            cost, current = pq.get()

            if current == end:
                return True, parent

            directions = [(1, 0), (0, 1), (-1, 0), (0, -1)]  # down, right, up, left
            for dr, dc in directions:
                new_row, new_col = current[0] + dr, current[1] + dc
                if 0 <= new_row < self.height and 0 <= new_col < self.width and
self.maze[new_row][new_col] == 1 and not visited[new_row][new_col]:
                    visited[new_row][new_col] = True
                    pq.put((cost + 1, (new_row, new_col)))
                    parent[(new_row, new_col)] = current

        return False, None

    def best_first_search(self, start, end):
        visited = np.zeros_like(self.maze, dtype=bool)
        pq = PriorityQueue()
        pq.put((0, start))
        visited[start] = True
        parent = {}

        while not pq.empty():
            _, current = pq.get()

            if current == end:
                return True, parent

            directions = [(1, 0), (0, 1), (-1, 0), (0, -1)]  # down, right, up, left
            for dr, dc in directions:
                new_row, new_col = current[0] + dr, current[1] + dc
                if 0 <= new_row < self.height and 0 <= new_col < self.width and
self.maze[new_row][new_col] == 1 and not visited[new_row][new_col]:
                    visited[new_row][new_col] = True
                    pq.put((self.heuristic((new_row, new_col), end), (new_row, new_col)))
                    parent[(new_row, new_col)] = current

        return False, None

    def dfs(self, start, end):
        visited = np.zeros_like(self.maze, dtype=bool)
        stack = [start]
        parent = {}

        while stack:
            current = stack.pop()

            if current == end:
                return True, parent

            visited[current] = True

            directions = [(1, 0), (0, 1), (-1, 0), (0, -1)]  # down, right, up, left
            for dr, dc in directions:
                new_row, new_col = current[0] + dr, current[1] + dc
                if 0 <= new_row < self.height and 0 <= new_col < self.width and
self.maze[new_row][new_col] == 1 and not visited[(new_row, new_col)]:
```

```
                    stack.append((new_row, new_col))
                    parent[(new_row, new_col)] = current

        return False, None

    def heuristic(self, node, end):
        # Manhattan distance heuristic
        return abs(node[0] - end[0]) + abs(node[1] - end[1])

    def mark_path(self, start, end, parent):
        row, col = end
        while (row, col) != start:
            self.maze[row][col] = 2
            row, col = parent[(row, col)]

    def find_path_ucs(self):
        start = (0, 0)
        end = (self.height - 1, self.width - 1)
        found, parent = self.ucs(start, end)
        if found:
            self.mark_path(start, end, parent)
            print("Path found!")
        else:
            print("No path found.")

    def find_path_best_first_search(self):
        start = (0, 0)
        end = (self.height - 1, self.width - 1)
        found, parent = self.best_first_search(start, end)
        if found:
            self.mark_path(start, end, parent)
            print("Path found using Best First Search!")
        else:
            print("No path found using Best First Search.")

    def find_path_dfs(self):
        start = (0, 0)
        end = (self.height - 1, self.width - 1)
        found, parent = self.dfs(start, end)
        if found:
            self.mark_path(start, end, parent)
            print("Path found using Depth First Search!")
        else:
            print("No path found using Depth First Search.")

    def find_path_a_star(self):
        start = (0, 0)
        end = (self.height - 1, self.width - 1)
        found, parent = self.a_star(start, end)
        if found:
            self.mark_path(start, end, parent)
            print("Path found using A* Algorithm!")
        else:
            print("No path found using A* Algorithm.")

    def find_path_bfs(self):
        start = (0, 0)
        end = (self.height - 1, self.width - 1)
        found, parent = self.bfs(start, end)
        if found:
            self.mark_path(start, end, parent)
            print("Path found using Breadth-First Search (BFS)!")
        else:
            print("No path found using Breadth-First Search (BFS).")

