An Offline Multi Agent Path Finding Approach using Multi Label A*(MLA*) and Pre-Scheduling Agent-Task pairs with Iterative Deepening A*(IDA*)

Abrar Hossain, Nithish Kannen, Sankalp Srivastava, Sarthak Chauhan
Indian Institute of Technology, Kharagpur

1. Introduction

Multi Agent Path Finding problem is a complex problem where multiple agents are assigned to perform a set of pickup-delivery tasks in a pre-defined warehouse map. The problem consists of assigning tasks to each agent and designing a path to pickup the task and deliver it in the desired location while avoiding collisions with other agents. Moreover, multiple agents are allowed to reside at certain blocks in the map (temporary storage, initial agent location, final agent location). The goal is to come up with an optimal approach (minimum make span) to carry out the given tasks. Figure 1 is a sample warehouse map.

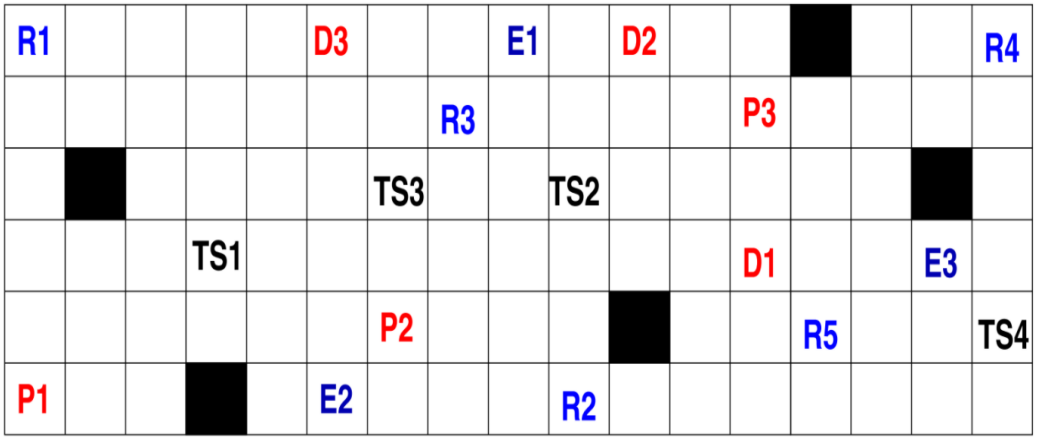

Fig. 1. Map of warehouse given to tackle the multi-agent pathfinding problem. *R- agent initial position, E- agent final position, P-task pickup position, D - task delivery position, T- temporary storage*

2. Methodology

The given problem statement is an offline version of Multi Agent Pickup and Delivery (MAPD), as opposed to the online version where tasks get updated in real time. In our case, all the tasks, task pickup and delivery and robot initial and final positions are available to us beforehand. We propose a novel Agent-Task pair scheduling approach using the conventional **Iterative Deepening A* (IDA*)**[1] algorithm. The heuristics used in this approach is an underestimate of the start (initial position)-> goal (final position of agent after completing all allotted tasks) computed using **Floyd Warshall** [2] Algorithm for All Pair Shortest Paths (APSP) in the given map. The details of how this heuristic is put to use is explained in the coming sections. We then employ an independent Multi Label A* algorithm for each agent using its start and goal locations along with the labels as defined by the scheduler. This algorithm is further explained in the coming sections. There have been several approaches in the past such as [3], [4], [5]. Figure 2 is a visual overview of the complete approach proposed in this paper.

