**CONFLICT - BASED SEARCH**

Multi Agent Path Finding

**Team Members**

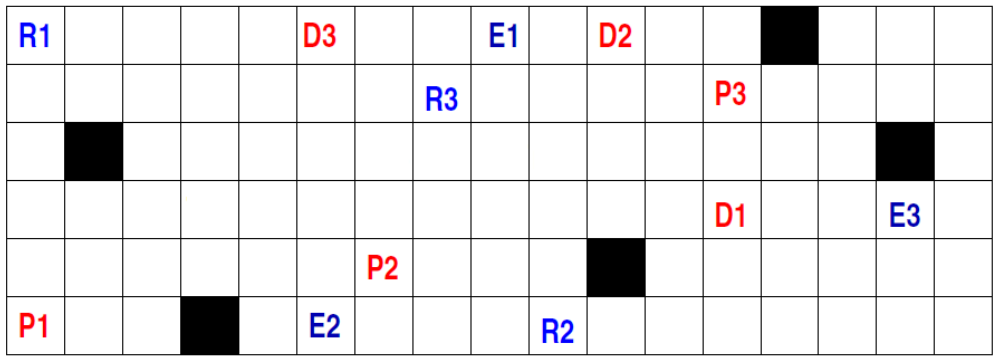
Anamika Kumari (20AE3AI01)

Niharika Singh (20MI3AI17)

Kartik Narayan Sahoo (20AE3AI04)

1. **Problem statement**

A certain shop floor is represented as a 2-D grid of size n x m, as shown in the figure below. In the shop floor, robots are employed to perform shipment tasks, where a task involves shipping a component from a designated pick-up cell location to a destination cell. There are ‘k’ robots and ‘k’ shipment tasks. Each robot performs a single task. A robot can move at most one cell (either vertically or horizontally) at a time step. A cell cannot be simultaneously occupied by more than one robot. Black coloured cells are obstacles and cannot be traversed through. All robots start at the same time. The ith robot starts from cell location Ri, picks up one of the components (say) at location Pj, delivers it in cell Ej and finally moves to cell Di. The objective is to choose the appropriate task for each robot and determine its corresponding travel path, so that overall completion time for the work schedule involving all tasks is minimised.



1. **A step-wise depiction of the solution approach**

1. We initiate our script by importing the necessary libraries. This ensures we have access to essential tools and functionalities required for our Python program.

2. Then we define the dimensions of our grid, comprising 6 rows and 17 columns. We're defining a task for three agents (`num\_agents = 3`). We set their initial positions, designated pickup spots, delivery destinations, goal positions and obstacle locations. This information serves as the foundational setup for our task.

3. We define a class named `Node`. This class encapsulates the essential attributes for path planning, including the current state, its parent node, the cost from the start node to the current node (`cost`), and the heuristic cost to the goal (`heuristic`). Moreover, we implement a comparison method (`\_\_lt\_\_`) which determines the ordering of nodes based on their total cost (`f(n) = g(n) + h(n)`), facilitating selection of the node with the minimal cost.

4. This function, `is\_valid\_location(state)`, checks if a given position on the grid (`state`) is within the grid's boundaries. It verifies that the `y` coordinate is greater than or equal to zero and less than the total number of rows, and similarly, it checks that the `x` coordinate is greater than or equal to zero and less than the total number of columns. This ensures that the position is valid within the grid's dimensions.

5. `is\_location\_constrained(agent, state, timestep, constraints)` function determines if a particular agent's position at a specific timestep is subject to predefined constraints. It achieves this by comparing the agent, state, and timestep with the constraints provided. If a matching constraint is found, the function returns true; otherwise, it returns False.

6. The function `is\_obstacle(state)` checks if a given `state` (representing a position on the grid) corresponds to an obstacle. The function `is\_obstacle(state)` simply checks if the provided `state` is among the obstacles listed in the `obstacles` list. If it is, the function returns `True`; otherwise, it returns `False`.

7. `evaluate\_cost\_heuristic(state, goal)` calculates a heuristic estimation of the cost to reach a goal state from a given current state. The calculation uses the Manhattan distance, acting as an estimate of the required cost (in grid cells) to reach the goal.

8. The function **`get\_valid\_neighbors(agent, state, timestep, constraints)`** is designed to find the valid neighboring states of a given state in a grid. This function systematically examines neighboring positions in four directions (right, left, up, and down) from a given state. It checks if each potential neighbor is within the grid bounds, not obstructed by an obstacle, or constrained by predefined conditions. If a position meets all these criteria, it is added to the list of valid neighbors, which is then returned.