    def mark_path(self, start, end, parent):
        row, col = end
        while (row, col) != start:
            self.maze[row][col] = 2
            row, col = parent[(row, col)]

maze = np.array([
    [1., 0., 1., 1., 1., 1., 1., 1., 1., 1.],
    [1., 1., 1., 1., 1., 0., 1., 1., 1., 1.],
    [1., 1., 1., 1., 1., 0., 1., 1., 1., 1.],
    [0., 0., 1., 0., 0., 1., 0., 1., 1., 1.],
    [1., 1., 0., 1., 0., 1., 0., 0., 0., 1.],
    [1., 1., 0., 1., 0., 1., 1., 1., 1., 1.],
    [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
    [1., 1., 1., 1., 1., 1., 0., 0., 0., 0.],
    [1., 0., 0., 0., 0., 0., 1., 1., 1., 1.],
    [1., 1., 1., 1., 1., 1., 1., 0., 1., 1.]
], dtype=np.int32)
maze_solver = xyz(maze)
maze_solver.find_path_ucs()
print(maze)
maze = np.array([
    [1., 0., 1., 1., 1., 1., 1., 1., 1., 1.],
    [1., 1., 1., 1., 1., 0., 1., 1., 1., 1.],
    [1., 1., 1., 1., 1., 0., 1., 1., 1., 1.],
    [0., 0., 1., 0., 0., 1., 0., 1., 1., 1.],
    [1., 1., 0., 1., 0., 1., 0., 0., 0., 1.],
    [1., 1., 0., 1., 0., 1., 1., 1., 1., 1.],
    [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
    [1., 1., 1., 1., 1., 1., 0., 0., 0., 0.],
    [1., 0., 0., 0., 0., 0., 1., 1., 1., 1.],
    [1., 1., 1., 1., 1., 1., 1., 0., 1., 1.]
], dtype=np.int32)
maze_solver = xyz(maze)
maze_solver.find_path_best_first_search()
print(maze)
maze = np.array([
    [1., 0., 1., 1., 1., 1., 1., 1., 1., 1.],
    [1., 1., 1., 1., 1., 0., 1., 1., 1., 1.],
    [1., 1., 1., 1., 1., 0., 1., 1., 1., 1.],
    [0., 0., 1., 0., 0., 1., 0., 1., 1., 1.],
    [1., 1., 0., 1., 0., 1., 0., 0., 0., 1.],
    [1., 1., 0., 1., 0., 1., 1., 1., 1., 1.],
    [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
    [1., 1., 1., 1., 1., 1., 0., 0., 0., 0.],
    [1., 0., 0., 0., 0., 0., 1., 1., 1., 1.],
    [1., 1., 1., 1., 1., 1., 1., 0., 1., 1.]
], dtype=np.int32)
maze_solver = xyz(maze)
maze_solver.find_path_dfs()
print(maze)
maze = np.array([
    [1., 0., 1., 1., 1., 1., 1., 1., 1., 1.],
    [1., 1., 1., 1., 1., 0., 1., 1., 1., 1.],
    [1., 1., 1., 1., 1., 0., 1., 1., 1., 1.],
    [0., 0., 1., 0., 0., 1., 0., 1., 1., 1.],
    [1., 1., 0., 1., 0., 1., 0., 0., 0., 1.],
    [1., 1., 0., 1., 0., 1., 1., 1., 1., 1.],
    [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
    [1., 1., 1., 1., 1., 1., 0., 0., 0., 0.],
    [1., 0., 0., 0., 0., 0., 1., 1., 1., 1.],
    [1., 1., 1., 1., 1., 1., 1., 0., 1., 1.]
], dtype=np.int32)
maze_solver = xyz(maze)
maze_solver.find_path_a_star()
print(maze)
maze = np.array([
    [1., 0., 1., 1., 1., 1., 1., 1., 1., 1.],
    [1., 1., 1., 1., 1., 0., 1., 1., 1., 1.],
    [1., 1., 1., 1., 1., 0., 1., 1., 1., 1.],
    [0., 0., 1., 0., 0., 1., 0., 1., 1., 1.],
    [1., 1., 0., 1., 0., 1., 0., 0., 0., 1.],
    [1., 1., 0., 1., 0., 1., 1., 1., 1., 1.],
    [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
    [1., 1., 1., 1., 1., 1., 0., 0., 0., 0.],
    [1., 0., 0., 0., 0., 0., 1., 1., 1., 1.],
    [1., 1., 1., 1., 1., 1., 1., 0., 1., 1.]
], dtype=np.int32)
maze_solver = xyz(maze)
maze_solver.find_path_bfs()
print(maze)
```

This code defines a maze-solving algorithm using three different search methods: Uniform Cost Search (UCS), Best First Search, and Depth First Search (DFS). Let's break down the methodology step by step:

<u>Initialization:</u>