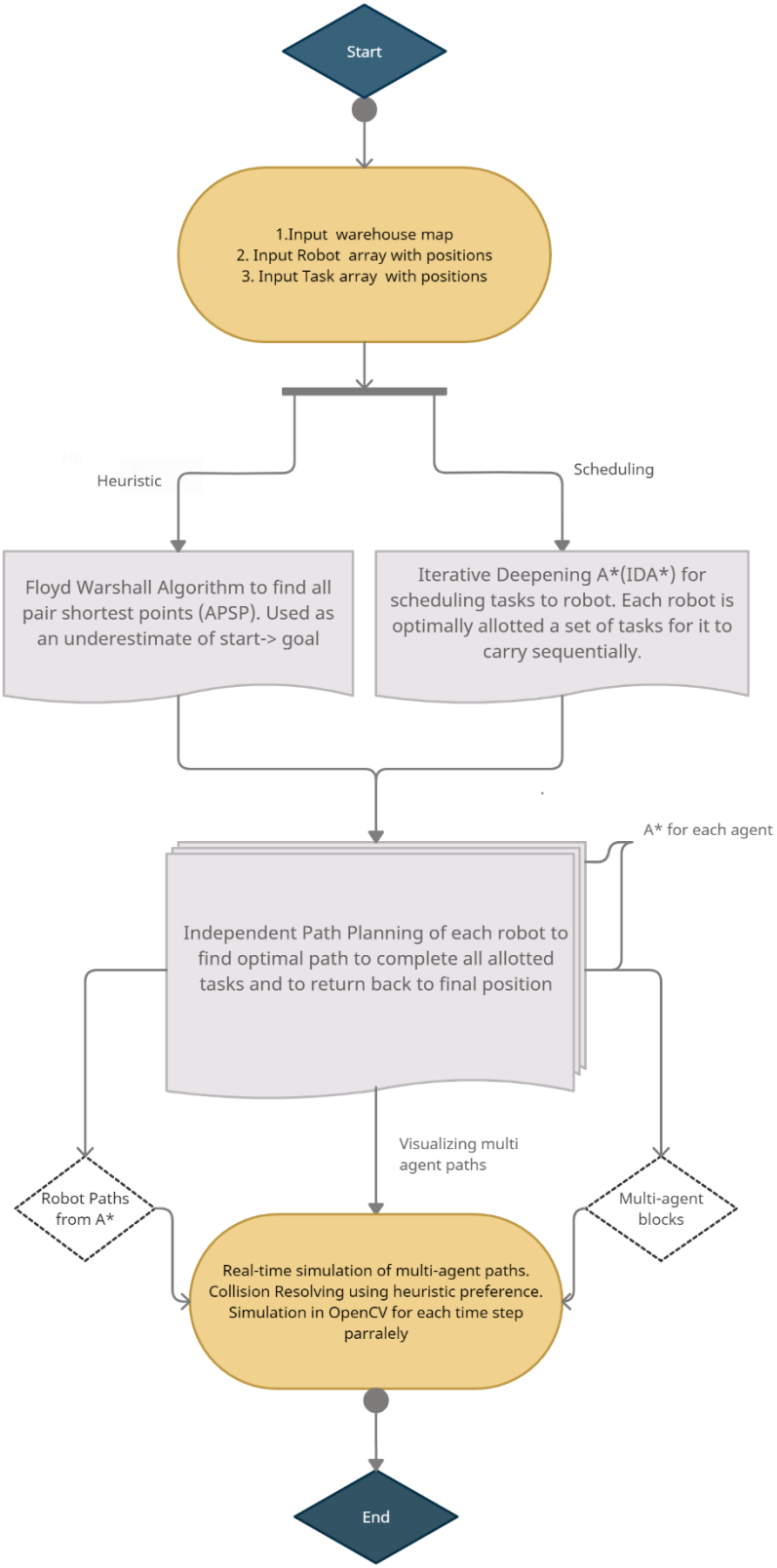

Fig. 2. A visual representation of the approach employed

2.1 Scheduling

This section describes the scheduling process employed in this paper. In other words, this section deals with agent-task pair allocation that leverages the offline allotment of tasks. We allocate a set of tasks for every agent, that has to carried out sequentially. In this approach we do not consider allow Task Swapping between agents. An agent has to complete the task allocated to it. We leverage the fact that all tasks are available to us before-hand, and an efficient algorithm to schedule the tasks can be employed to minimize make span.

2.1.1 Iterative Deepening A*

Scheduling the task to each robot is basically finding the suitable agent-task pair which would give the minimum time. A naive approach would be to find the set of agent-task pair through a brute force search where we will see each and every combination. Here we have used IDA* algorithm for the same. We have created an extensive graph of agent-task pair as nodes connected to each other. Weight of the edge connecting those nodes will depend upon the previous path we have taken to reach the parent node. For that we have used a vector of visited nodes.

In IDA* algorithms we have fixed the max time allowed to visit any path and started a **Depth First Search (DFS)** search for the path, This DFS would backtrack from the node where time consumed for going to that node through that particular path, exceeds this maximum time. And the next limit is changed to the minimum of the time from where it has back tracked. The iteration will continue till we get any of the goal node through any path.As Maximum time limit increases gradually in every iteration guarantee us the optimal solution. Also the number of visited nodesb and time complexity in IDA* is very less as compared to BFS and DFS respectively.

