**Assignment1**

**Name:** Maherun Nessa Isty

**ID:** 2021-2-60-86

**Course code:** CSE366

**Course title:** Artificial Intelligence

**Section:** 3

**Enhanced Dynamic Robot Movement Simulation**

**Creating node/State:**

I have used priority queue data structure (Python's heapq module). It efficiently manages elements with associated priorities, allowing insertion with priority values and retrieval in ascending order of priority. The class includes methods for checking if the queue is empty, inserting elements with specified priorities, and retrieving the element with the highest priority. The Node class represents a state within a search tree, commonly used in algorithms like A\* and UCS. It encapsulates information about the current state, parent node, action taken to reach this state, path cost from the start node, and battery level. It includes a comparison method (**lt**) for sorting nodes based on path costs, facilitating their ordered insertion into the priority queue.

**Environment:**

The Environment class serves as a simulation framework for agent navigation in a grid-based environment. It provides methods for determining feasible actions, calculating new states resulting from actions, and checking if the agent has reached its goal. This enables the agent to navigate from an initial position to a specified goal while avoiding obstacles. The class is designed to support the development and testing of algorithms related to agent navigation, obstacle avoidance, and goal achievement in a simulated grid world. Here,I have attached two new attributes named state\_requiring\_charging and battery\_comsumption. Also added a method to check if the grid is valid (not completely blocked by obstacles and has valid start and goal positions and ensures that both the start and goal positions are free spaces)

New Attributes:

* + states\_requiring\_charging: This attribute is a set that keeps track of states where the agent needs to recharge its battery.
  + battery\_consumption: This attribute represents the battery consumption per move in percentage. It's set to 10% of the battery capacity, which is assumed to be 100 units.

**UCS\_Agent:**

In this agent section I have added some mechanism for enhancing the functionality of the agent to handle scenarios where energy (battery) constraints are involved. The agent now not only finds the lowest-cost path but also considers energy management, ensuring it can reach the goal while maintaining a sufficient battery level.

1. Battery Management:
   * “battery\_level” This includes tracking the battery level for each node during search (battery\_level attribute in the Node class).
   * The battery level is updated for each action taken in the search loop, considering the battery consumption per move(self.env.battery\_consumption).
   * If the battery level drops to zero during exploration, the agent must recharge the battery to continue. In such cases, the battery level is set back to 100%.
   * States requiring charging are tracked using the states\_requiring\_charging attribute in the Environment class.
2. Handling Invalid Grids:
   * Before starting the search, the Agent\_ucs class checks if the grid is valid using is\_valid method in the Environment class.
   * If the grid is invalid, the search terminates early and returns an empty path along with an empty dictionary for battery levels.
3. Tracking Battery Levels:
   * The battery\_levels dictionary is introduced to keep track of the battery levels for each state encountered during the search.
   * This dictionary is updated along with the “cost\_so\_far dictionary” in the search loop.

**A\*\_agent:**

In this agent, I have added several new features related to battery management have been added. The added attributes are working in the way that the UCS agent are working.

* “battery\_level” attribute represents the current battery level of the agent, which is set to 100 at the beginning.
* During the search process, the battery level of the current node is updated with each action taken. After each move, the battery level is reduced by the amount specified by “self.env.battery\_consumption”.
* If the battery level drops to or below 0 after an action, it is recharged to 100%, and the state is added to the set “self.env.states\_requiring\_charging”, indicating that it needs recharging.

Top of Form

**Visualization**:

In the “visualize\_grid\_and\_path” function, several additions are made to accommodate battery level visualization and charging state indication.

1. Battery Level Visualization:
   * The function now takes an additional argument battery\_levels, which is a dictionary containing the battery level of each grid position.
   * Inside the function, for each position in the path, it retrieves the corresponding battery level from the battery\_levels dictionary using battery\_levels.get(pos, 100). If the battery level is not available (i.e., the position is not in the dictionary), it defaults to 100.
   * It plots the battery level as text on each grid position along the path, indicating the battery level percentage.
2. Charging State Indication:
   * If the battery level is at 100% and the position requires charging (if pos in environment.states\_requiring\_charging), it plots a yellow star marker ('y\*') at that position with an increased markersize to make it more visible.

Lastly, Python script generates a random grid-based with obstacles and defines start and goal positions within it. Using A\* Search and Uniform Cost Search algorithms, it finds the optimal path from the start to the goal while avoiding obstacles.The script creates an Environment object to represent the grid and an Agent\_ucs or A\*\_search object to perform the search. After finding the optimal path, it prints the solution path and battery levels. Finally, it visualizes the path on the grid.