The xyz class is defined to represent the maze and implement various maze-solving algorithms.

The maze is represented as a 2D NumPy array where 1 represents a wall or obstacle, and 0 represents an empty space.

The init method initializes the maze object with the provided maze.

<u>Rendering</u>:

The render method visualizes the maze using OpenCV.

It creates a window named 'Maze' and displays the maze grid where walls are shown in red and empty spaces are shown in black.

The render method also allows for adjusting the rendering delay to control the speed of visualization.

<u>Uniform Cost Search (UCS):</u>

The ucs method implements the Uniform Cost Search algorithm to find the shortest path from the start to the end in the maze.

It maintains a priority queue (PriorityQueue) to explore nodes based on their cost.

The cost of reaching each node is calculated as the sum of the costs of all the steps taken to reach that node.

The algorithm explores neighboring nodes in all four directions (up, down, left, right) and continues until the goal node is reached or all nodes are explored.

<u>A* Search</u>

This method works like UCS but it also takes a heuristic into account. A heuristic is a fuction that helps to make a guided decision.The heuristic in this case is the manhattan distance of the current node from the goal node.

The cost of reaching each node is calculated as the sum of the costs of all the steps taken to reach that node added to the heuristic of the node. All the unvisited neighbours of the node are enqueued in a priority queue and the neighbour with the least cost is dequed. Due to this nature of the algorithm, we might see some jumps between nodes as the node may have a lesser total cost.

Best First Search:

The best_first_search method implements the Best First Search algorithm.

Similar to UCS, it explores neighboring nodes based on their heuristic value, which estimates the distance from the current node to the goal.

In this implementation, the Manhattan distance heuristic is used, which is the sum of the absolute differences in the x and y coordinates between the current node and the goal.

Depth First Search (DFS):

The dfs method implements the Depth First Search algorithm.

It explores nodes by following a single path as far as possible before backtracking.

This algorithm uses a stack to keep track of nodes to be explored.

Path Marking:

The mark_path method marks the path from the start to the end found by the search algorithms.

It modifies the maze array by setting the cells along the path to a different value (e.g., 2).

Finding Paths:

The find_path_ucs, find_path_best_first_search, and find_path_dfs methods are used to find paths in the maze using UCS, Best First Search, and DFS, respectively.

These methods call the respective search algorithm and mark the path if found.

Testing:

The code tests each search algorithm on three different maze configurations (maze, maze1, and maze2) by creating instances of the xyz class with each maze and calling the appropriate methods.

This methodology provides a comprehensive approach to solving mazes using different search algorithms and visualizing the results. Each algorithm has its advantages and limitations, making them suitable for different types of mazes and search requirements.

3.2 Reinforcement Learning