2.2. Heuristics and Multi-Agent Planning

Developing robust heuristics are a critical part of any path finding algorithm. Ideally a good heuristic estimate is an underestimate of the time/distance taken from state-> goal. The closer the heuristic is to the actual value, the better it is. As discussed in class, if h1 and h2 are 2 underestimates(heuristic) of a given state-> goal, h1 < h2 => h2 is a more accurate heuristic. Heuristic designing is a well-studied field in itself, with a lot of studies being conducted even today [6].

2.2.1 Floyd Warshall Algorithm

In our approach, we employed the widely used Floyd Warshall Algorithm to compute shortest distance between all pairs in the given graph. Moreover, we used these shortest distances as the heuristic estimates for MLA* for path finding and IDA* for scheduling. In other words

$$\text{heuristic}([\{i1, j1\}], [\{i2, j2\}]) = \text{FloydWarshallShortestDistance}([\{i1, j1\}], [\{i2, j2\}]) \quad -[1]$$

$$h(v1, v2) = (M[v1][v2]) \quad -[2]$$

As opposed to the conventional Manhattan distance used as a heuristic estimate, we use the Floyd Warshall distance as a heuristic and show that it is a better underestimate, i.e. it is more closer to the actual value than a naive Manhattan approach. We explain the heuristic functions used in more detail in section 2.2.3.

```
Create a |V| x |V| matrix, M, that will describe the distances between vertices
For each cell (i, j) in M:
    if i == j:
        M[i][j] = 0
    if (i, j) is an edge in E:
        M[i][j] = Weight(i, j)    // in our case uniform weight of 1 – {normal} or 1e9 – {block}
    else:
        M[i][j] = 1e9;
For k from 1 to |V|:
    For i from 1 to |V|:
        For j from 1 to |V|:
            if M[i][j] > M[i][k] + M[k][j]:
                M[i][j] = M[i][k] + M[k][j]
```

Algorithm 1: Outline of Floyd Warshall Algorithm for APSP

2.3 Multi Label A*(MLA*)

Multi Label A* (MLA*) algorithm used for path finding of agents in this approach is inspired from [3]. Formally we use, the classic A* algorithm [7], with modifications and amendments to handle the multi-agent case, the details of which are explained below.

The scheduler function in 2.1 returns a sequential list of tasks for each robot along with the heuristic time taken for it to complete the tasks. Now, we independently apply the MLA* algorithm for each of these robots. For each search node n of the MLA* algorithm, we define g_n, p_n and l_n, which respectively denote the node's g-value (number of time steps elapsed), position(x,y), and label. The label indicates the current state of the node; if $l_n = 1$ the agent is seeking a path to the pickup location, if $l_n = 2$ the agent is seeking a path to the delivery location and if $l_n = 3$ the agent has delivered the final task allotted to it and is on its way back to the final position. We also define initP and finalP to be the pickup and delivery locations, respectively. Apart from this we maintain a 3-Dimensional array of dimension (M, N, Max Time of agent). This array maintains a bool datatype which denotes if a given cell in the warehouse is occupied or not at a given time.

We repeatedly call MLA* on each agent in descending order of heuristic time estimate for task completion of each agent. In other words, the agent with the largest estimate time for completion of task (pickup and delivery of each task allotted and return back to final position) gets an optimal path first. At each iteration of A*, we have 5 different options of next node, i.e. move left, right, up, down or stay at the same position. If the node stays at the same position, the same node with a g = g + 1 is pushed inside priority queue. After this repeated call, we get an optimal path for each agent without any collision or path overlap. The overview of MLA* algorithm is shown below.

For a given agent with a list of tasks that is allotted to it:

Do:

```
Maintain a priority_queue(min heap) pq that maintains list of open nodes
Maintain a list CLOSED that contains visited nodes along with their parent node id
While (pq is not empty do)
    Cur = pq.top()
    CLOSED.append(Cur)
    pq.pop()
    if l_n = 1 and p_n = initP then
        clear pq
        create node n0 with p_n0 = p_n, g_n0 = g_n and l_n0 = 2, and add n0 to Q;
    if l_n = 2 and pn = finalP then
        if no more task is left for agent then
            clear pq
            create node n0 with p_n0 = p_n, g_n0 = g_n and l_n0 = 3, and add n0 to Q;
        else
            clear pq
            create node n0 with p_n0 = p_n, g_n0 = g_n and l_n0 = 1, and add n0 to Q;
    else if l_n = 3 and p_n = agent.finalP
        return Found Path
    else
        For i = 1-> 5:          // 5 new nodes potentially
            if (new node is valid)
                pq.push(new node)
    end
return False (no Feasible path)
```

