Multi-Agent Pickup and Delivery with Task Deadlines

Xiaohu Wu
Nanyang Technological University
Singapore
xiaohu.wu@ntu.edu.sg

Yihao Liu
Nanyang Technological University
Singapore
yihao002@ntu.edu.sg

Xueyan Tang
Nanyang Technological University
Singapore
asxytang@ntu.edu.sg

Wentong Cai
Nanyang Technological University
Singapore
aswtcai@ntu.edu.sg

Funing Bai
NCS Group
Singapore
eliza.bai@ncs.com.sg

Gilbert Khonstantine
Nanyang Technological University
Singapore
gkhonsta001@ntu.edu.sg

Guopeng Zhao
NCS Group
Singapore
leo.zhao@ncs.com.sg

ABSTRACT

We study the multi-agent pickup and delivery problem with task deadlines, where a team of agents execute a batch of tasks with individual deadlines to maximize the number of tasks completed by their deadlines. Existing approaches to multi-agent pickup and delivery typically address task assignment and path planning separately. We take an integrated approach that assigns and plans one task at a time taking into account the agent states resulting from all the previous task assignments and path planning. We define metrics to effectively determine which task is most worth assignment next and which agent ought to execute a given task, and propose a priority-based framework for joint task assignment and path planning. We leverage the bounding and pruning techniques in the proposed framework to greatly improve computational efficiency. We also refine the dummy path method for collision-free path planning. The effectiveness of the framework is validated by extensive experiments.

ACM Reference Format:
Xiaohu Wu, Yihao Liu, Xueyan Tang, Wentong Cai, Funing Bai, Gilbert Khonstantine, and Guopeng Zhao. 2022. Multi-Agent Pickup and Delivery with Task Deadlines. In *WI-IAT'21.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Multi-agent systems arise in many real-world applications, including warehouse management [13], aircraft towing [9] and mobile office service [11]. They are operated in a common environment and plan collision-free paths among agents, each continuously attending to tasks one by one. Each task is characterized by a pickup location and a delivery location. To execute a task, the agent has to move from its current location via the pickup location to the delivery location. This is known as Multi-Agent Pickup-and-Delivery (MAPD) [1, 4–7]. When planning paths for agents, some metric is to be optimized. Previous works have considered either makespan [5] or a common deadline for all tasks [8]; the latter is motivated by scenarios such as emergency evacuation where it is necessary to move as many agents as possible to a safe area before a disaster occurs.

In reality, there are also many scenarios where tasks have individual deadlines. Each task has to be completed (i.e., the agent executing the task arrives at the delivery location) by a specific deadline, in order to satisfy distinct customers with the delivered services/items in a timely manner. The tasks and their deadlines are often known a priori. For example, in the day-to-day operations of a warehouse, items have to be picked up from storage locations to inventory stations by specific deadlines so that they can become available for further processing. In an aircraft towing system, aircrafts need to be transported from the airport gates to the runway on time to ensure timely takeoff. In this paper, we study MAPD with task deadlines. Our objective is to maximize the number of tasks completed by their deadlines.

Real-time job scheduling has been studied extensively in computer systems [3]. In this paper, we enable the application of priority-based rules in real-time scheduling to the domain of MAPD. What complicates our problem is that it additionally involves path planning with collision-free requirements. As such, tasks have distinct completion times when executed by different agents, and the time needed by an agent to execute a task is not known beforehand in that it is dependent on the assignment and path planning of other concurrent tasks. We focus on two questions: (i) given a set of unassigned tasks, which task is most worth assignment next? and (ii) given a task, which agent should be used to execute it? We define proper metrics to address these questions and propose an effective priority-based framework for task assignment and path planning.

Related Work. Multi-Agent Path Finding (MAPF) is a classical problem that aims to find collision-free paths for a group of agents to

WI-IAT'21, December 2021, Melbourne, Australia

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

move from their current locations to their respective target locations with some metric optimized [10]. Deadlines have been considered in the MAPF problem where there is a common deadline for all agents and the objective is to maximize the number of agents that can reach their target locations by the deadline. This problem is NP-hard. Optimal solutions can be derived via search-based approaches or integer linear programming [8].

MAPD is an extension to the MAPF problem where a set of delivery tasks are to be assigned to the agents for execution. A MAPD solution needs to determine the tasks as well as their order to be executed by each agent and plan collision-free paths for the agents to complete their assigned tasks. Heuristic approaches have been proposed to optimize the makespan metric for MAPD [1, 4, 5]. These approaches typically consist of two separate phases. The first phase determines the task assignment without considering the potential conflicts among agents. The second phase plans collision-free paths for the agents to execute their tasks.