9. Considering specified constraints, we then define a search algorithm to find a path from an initial state to a goal state for a given agent. This `search` function utilizes a priority queue (`OPEN`) and a loop to explore possible states. It follows A\* search algorithm principles, considering costs and heuristics to find the optimal path from the initial state to the goal state while adhering to specified constraints. The function returns the path if one is found and `None` otherwise.

10. The function `validate\_paths(node)` checks the validity of a given solution with respect to conflicts between agents. This `validate\_paths` function checks for conflicts between agents in a given solution. It determines conflicts by comparing the positions of agents at each time step. If a conflict is found, it returns the details of the conflict; otherwise, it returns `None`, indicating that the solution is valid. This function verifies the correctness of generated paths for multiple agents.

11.The function `find\_paths(initial\_pos, final\_pos, constraints)` generates paths for multiple agents from their respective initial positions to their final positions while considering any specified constraints. The function iterates through each agent's initial and final positions. For each agent, it uses the `search` function to find a path while considering any provided constraints. The resulting paths are collected in a list and returned as the final output.

12. The function `SIC(path)` calculates the overall cost of a set of paths by summing up the lengths of all individual paths. This provides a measure of the total effort or time required to execute the specified paths.

13. The `cbs\_algo(initial\_pos, final\_pos)` function implements the Conflict-Based Search (CBS) algorithm to find a solution for multiple agents to navigate from their initial positions to their respective final positions. The function implements the CBS algorithm to find a solution for multiple agents in a constrained environment. It iteratively explores states, resolves conflicts, and updates the constraints to refine the solution. The process continues until a valid solution is found or all possible states have been explored. If no valid solution is found, the function returns `None’.

# **Solution Approach**

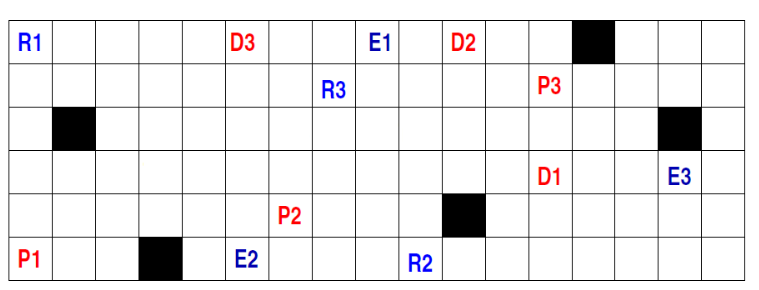
After defining the CBS algorithm, we apply it in three distinct stages to plan paths for our agents. These stages involve navigating from initial positions to pickup locations, then from pickup locations to delivery spots, and finally from delivery locations to the agents' ultimate goal positions.

Upon obtaining optimized paths for each respective stage, we proceed to determine the maximum number of steps or time required to complete each phase of the task. This encompasses the maximum time needed for agents to transition from initial positions to pickup locations, the maximum time from pickup to delivery, and the maximum time from delivery to the goal positions.

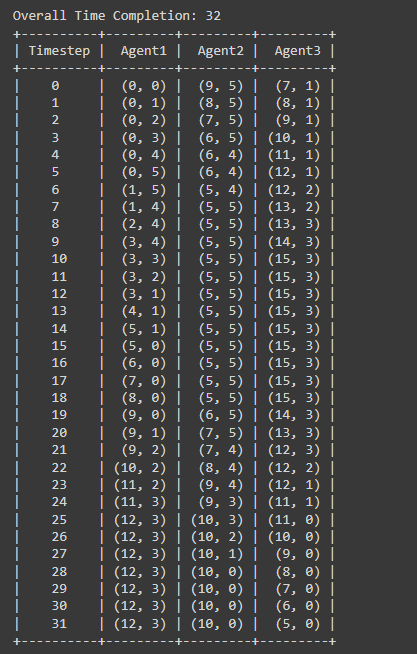
Next, we meticulously concatenate segments of paths corresponding to different task stages for each agent. This ensures that each agent remains stationed at their respective positions until all agents have concluded a given stage. Specifically, all agents complete stage 1, and those that reach their destinations early maintain their positions. Subsequently, upon completion of stage 1 by all agents, they collectively commence stage 2, progressing from pickup to delivery locations. Once all agents successfully accomplish stage 2, they embark on stage 3, transitioning from delivery locations to their final goal positions. The total time taken is computed once all agents have reached their designated goal states.