```
import numpy as np
import cv2
import gymnasium

class MazeSolver(gymnasium.Env):
    def _init_(self, maze, target, rat=(0, 0)):
        self.og_rat = rat
        self.path = []
        self.rat = rat
        self.target = target
        self.tr = 0
        self.info = {}
        self.action_space = gymnasium.spaces.Discrete(4)
        self.maze = maze
        self.height, self.width = maze.shape
        self.observation_space = gymnasium.spaces.Box(low=0, high=1, shape=(self.height,
self.width, 2), dtype=np.int16)

    def reset(self, **kwargs):
        self.rat = self.og_rat
        self.path = []
        self.tr = 0
        return (self._get_observation(),self.info)

    def step(self, action):
        prev = self.rat
        done = False
        reward = 0

        if action == 0:
            self.rat = (self.rat[0] - 1, self.rat[1])
        elif action == 1:
            self.rat = (self.rat[0], self.rat[1] + 1)
        elif action == 2:
            self.rat = (self.rat[0] + 1, self.rat[1])
        else:
            self.rat = (self.rat[0], self.rat[1] - 1)

        if self.rat[0] < 0 or self.rat[1] < 0 or self.rat[0] >= self.height or self.rat[1] >= self.width:
            reward += -0.8
            self.rat = prev
            print("Hit a boundry")
        elif self.maze[self.rat[0]][self.rat[1]] == 0:
            reward += -0.8
            print("Hit obstacle")
            self.rat = prev
        elif self.rat in self.path:
            reward += -0.25
        elif self.maze[self.rat[0]][self.rat[1]] == 0:
            reward += -0.04
        elif self.rat == self.target:
            reward += 1.0
            print("target found")
            done = True

        self.tr += reward
        if self.tr < (-0.5 * self.height * self.width):
            print("Overtime")
            done = True

        self.path.append(self.rat)
        self.info = {"TimeLimit.truncated": False}
        return self._get_observation(), reward, done, False, self.info

    def _get_observation(self):
        rat_pos = np.zeros((self.height, self.width), dtype=np.int16)
        rat_pos[self.rat[0], self.rat[1]] = 1
        observation = np.dstack((self.maze, rat_pos))
        return observation.astype(np.int16)
    def render(self, mode='human'):
        canvas = np.zeros((self.height, self.width, 3), dtype=np.uint8)
        canvas[:, :, 2] = self.maze * 255
        canvas[self.rat[0], self.rat[1], :] = [0, 255, 0]
        canvas[self.target[0], self.target[1], :] = [255, 0, 0]
        resized_canvas = cv2.resize(canvas, None, fx=50, fy=50,
interpolation=cv2.INTER_NEAREST)
        cv2.imshow('Maze', resized_canvas)
        cv2.waitKey(100)




model = DQN("MlpPolicy", env, verbose=1, tensorboard_log=log_path)
model.learn(total_timesteps=100000)
```

```
episodes = 3
for episode in range(1, episodes+1):
    obs,_ = env.reset()
    done = False
    score = 0
    while not done:
        action, _states = model.predict(obs)
        obs, reward, done,_, info = env.step(action)
        score+=reward
        env.render()
    print('Episode:{} Score:{}'.format(episode, score))
    cv2.destroyAllWindows()
```

The provided code encompasses the implementation of a reinforcement learning (RL) model for solving maze navigation problems. It consists of two main components: the MazeSolver environment and the RL algorithm, specifically the Deep Q-Network (DQN) model.

Firstly, the MazeSolver environment is defined as a custom class that inherits from the gymnasium.Env class, making it compatible with OpenAI Gym environments. This environment encapsulates the maze layout, agent's position, and target position. It provides methods for resetting the environment (reset), executing actions (step), and rendering the environment for visualization (render). The step method computes rewards based on the agent's actions, updating the agent's position and checking for termination conditions such as hitting boundaries or reaching the target. The reward function is finely tuned: the agent receives a negative reward of -0.8 for hitting maze boundaries or obstacles, discouraging reckless movements. Moreover, revisiting previous positions incurs a penalty of -0.25, encouraging exploration of new paths and preventing the agent from getting stuck in loops. Additionally, the reward for finding the target is set to 1.0, signaling successful maze traversal. The _get_observation method constructs the observation space by combining the maze layout and the agent's position. The render method visualizes the maze and the agent's movements using OpenCV.

Secondly, the RL algorithm is implemented using the DQN algorithm. The DQN model is initialized with the "MlpPolicy", an MLP policy that serves as the function approximator for the DQN algorithm. This MLP policy provides flexibility and scalability to handle high-dimensional input spaces and non-linear relationships between inputs and outputs. The DQN model is instantiated with the DQN class and trained using the learn method with a specified number of total timesteps. During training, the model interacts with the MazeSolver environment, learning optimal navigation strategies through trial and error. After training, the model is evaluated over a specified number of episodes. In each episode, the agent interacts with the environment by selecting actions based on the learned policy and observing rewards. The episode terminates when the agent reaches the target or exceeds a maximum number of steps. Finally, the total score achieved by the agent in each episode is printed to evaluate its performance.

In summary, the provided code implements a RL-based maze navigation system using the DQN algorithm with an MLP policy. It demonstrates how RL agents can learn to navigate through maze environments, guided by a finely-tuned reward-based learning mechanism, and showcases the interaction between the RL model and the custom maze-solving environment.

CHAPTER-4: RESULTS AND DISCUSSIONS

Reinforcement Learning:

```
Given maze is:
[[1 0 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 0 1 1 1 1]
 [1 1 1 1 1 0 1 1 1 1]
 [0 0 1 0 0 1 0 1 1 1]
 [1 1 0 1 0 1 0 0 0 1]
 [1 1 0 1 0 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 0 0 0 0]
 [1 0 0 0 0 0 1 1 1 1]
 [1 1 1 1 1 1 1 0 1 1]]

Episode:1 Score:0.25
[(1, 0), (1, 1), (1, 2), (1, 3), (1, 4), (0, 4), (0, 5), (0, 6), (0, 7), (0, 8), (0, 9), (1, 9), (2, 9), (3, 9), (4, 9), (5, 9), (4, 9), (5, 9), (5, 8), (5, 7), (5, 6), (5, 5), (6, 5), (6, 4), (6, 3), (7, 3), (7, 2), (7, 1), (7, 0), (6, 0), (7, 0), (8, 0), (9, 0), (9, 1), (9, 2), (9, 3), (9, 4), (9, 5), (9, 6), (8, 6), (8, 7), (8, 8), (9, 8), (9, 9)]
target found
Episode:2 Score:1.0
[(1, 0), (1, 1), (1, 2), (1, 3), (1, 4), (0, 4), (0, 5), (0, 6), (0, 7), (0, 8), (0, 9), (1, 9), (2, 9), (3, 9), (4, 9), (5, 9), (5, 8), (5, 7), (5, 6), (5, 5), (6, 5), (6, 4), (6, 3), (7, 3), (7, 2), (7, 1), (7, 0), (8, 0), (9, 0), (9, 1), (9, 2), (9, 3), (9, 4), (9, 5), (9, 6), (8, 6), (8, 7), (8, 8), (9, 8), (9, 9)]
Hit a boundary
Hit obstacle
Hit a boundary
target found
Episode:3 Score:-2.1500000000000004
```

[(1, 0), (1, 1), (1, 2), (1, 3), (1, 4), (0, 4), (0, 4), (0, 5), (0, 6), (0, 7), (0, 8), (0, 9), (1, 9), (2, 9), (3, 9), (4, 9), (5, 9), (5, 8), (5, 7), (5, 6), (5, 5), (6, 5), (6, 4), (6, 3), (7, 3), (7, 4), (7, 4), (7, 3), (7, 2), (7, 1), (7, 0), (8, 0), (9, 0), (9, 0), (9, 1), (9, 2), (9, 3), (9, 4), (9, 5), (9, 6), (8, 6), (8, 7), (8, 6), (8, 7), (8, 8), (9, 8), (9, 9)]

Reiforcement learning can prove to be more useful when more incentives are provided such

A* path finding:

Given maze is:
```
[[1 0 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 0 1 1 1 1]
 [1 1 1 1 1 0 1 1 1 1]
 [0 0 1 0 0 1 0 1 1 1]
 [1 1 0 1 0 1 0 0 0 1]
 [1 1 0 1 0 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 0 0 0 0]
 [1 0 0 0 0 0 1 1 1 1]
 [1 1 1 1 1 1 1 0 1 1]]
```
The A* path is:
[(0, 0), (1, 0), (1, 1), (2, 0), (1, 2), (2, 1), (1, 3), (2, 2), (0, 2), (1, 4), (2, 3), (3, 2), (0, 3), (2, 4), (0, 4), (0, 5), (0, 6), (0, 7), (1, 6), (0, 8), (1, 7), (2, 6), (0, 9), (1, 8), (2, 7), (1, 9), (2, 8), (3, 7), (2, 9), (3, 8), (3, 9), (4, 9), (5, 9), (6, 9), (5, 8), (6, 8), (5, 7), (6, 7), (5, 6), (6, 6), (5, 5), (6, 5), (7, 5), (6, 4), (4, 5), (7, 4), (6, 3), (3, 5), (7, 3), (5, 3), (6, 2), (7, 2), (4, 3), (6, 1), (7, 1), (5, 1), (6, 0), (7, 0), (8, 0), (4, 1), (5, 0), (9, 0), (4, 0), (9, 1), (9, 2), (9, 3), (9, 4), (9, 5), (9, 6), (8, 6), (8, 7), (8, 8), (8, 9), (9, 8), (9, 9)]

Uniform Cost Search:

Given maze is:
```
[[1 0 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 0 1 1 1 1]
 [1 1 1 1 1 0 1 1 1 1]
 [0 0 1 0 0 1 0 1 1 1]
 [1 1 0 1 0 1 0 0 0 1]
 [1 1 0 1 0 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 0 0 0 0]
 [1 0 0 0 0 0 1 1 1 1]
 [1 1 1 1 1 1 1 0 1 1]]
The UCS path is:
[(0, 0), (1, 0), (1, 1), (2, 0), (1, 2), (2, 1), (0, 2), (1, 3), (2, 2), (0, 3), (1, 4), (2, 3), (3, 2), (0,
4), (2, 4), (0, 5), (0, 6), (0, 7), (1, 6), (0, 8), (1, 7), (2, 6), (0, 9), (1, 8), (2, 7), (1, 9), (2, 8),
(3, 7), (2, 9), (3, 8), (3, 9), (4, 9), (5, 9), (5, 8), (6, 9), (5, 7), (6, 8), (5, 6), (6, 7), (5, 5), (6,
6), (4, 5), (6, 5), (3, 5), (6, 4), (7, 5), (6, 3), (7, 4), (5, 3), (6, 2), (7, 3), (4, 3), (6, 1), (7, 2),
(5, 1), (6, 0), (7, 1), (4, 1), (5, 0), (7, 0), (4, 0), (8, 0), (9, 0), (9, 1), (9, 2), (9, 3), (9, 4), (9,
5), (9, 6), (8, 6), (8, 7), (8, 8), (8, 9), (9, 8), (9, 9)]
```

Best first Search:

```
[[1 0 2 2 2 2 2 2 2 2]
 [2 2 2 1 1 0 1 1 1 2]
 [1 1 1 1 1 0 1 1 1 2]
 [0 0 1 0 0 1 0 1 1 2]
 [1 1 0 1 0 1 0 0 0 2]
 [1 1 0 1 0 2 2 2 2 2]
 [2 2 2 2 2 2 1 1 1 1]
 [2 1 1 1 1 1 0 0 0 0]
 [2 0 0 0 0 0 2 2 2 2]
 [2 2 2 2 2 2 2 0 1 2]]
```

Where 2 indicates the path traversed.

CHAPTER-5:-CONCLUSION AND FUTURE ENHANCEMENTS

Conclusion:

Our project provides UI visualizations to different maze traversal methods

Our exploration encompassed five distinct path traversal methods: Uniform Cost Search (UCS), A* Search, Breadth-First Search (BFS), Depth-First Search (DFS), and reinforcement learning (RL), each offering unique advantages and limitations in navigating complex mazes.

UCS, focusing on exploring the least-costly paths, excels in finding the most optimal solution. It guarantees optimality but can become computationally expensive, especially in mazes with numerous paths or intricate structures. A* Search, leveraging heuristic information, strikes a balance between optimality and computational efficiency. Its use of heuristic estimates enables faster convergence towards the goal, but the choice of heuristic function can significantly impact performance.

BFS guarantees finding the shortest path but may be impractical for large or infinite graphs due to its exhaustive nature. While BFS ensures optimality and completeness, its memory requirements grow exponentially with the branching factor, limiting scalability. DFS, prioritizing depth-first exploration, is memory efficient and suitable for large, branching mazes. However, it may overlook shorter paths in favor of deeper exploration, leading to suboptimal solutions.

RL introduces a dynamic approach to maze navigation by learning optimal strategies through interaction with the environment and feedback in the form of rewards. While RL offers adaptability and autonomous decision-making capabilities, its performance may vary depending on the complexity of the maze environment and the design of the reward function.