Liu et al. [5] construct a virtual complete graph among all tasks and agents and find a Hamiltonian cycle in the graph. The task sequence from one agent to the next agent along the cycle is assigned to the former. Two approaches are then developed for path planning. The first approach plans paths for the agents in a decreasing order of the estimated timesteps to complete their task sequences. The second approach improves the first one by allowing agents to swap their next tasks to be executed. Such swapping can optimize the costs based on the current states of the agents. Li et al. [4] assume that there is a task assigner that is independent of the path planner and continuously assigns tasks to agents. They propose a windowed scheme to replan paths once every fixed number of timesteps. Farinelli et al. [1] apply the token-passing scheme where agents take actions in the same cyclic order in the two phases. First, agents take turn to greedily get one task at a time. For each agent, the order of acquiring tasks is also the order of executing tasks. Second, each agent plans its own collision-free path based on the paths that have been planned for the other agents so far. The above approaches do not consider any deadline requirements and cannot be directly applied to our problem where the tasks have deadlines to meet.

Contributions. In this paper, we adopt an integrated approach that conducts task assignment and path planning together. In each task assignment, a favorable agent is chosen to execute the next task that is currently the most urgent according to the paths already planned for the tasks previously assigned. We define a metric called the flexibility of a task as the task deadline minus the earliest possible completion time among all the agents to execute the task. This metric allows us to effectively determine which task is most worth assignment next. Based on this metric, we propose a priority-based framework for joint task assignment and path planning. Its effectiveness is verified through experimental evaluations.

In our approach, after the assignment and path planning of every task, the states of the agents change. Thus, a key challenge of implementation is to compute the flexibility values of the unassigned tasks at every assignment based on the current states of the agents. We leverage the bounding and pruning techniques in our framework to greatly improve computational efficiency. A state-of-the-art method to avoid collisions in path planning is to reserve for every agent a dummy path from the agent's current location to its parking location whenever the agent finishes one task [5]. This may involve plenty of extra vain computation of paths that the agents will never use. In this paper, we improve this method by identifying the conditions under which planning such dummy paths is necessary and selectively reserving dummy paths for agents.

A brief introduction and some preliminary results of our work were presented as a 2-page extended abstract [12].

2 PROBLEM DEFINITION

Consider an undirected connected graph $\mathcal{G} = (V, E)$ where the nodes in V correspond to locations and each edge in E corresponds to a connection between two locations along which agents can move. There are a set of M agents $\mathcal{A} = \{a_1, \ldots, a_M\}$, and a set of N tasks $\mathcal{T} = \{t_1, \ldots, t_N\}$. All tasks are available at timestep 0. Each task t_j has a pickup location $s_j \in V$, a delivery location $g_j \in V$ and a deadline d_j. To execute a task t_j, an agent has to move from its current location via the pickup location s_j to the delivery location g_j. Each agent has a unit carrying capacity and can execute only one task at a time. Each agent a_i has a unique parking location $p_i \in V$ where it initially stays at timestep 0 and it can exclusively access at any time. After an agent completes all its tasks, it returns to its parking location. We would like to assign tasks to agents and plan paths for agents to execute them. Our objective is to maximize the number of tasks completed by their deadlines. Between two consecutive timesteps, an agent can execute either a move action to go to an adjacent location or a wait action to stay at its current location. Collisions may occur among agents at a location or along an edge. To avoid collisions, the following constraints are imposed in path planning: (i) two agents cannot occupy the same location at the same timestep, and (ii) two agents cannot traverse the same edge in opposite directions between the same two consecutive timesteps. We refer to our problem as Multi-Agent Pickup and Delivery with Task Deadlines (MAPD-TD).

A solution to the MAPD-TD problem specifies, for every agent $a_i \in \mathcal{A}$, (i) a sequence of tasks to be executed by a_i, and (ii) a path $\mathcal{P}_i$ along which a_i visits the pickup and delivery locations of its assigned tasks in sequence and finally returns to its parking location. The task sequences assigned to different agents are disjoint. The paths of different agents do not collide with each other. Finding the optimal MAPD-TD solution is computationally expensive since it involves (a) searching all possible partitions of the tasks $\mathcal{T}$ into task sequences among the agents $\mathcal{A}$, and (b) planning the globally optimal paths for the agents to execute their tasks. Hence, we focus on developing heuristic solutions for MAPD-TD.

Existing studies on MAPD problems often focus on a class of solvable instances known as well-formed instances [5, 7]. The pickup, delivery and parking locations are referred to as endpoints. A MAPD instance is well-formed iff (1) the number of tasks is finite, (2) the parking location of each agent is different from all task pickup and delivery locations, and (3) there exists a path between any two endpoints that traverses no other endpoints [5]. In such instances, agents can always stay in their parking locations for long enough periods to avoid collisions with other agents. Well-formed instances are typical for many real-world applications such as automated warehouses. Thus, we also focus on well-formed instances.