Algorithm 2: Outline of MLA* algorithm used for agent path finding

Algorithm 3 is the main function that calls the scheduler and also calls the MLA* function for each agent. Higher preference is given to agents with a higher estimated completion time, and their paths are fixed first. The algorithm is described below

```
Set the time step t = 0;
obtain agent – task schedule using scheduler explained in 2.1
sort agents in descending order of completion time
While(1)
    For each agent:
        use paths obtained from MLA* successfully
        update the botstatus, bot2task structure, botPos
        if an agent is at task.finalP then
            delete task, reinitialise bot2task with the next task and update task status
    store the complete frame for all bots and task locations in a list FRAME
    t = t + 1
end
```

Algorithm 3: Outline of the solver function that calls the scheduler and repeatedly calls MLA*

2.2.2 Multi label H value-based heuristic

In our A* algorithm, each cell or node in the warehouse map has the a label depending on the task that it is performing at a given time. The labels are 1- Agent going to pickup the task, 2- Agent has picked up the task and it is on its way to delivery and 3- Agent has delivered all its allocated tasks and is on it's way to the final position. Each of these labels have a different heuristic function. The label wise heuristic is similar to the one used in [3], with an additional label = 2, which adds the heuristic distance of the agent going back to its final position. The heuristics are computed as follows:

Table I. Shows the label wise Heuristic estimate used in this paper

Label	Heuristic estimate
1	h(agent.pos, task.initP) + h(task.initP, task.finalP) + h(task.finalP, agent.finalP)
2	h(agent.pos, task.finalP) + h(task.finalP, agent.finalP)
3	h(agent.pos, agent.finalP)

In label 2, once the agent reaches the tasks final position, there are 2 cases: 1) It is the final task of agent, hence a node with same position and the next time instant and label = 3 is pushed inside the queue, 2) The robot has more tasks after dropping the current one, hence a node with label = 1 is pushed inside the queue. As discussed, we use the pre-computed Floyd Warshall distance between any 2 points as h(point1, point2).

3. Results

We visualize the movement of agents with their task using OpenCV library. The compute and memory complexity of the algorithm is reasonable and generates all the frames withing a few seconds for agents in the order of ~10-20. The proposed algorithm is scalable for a denser grid with a greater number of agents and task too. The following images are retrieved from a simulation for a particular test case obtained at different time steps.

