## Q1.a) Knowledge representation & Planning

The architecture of autonomous embodied robots will be composed of several layers and components that work together to enable the robot to perceive its environment, make decisions, and perform actions.

**Below is the sequential list of layers from bottom to top**

*Perception Layer* - Sensors(to capture real-time data from the environment)

*Knowledge Representation Layer* -

*Semantic Network* (to represent relationships between various objects and concepts)

*State Representation Layer* - Frames (Encapsulate knowledge about specific objects and situations; for ex <id,location,weight,default\_value>)

*Decision Making Layer* - Production Rules(Define the logic for decision-making based on conditions and actions)

*Action Execution Layer* - Actuators(to executes planned actions)

*Feedback loop* (to continuously monitors the outcome of actions and updates the perception and state layers)

**Handling Challenges of Ambiguous or Incomplete Information**

* Probabilistic Reasoning (if the location of an item is uncertain, the system assigns probabilities to possible locations and updates them as new information is obtained.)
* Default Reasoning with Frames (default values that the system uses when specific information is missing)
* Dynamic Updating(The environment state is dynamically updated based on new sensor data. If a new obstacle is detected, it is added to the environment model, and the decision-making process adjusts accordingly)
* Redundancy and Cross-Checking(Multiple sensors provide overlapping data, which helps cross-verify information and reduce ambiguity. For instance, if one sensor detects an obstacle, it is cross-checked with data from another sensor)

**Unknown Obstacle Encounter**

1. **Perception Layer**: Sensors detect an unexpected obstacle.
2. **Knowledge Representation Layer**: Updates the semantic network and frames to include the new obstacle.
3. **State Representation Layer**: Adjusts the environment state to reflect the updated obstacle location.
4. **Decision-Making Layer**:
   * **Inference Engine**: Triggers production rules to assess the new situation.
   * **Planner**: Re-plans the route to avoid the obstacle.
5. **Action Execution Layer**: Actuators execute the new plan to navigate around the obstacle.
6. **Feedback Loop**: Monitors the outcome, ensuring the obstacle is successfully avoided, and updates the perception and state layers accordingly.

**Flowchart to handle unknown obstacles -**

A[Detect Obstacle] --> B{Is Obstacle Known?}

B -- Yes --> C[Update Known Obstacle]

B -- No --> D[Add New Obstacle]

C --> E[Update Environment State]

D --> E

E --> F[Re-plan Route]

F --> G[Execute New Plan]

G --> H[Monitor Outcome]

H --> I{Obstacle Avoided?}

I -- Yes --> J[Update Success]

I -- No --> K[Adjust Plan and Retry]

K --> G

J --> L[Continue Task]

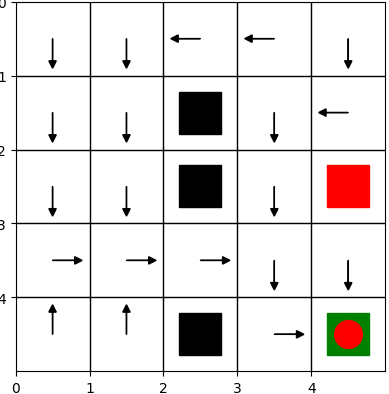
**Dynamic Item Location**

1. **Perception Layer**: Detects that an item is not in its expected location.
2. **Knowledge Representation Layer**: Updates the semantic network with the item's new possible locations.
3. **State Representation Layer**: Adjusts the environment state to include the uncertainty about the item's location.
4. **Decision-Making Layer**:
   * **Inference Engine**: Applies probabilistic reasoning to determine the most likely location of the item.
   * **Planner**: Formulates a search plan to locate the item.
5. **Action Execution Layer**: Executes the search plan.
6. **Feedback Loop**: Continuously updates the system with new information as the item is being searched.

## Q1.b)

#### **Environment Representation**

***Assumption***: Consider the below Grid as the warehouse environment representation, where the embodied robot is to optimize its path to collect a maximum number of items and return to the charging station before the battery runs out.



Where

Black box - Empty

Red - Obstacles

Green\_Red - Charging Point

Allowed Movements - 2-D(up, down, left, or right)

Arrowed box - Item locations & movement paths

Current State - Above diagram represents the optimized path the robot should take to avoid collision from it’s different current position.

***Another form of Representation***

Above warehouse environment can also be represented as below, where each cell can have the following states:

* **Empty (E):** No obstacles or items.
* **Item (I):** Contains an item to be collected.
* **Obstacle (O):** Contains an obstacle that the robot cannot pass through.
* **Charging Station (C):** A location to recharge its battery.

grid = [

[I, I, I, I, I],

[I, I, E, I, I],

[I, I, E, I, O],

[I, I, I, I, I],

[I, I, E, I, C]

]

**Planning Strategy**

#### **A\* Algorithm**

This is efficient in finding the shortest path with the least cost. A\* combines the benefits of Dijkstra's algorithm and greedy best-first search by using a heuristic function to estimate the cost to reach the goal.