3 ALGORITHMS FOR MAPD-TD

We propose a priority-based framework to perform task assignment and path planning in an integrated manner. Each task assignment decision is made based on the paths already planned for the tasks previously assigned. Once a task gets assigned, the path for executing the task is planned immediately. We start by defining metrics to decide which task and which agent to choose for an assignment.

3.1 Choosing An Agent

Suppose an agent a_i has been assigned a sequence of tasks and according to the planned path for a_i, it takes τ_i timesteps to complete these tasks. That is, denoting by u_i the delivery location of a_i's last assigned task, a_i arrives at u_i at timestep τ_i and then a_i becomes available for executing new tasks. Given an unassigned task t_j, we can compute an optimal path, using A* search, for a_i to execute task t_j starting from timestep τ_i and finish it fastest. Let $c_{i,j}$ denote the timestep at which t_j is completed using the optimal path. Then, the cost of the optimal path, i.e., the number of timesteps required to execute t_j, is given by $c_{i,j} - \tau_i$.

Agents differ in the timesteps when they become available and the locations where they become available. Thus, they can have different costs to complete an unassigned task t_j. To improve the resource efficiency, among all the agents that can complete task t_j by its deadline d_j, we choose the agent a_{i^*} that has the lowest cost to execute t_j:

$$i^* = \underset{c_{i,j} \le d_i}{\operatorname{argmin}} (c_{i,j} - \tau_i). \tag{1}$$

3.2 Choosing A Task

To decide which task to assign next, we define a metric called the *flexibility*. The flexibility f_j of a task t_j is given by the task deadline minus the earliest possible completion time among all the agents to execute this task:

$$f_j = d_j - \min_{a_i \in \mathcal{A}} c_{i,j}. \tag{2}$$

The flexibility metric measures the urgency of the task. A lower flexibility value indicates that there is less time buffer and the task is more urgent to execute. If a task has a negative flexibility, it suggests that the task deadline cannot be met no matter which agent is assigned to execute the task.

Among all the unassigned tasks with non-negative flexibility values, we choose the task t_{j^*} with the lowest flexibility value to assign next:

$$j^* = \underset{f_j \ge 0}{\operatorname{argmin}} f_j. \tag{3}$$

3.3 Prioritized Task Assignment

Algorithm 1 shows our priority-based framework for joint task assignment and path planning, where $\mathcal{P}_i$ represents the planned path for each agent a_i to execute its assigned tasks, and τ_i and u_i represent the timestep and a_i's location at the end of $\mathcal{P}_i$. The high-level idea is as follows. Let $\mathcal{T}'$ denote the set of unassigned tasks. Initially, $\mathcal{T}' = \mathcal{T}$ (line 5). Tasks are considered and assigned to agents one at a time. In each task assignment, the algorithm first computes the completion time of executing each unassigned task by each agent (line 7) and then derives the flexibility of each task (line 8). After that, the algorithm chooses from $\mathcal{T}'$ the task t_{j^*} that satisfies (3) (lines 9-10) and assigns t_{j^*} to the agent a_{i^*} that satisfies (1) (line 11). Finally, the algorithm plans some dummy paths if needed (line 12, to be discussed in Section 3.5) and append the path for a_{i^*} executing t_{j^*} to a_{i^*}'s path $\mathcal{P}_{i^*}$ (lines 13-14). After the task assignment process is completed, the algorithm plans a path for each agent to return to its parking location (lines 15-16).

Algorithm 1: Integrated Task Assignment & Path Planning

```
1  for each agent a_i ∈ A do
2      τ_i ← 0;                          // the time when a_i is available
3      u_i ← p_i;                        // the location of a_i at time τ_i
4      P_i ← ∅;                          // the path for a_i
5  T' ← T;                               // the set of unassigned tasks
6  while T' ≠ ∅ do
7      For every pair (t_j, a_i) ∈ (T', A), plan the path and
         compute the completion time c_{i,j} of executing task t_j
         by agent a_i;
8      Compute the flexibility f_j of each task t_j ∈ T'
         according to (2);
9      Remove from T' all tasks t_j where f_j < 0;
10     Select task t_{j*} satisfying (3); remove it from T';
11     Assign t_{j*} to agent a_{i*} satisfying (1);
12     Dummy-Path-Planning(t_{j*}, a_{i*}) (Algorithm 3);
13     Append the path for a_{i*} executing t_{j*} to P_{i*};
14     τ_{i*} ← c_{i*,j*}; u_{i*} ← g_{j*};
15 for each agent a_i ∈ A do
16     Plan a path for a_i to move from u_i to the parking
         location p_i and append the path to P_i;
```

In Algorithm 1, line 7 involves planning a path for an agent to execute a task. We make use of the multi-label A* algorithm [2] to plan an optimal path for the agent to move from its current location via the task pickup location to the task delivery location. The A* search is conducted in the space of location-timestamp pairs taking into account the node and edge access constraints imposed by the paths $\{\mathcal{P}_i\}_{a_i \in \mathcal{A}}$ already planned for the previously assigned tasks.