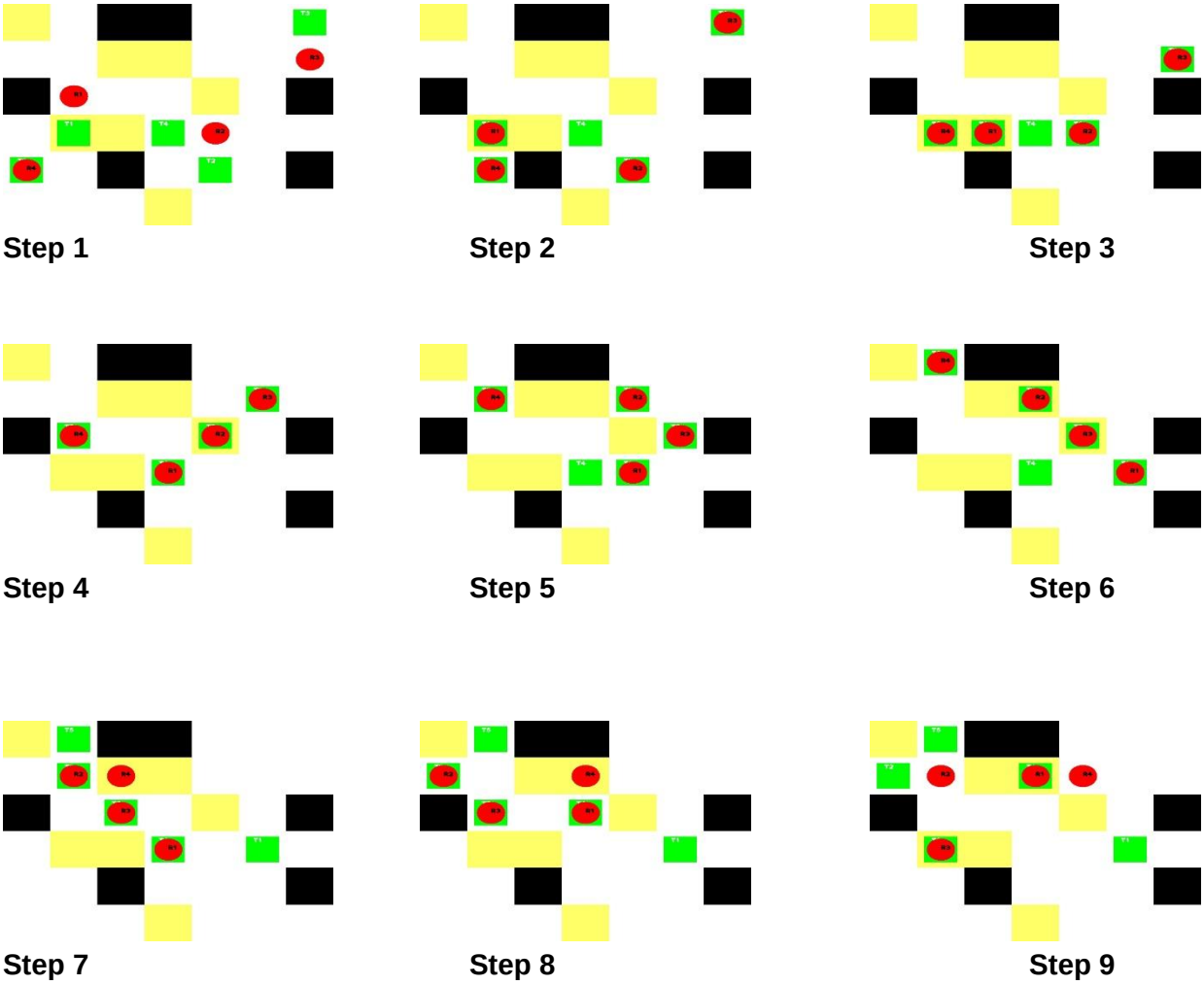

Fig. 3. A visual representation of the solution obtained using the proposed approach. Red- agent, Green- task, Black – blockage, Yellow- temporary storage

4. Conclusion

In this assignment we propose a novel algorithm for solving the Offline version of Multi Agent Pick-up and Delivery (MAPD). We leverage the fact that all tasks and agent locations (initial and final) are given to us, and we schedule the agent-task combinations optimally using IDA*. We then employ a modified MLA* algorithm that updates the path for each agent. We resolve collisions using heuristics and optimally provide unique paths to each agent in order to minimize total timespan. We use the Floyd Warshall algorithm to compute heuristics at each time

step of MLA* and show that Floyd Warshall is a good underestimate and a good heuristic for path finding algorithms. Out final simulation Figure 3 shows the working simulation of the proposed approach.

References:

[1] C. Mencía, M. R. Sierra, and R. Varela, "Intensified iterative deepening A* with application to job shop scheduling," *J. Intell. Manuf.*, vol. 25, no. 6, pp. 1245–1255, Dec. 2014, doi: 10.1007/s10845-012-0726-6.

[2] R. Risald, A. Mirino, and S. Suyoto, "Best routes selection using Dijkstra and Floyd-Warshall algorithm," Oct. 2017, pp. 155–158, doi: 10.1109/ICTS.2017.8265662.

[3] F. Grenouilleau, W.-J. van Hoeve, and J. N. Hooker, "A Multi-Label A* Algorithm for Multi-Agent Pathfinding," *Proc. Int. Conf. Autom. Plan. Sched.*, vol. 29, pp. 181–185, Jul. 2019.

[4] F. Kolushev and A. Bogdanov, "Multi-agent Optimal Path Planning for Mobile Robots in Environment with Obstacles," Jul. 1999, vol. 1755, pp. 503–510, doi: 10.1007/3-540-46562-6_45.

[5] H. Ma and S. Koenig, "AI buzzwords explained: multi-agent path finding (MAPF)," *AI Matters*, vol. 3, no. 3, pp. 15–19, Oct. 2017, doi: 10.1145/3137574.3137579.

[6] D. Ferguson, M. Likhachev, and A. Stentz, "A Guide to Heuristic-based Path Planning," p. 10.

[7] P. Mehta, H. Shah, S. Shukla, and S. Verma, "A Review on Algorithms for Pathfinding in Computer Games," Mar. 2015.