# **Screenshots of both input and output for various test cases including the one shown in the figure**

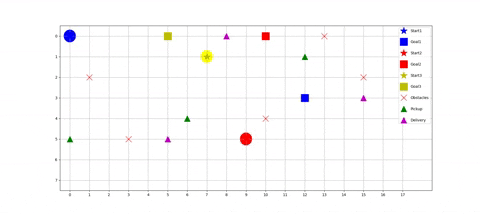
Test case 1: **INPUT**



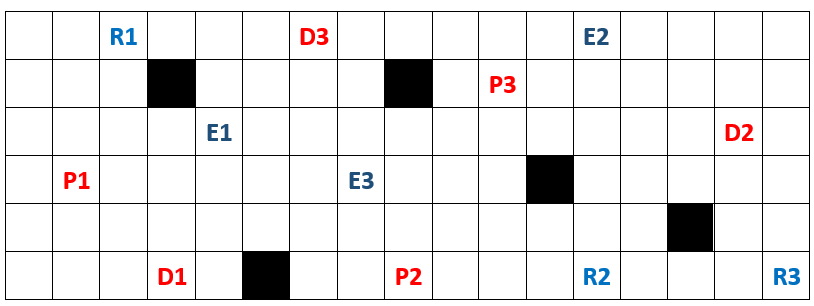
**OUTPUT**

****

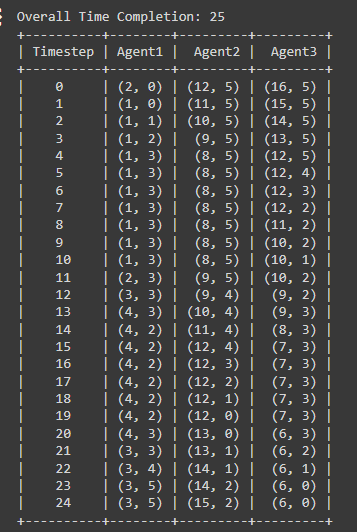
**VISUALISER**

****

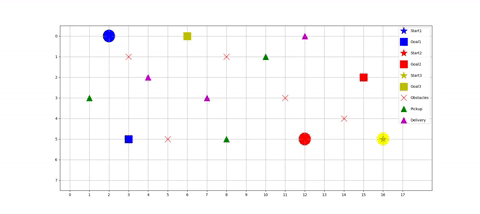
Test case 2: **INPUT**

****

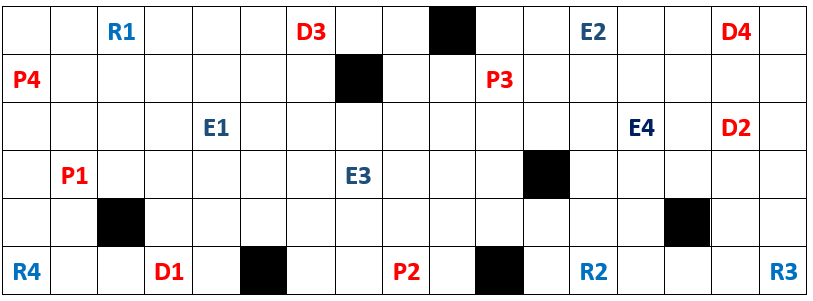
**OUTPUT**



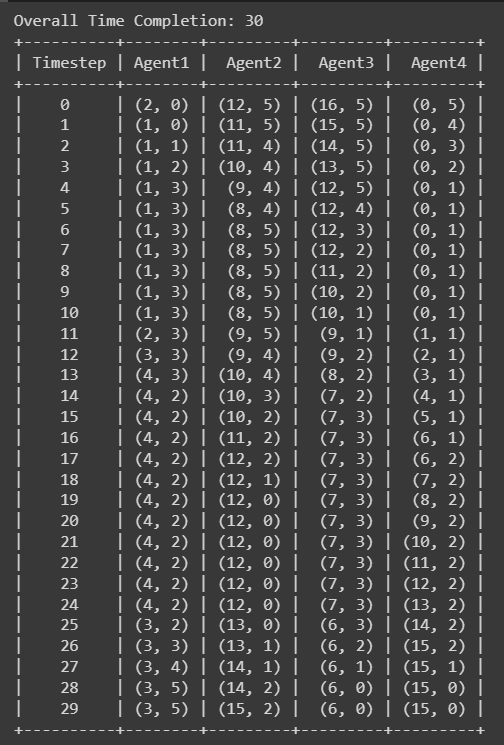
**VISUALISER**

****

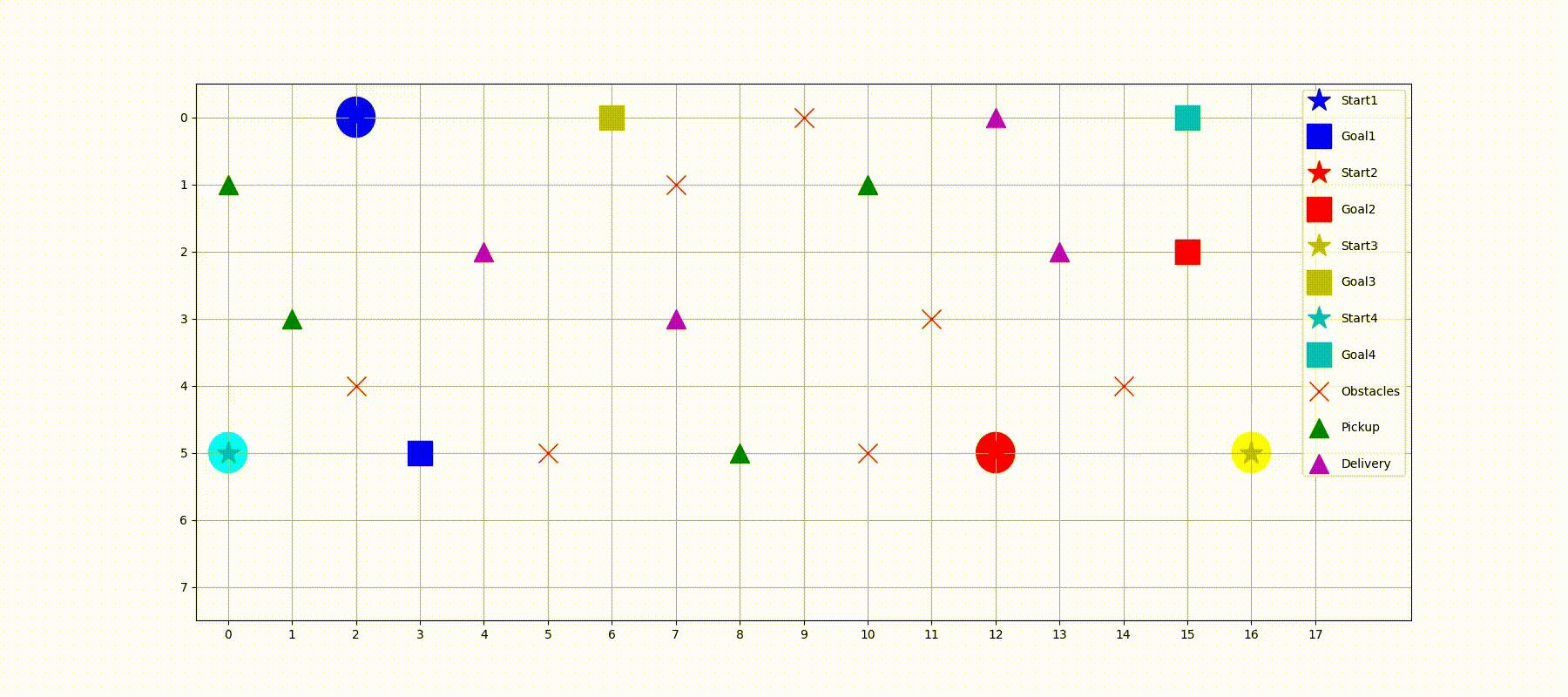
Test case 3: **INPUT**

****

**OUTPUT**

****

**VISUALISER**

****

1. **Analysis and discussion**

Our approach does not yield an optimal solution. It provides a suboptimal solution because our algorithm does not consider the entire path from the initial to the final position simultaneously. Instead, it involves a sequential process: first, we move from the initial positions to the pickup locations, then from pickup to delivery positions, and finally to the goal positions. Consequently, in our approach, once an agent reaches its desired pickup location, it must remain there until all agents reach their respective pickup locations. The same holds true for the transition from pickup to delivery and then to the goal positions. As a result, this approach does not guarantee the most efficient path for minimizing time.

Furthermore, our current approach does not efficiently allocate pickup and delivery tasks to minimize the overall completion time. The problem's complexity increases with a greater number of obstacles and their positions, making it more challenging to identify an optimal path as the obstacle count rises.

To achieve an optimal solution, it is essential to address these issues. This will necessitate a combined solution that addresses both path planning and task allocation problems.

# 

# 