East West University

Department of Computer Science and Engineering

Semester: Spring 2024

Assignment Report

Course Code: CSE366

Course Title: Artificial intelligence

Section: 03

Assignment No: 01

Submitted By,

Raiyan Gani

ID: 2021-2-60-120

Submitted To,

Dr. Mohammad Rifat Ahmmad Rashid

Assistant Professor

Department of Computer and Science Engineering

East West University

Date of Report Submission: 05 March, 2024

Assignment: Enhanced Dynamic Robot Movement Simulation

Objective: Design and implement an advanced simulation environment for a robot navigating through a dynamically created grid. This project aims to deepen understanding of basic programming concepts, object-oriented programming (OOP), algorithms for navigation and pathfinding, task optimization, safety, and energy management strategies

Initial: We are working with two popular algorithms which are Uniform Cost Search and A-star search.

Environment: Environment class serves as a simulation framework for an agent navigating through a grid-based environment. The primary goal is to create a representation of the agent's surroundings, enabling it to move from an initial position to a specified goal while avoiding obstacles. The environment is initialized with a grid layout, where 1 denotes obstacles and 0 represents open spaces. The starting position (start) and goal position (goal) are essential attributes.

The class offers methods to facilitate the agent's movement and decision-making. The actions method determines the feasible actions the agent can take from a given state, considering grid boundaries and obstacles. These actions include moving 'UP,' 'DOWN,' 'LEFT,' or 'RIGHT.' The result method calculates the new state resulting from taking a specific action, allowing the agent to transition between states during its exploration.

A critical aspect is the is_goal method, which checks whether the agent has reached its destination by comparing the current state with the predefined goal. This method assists in determining when the agent has successfully completed its navigation task.

Overall, the code establishes a flexible environment for grid-based pathfinding problems, offering a foundation for developing and testing algorithms related to agent navigation, obstacle avoidance, and goal achievement in a simulated grid world.

Priority Queue: The PriorityQueue class is designed as a wrapper around Python's heapq module, providing a priority queue data structure. This data structure is essential for managing elements with associated priorities efficiently. The queue allows elements to be inserted with a priority value, and retrieval is done in ascending order of priority. The class includes methods for checking if the queue is empty, inserting an element with a specified priority, and retrieving the element with the highest priority.

The Node class represents a state within a search tree, commonly used in algorithms like A* or Dijkstra's. Each node encapsulates information about the current state (position of the agent in a grid), its parent node in the search tree, the action taken to reach this state, the path cost from the start node to this node, and the battery level (represented as a percentage).

The Node class also includes a special comparison method (__lt__) to facilitate the sorting of nodes within the priority queue. This method compares nodes based on their path costs, enabling them to be ordered in ascending order of cost when inserted into the priority queue.

Agent with Uniform Cost Search: a variant of Dijkstra's algorithm known as Uniform Cost Search for finding the minimum-cost path from a start node to a goal node in a weighted graph. The algorithm utilizes a priority queue and additional data structures to manage the exploration process efficiently.

The process begins by initializing a priority queue with the start node and its cost (initialized to zero). Three dictionaries are set up to store information about each node: the parent node, the cost to reach the node, and the battery level (with one as the initial value for the start node).

The algorithm then iteratively dequeues the node with the lowest cost from the priority queue. If the dequeued node is the goal node, the algorithm returns the path and battery levels by tracing back through the parent nodes. Otherwise, for each neighbor of the current node, it calculates a new cost by adding one to the current cost and a new battery level by subtracting 0.1 from the current level (representing a 10% decrease for each move).

If the battery level drops below or equal to 0.1, the algorithm recharges the battery to one and increments the recharge count. The algorithm then updates the cost, priority, parent, and battery level of the neighbor if it is not in the cost dictionary or if the new cost is lower than the previous cost. The updated neighbor is enqueued back into the priority queue.

In case no valid path is found, the algorithm prints a message and returns an empty list and an empty dictionary. The algorithm includes a helper function to reconstruct the path from the parent dictionary, starting from the goal node and working backward to the start node. The resulting path is then reversed and returned.

In essence, the algorithm efficiently explores the graph, adjusting costs and battery levels as needed, and provides a solution by returning the optimal path and associated battery levels from the start node to the goal node.

Agent with A-star Search: A star search is an informed best-first search algorithm that efficiently determines the lowest cost path between any two nodes in a directed weighted graph with non-negative edge weights123

The code defines a class Agent_aStar that takes an environment as an argument and implements the a_star_search method. The method takes an optional heuristic function as an argument and returns a path and a dictionary of battery levels for each node in the path. The method also keeps track of the number of times the battery was recharged.

The code uses a priority queue to store the nodes in the open list, where the priority is determined by the sum of the path cost and the heuristic value. The code also uses dictionaries to store the parent, the cost, and the battery level of each node. The code iterates through the open list until it finds the goal node or the list is empty. For each node, it generates its successors and updates their values according to the A star search algorithm. The code also checks the battery level of each node and recharges it if it falls below a threshold. The code allows for a maximum number of retries if no valid path is found. The code also defines a helper method reconstruct_path that returns the path from the start node to the current node by following the parent pointers.