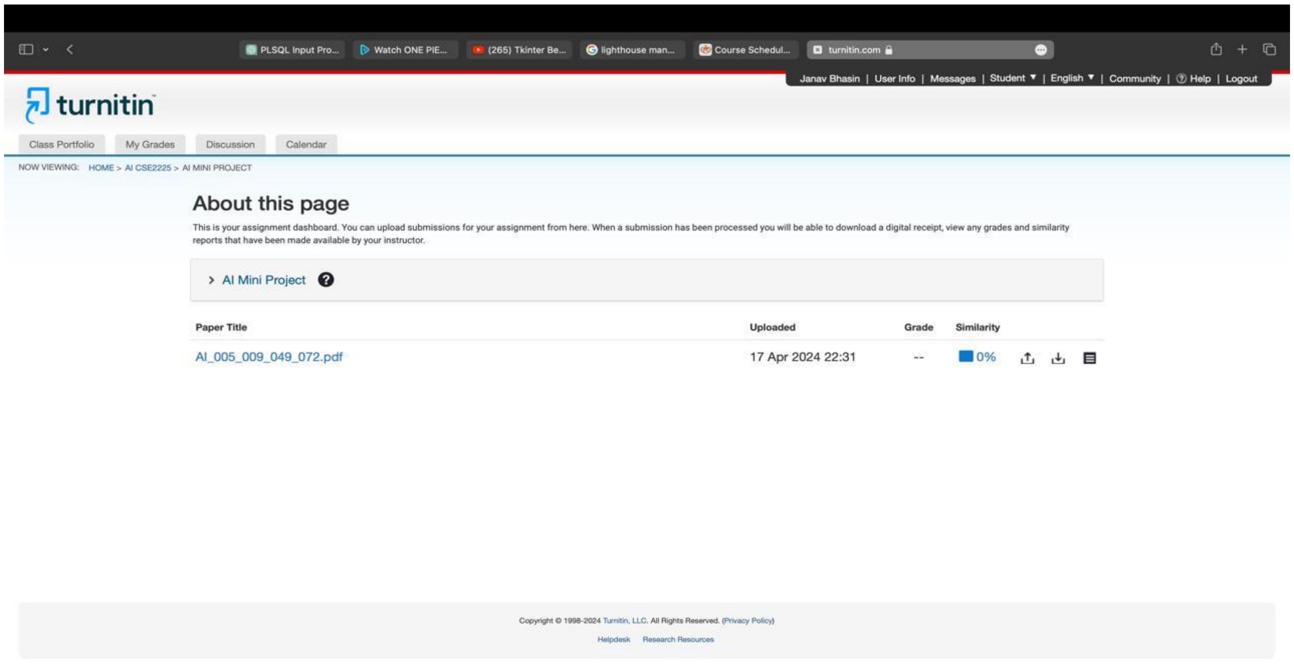
In conclusion, each path traversal method presents a trade-off between optimality, computational efficiency, and memory usage. Understanding the strengths and limitations of each method is crucial for selecting the most appropriate approach based on specific maze characteristics and computational constraints.

Future Enhancements:

Future enhancements could prioritize the development of parallel and distributed algorithms to leverage the computational power of modern hardware architectures, such as multi-core processors and GPU accelerators. By parallelizing the exploration and evaluation of paths, algorithms can effectively scale to larger mazes and reduce overall computation time, enabling real-time or near-real-time maze-solving capabilities.

Lastly, there is a growing interest in applying maze-solving algorithms to real-world applications beyond gaming and robotics, such as logistics, urban planning, and disaster response. Future research could explore the adaptation of maze-solving algorithms to address these practical challenges, potentially leading to innovative solutions for optimizing resource allocation, route planning, and emergency evacuation strategies in complex environments.

Turnitin Result:



Janav Bhasin | User Info | Messages | Student ▼ | English ▼ | Community | ⓘ Help | Logout

turnitin

Class Portfolio | My Grades | Discussion | Calendar

NOW VIEWING: HOME > AI CSE2225 > AI MINI PROJECT

About this page

This is your assignment dashboard. You can upload submissions for your assignment from here. When a submission has been processed you will be able to download a digital receipt, view any grades and similarity reports that have been made available by your instructor.

> AI Mini Project

Paper Title	Uploaded	Grade	Similarity
AI_005_009_049_072.pdf	17 Apr 2024 22:31	--	0%

Copyright © 1998-2024 Turnitin, LLC. All Rights Reserved. (Privacy Policy)

Helpdesk Research Resources