* **Cost Function (g):** Actual cost from the start node to the current node.
* **Heuristic Function (h):** Estimated cost from the current node to the goal (using Manhattan distance in the grid).
* **Total Cost (f):** Sum of the cost function and heuristic function (f = g + h).

#### **Planning Algorithm**

1. **Initialization:**
   * Start at the robot's initial position.
   * Initialize an open list with the starting node.
   * Initialize a closed list to keep track of visited nodes.
2. **Path Finding:**
   * While the open list is not empty, perform the following steps:
     1. Select the node with the lowest f value from the open list.
     2. If the selected node is the charging station and battery level is low, add the path to the final path and recharge the battery.
     3. Generate the neighboring nodes (up, down, left, right).
     4. For each neighbor, calculate the g, h, and f values.
     5. If the neighbor is not in the open or closed list, add it to the open list.
     6. If the neighbor is already in the open list, update its g, h, and f values if a better path is found.
   * Continue until the robot collects the maximum number of items and returns to the charging station before the battery runs out.
3. **Re-planning:**
   * If dynamic changes occur (new obstacles or items), update the grid representation.
   * Re-run the A\* algorithm from the current position to adapt to the changes.

### **Handling Dynamic Changes**

1. **Dynamic Update:**
   * Continuously monitor the environment using sensors.
   * If new obstacles or items are detected, update the grid representation accordingly.
2. **Re-planning Trigger:**
   * Trigger the re-planning process if significant changes are detected (e.g., new obstacle blocking the planned path).
   * Use the current position and state of the robot to re-run the A\* algorithm and find a new optimal path.

### **Evaluation of Efficiency**

**Computational Complexity:**

* The A\* algorithm has a time complexity of O(bd)O(b^d)O(bd), where b is the branching factor and d is the depth of the solution. In a grid, this is generally efficient as the heuristic function (Manhattan distance) provides good guidance towards the goal.

**Quality of Path:**

* A\* ensures that the path found is optimal in terms of the total cost, combining both the distance traveled and heuristic estimates.
* The algorithm adapts well to dynamic changes, ensuring the robot can always find a feasible path even with updates to the environment.

### 

***Pseudo code for the Planning Algorithm-***

***BEGIN***

***Step 1: Start - Initialize grid and robot state***

***Step 2: Initialize Grid and Robot State***

***OPEN\_LIST = empty list***

***CLOSED\_LIST = empty list***

***Add start node to OPEN\_LIST***

***Step 3: Add Start Node to Open List***

***WHILE OPEN\_LIST is not empty DO***

***Step 4: Open List Not Empty***

***Step 5: Select Node with Lowest f***

***current\_node = node in OPEN\_LIST with the lowest f***

***Step 4: Is Charging Station and Battery Low***

***IF current\_node is a charging station AND battery is low THEN***

***Step 5: Recharge Battery***

***Step 6: Re-plan path if Dynamic Change***

***ELSE***

***Step 7: Generate Neighbors***

***neighbors = generate neighbors of current\_node***

***FOR EACH neighbor IN neighbors DO***

***Step 8: Calculate g, h, f for Neighbors***

***neighbor.g = current\_node.g + distance(current\_node, neighbor)***

***neighbor.h = heuristic(neighbor, goal)***

***neighbor.f = neighbor.g + neighbor.h***

***Step 9: Neighbor Not in Open or Closed List***

***IF neighbor NOT IN OPEN\_LIST AND neighbor NOT IN CLOSED\_LIST THEN***

***Step 10: Add Neighbor to Open List***

***ELSE***

***Step 11: Better Path Found***

***IF better path to neighbor is found THEN***

***Step 12: Update neighbor with new values of g, h, f for Neighbor***

***Add current\_node to CLOSED\_LIST***

***Move current\_node from OPEN\_LIST to CLOSED\_LIST***

***Step 12: Continue loop***

***END WHILE***

***Step 13: Maximum Items Collected and Return to Charging Station***

***Step 14: End***

***End***

***END***

***Explanation:***

Initialization: grid & robot state initialized, including the open and closed lists, and the start node is added to the open list.

Main Loop: enters a loop that continues as long as there are nodes in the open list.

Node Selection: node with the lowest f value is selected from the open list.

Battery Check and Recharge: If the current node is a charging station and the battery is low, the robot recharges. If there are dynamic changes, the path is re-planned.

Neighbor Generation: Neighbors of the current node are generated.

Cost Calculation: For each neighbor, the g, h, and f values are calculated.

Neighbor Handling: If the neighbor is not in the open or closed list, it is added to the open list. If a better path to the neighbor is found, the neighbor's g, h, and f values are updated.

List Management: The current node is moved from the open list to the closed list.

End Condition: The loop continues until the open list is empty, and then the robot ensures maximum items are collected and returns to the charging station.