3.4 Bounding and Pruning

Algorithm 1 computes an optimal path using A* search to derive the completion time $c_{i,j}$ for each pair of agents a_i and unassigned task t_j (line 7). In a naive implementation of Algorithm 1, the total number of A* calls is $O\left(MN^2\right)$ (where M and N are the numbers of agents and tasks respectively). To improve computational efficiency, we propose several bounding and pruning techniques to help discard agents and tasks that are unlikely to be chosen for assignment before A* searches are called or run to completion. Algorithm 2 summarizes the improved process for identifying t_{j^*} in each task assignment (in place of lines 7-10 of Algorithm 1).

Truncated A* Search. For each unassigned task t_j, we maintain its earliest completion time b_j among all the agents that have been examined. Initially, b_j is set to $d_j + 1$ (line 4, Algorithm 2), which implies that t_j cannot be completed by its deadline d_j if no agent is

Algorithm 2: Bounding and Pruning

```
1  B ← +∞;                                   // minimum flexibility among tasks
2  Sort all unassigned tasks t_j ∈ T' in increasing order of f_j
     from the previous task assignment;
3  for each task t_j ∈ T' do
4      b_j ← d_j + 1;                        // earliest completion time of t_j
5      Sort all agents a_i ∈ A in increasing order of c_{i,j} from
         the previous task assignment (or
         τ_i + d(u_i, s_j) + d(s_j, g_j) if c_{i,j} = ⊥ or a_i was selected
         for the previous task assignment);
6      if max_{a_i∈A} τ_i + min_{a_i∈A} d*(u_i, s_j) + d*(s_j, g_j)
         ≥ d_j − B then
7          for each agent a_i ∈ A do
8              if the path giving c_{i,j} in the previous task
                 assignment collides with the path planned for
                 executing the task selected in the previous task
                 assignment or c_{i,j} > d_j − B then
9                  c_{i,j} ← Truncated-A*(u_i, τ_i, t_j, b_j);
10             if c_{i,j} ≠ ⊥ then
11                 b_j ← c_{i,j};
12                 if c_{i,j} ≤ d_j − B then
13                     break;
14         f_j ← d_j − b_j;                  // flexibility of task t_j
15         if f_j < 0 then
16             remove t_j from T';
17         else
18             if f_j < B then
19                 B ← f_j;
20                 j* ← j;
```

going to execute it. After computing the completion time $c_{i,j}$ of t_j by an agent a_i, we update b_j by setting $b_j = c_{i,j}$ if $c_{i,j} < b_j$ (line 11).

The computation of each $c_{i,j}$ involves an A* search. The A* algorithm maintains an OPEN set recording the states to be searched. A score $f(n)$ is associated with each state n in OPEN, indicating the estimated timestep when task t_j can be completed if the path goes through n. In each iteration, a state n^* with the least score $f(n^*)$ is chosen and removed from OPEN. Then, this state is expanded by adding all its successor states to OPEN.

Normally, the A* search algorithm ends when a goal state is chosen from OPEN or the OPEN set is exhausted. To compute the flexibility f_j of a task t_j, we are interested in only the earliest completion time of t_j among all the agents. Thus, b_j can be added as an input to the A* algorithm so that A* can stop earlier, avoiding exploring unnecessary states. We adapt the A* algorithm by also terminating the search when the state n^* chosen from OPEN satisfies $f(n^*) > b_j$. When a state n^* satisfying $f(n^*) > b_j$ is chosen, it implies that all the states satisfying $f(n) \leq b_j$ have been examined. If no goal state was found, the agent cannot complete the task by time b_j and it is safe to stop searching any further states (in this case, we let the A* algorithm return a special value $\perp$). The adapted A* algorithm is referred to as Truncated-A*(u_i, τ_i, t_j, b_j), where u_i and τ_i are the location and timestep when agent a_i becomes available, t_j is the unassigned task to execute, and b_j is the upper bound on the completion time. If the A* algorithm ends with a goal state found, it implies that t_j has an earlier completion time than b_j by agent a_i. Then, we set $b_j = c_{i,j}$ and use this new bound in the A* call for the next agent (lines 9–11).

Note that in our solution, tasks are assigned one at a time. After a task is assigned, a path to execute the task is planned and appended to the designated agent, while the paths of all the other agents would not change. Thus, we do not expect significant changes to the node and edge access constraints for planning new paths between successive task assignments. The completion times of a task t_j by different agents in the previous task assignment can be good references for the next task assignment if t_j was not selected for assignment. To maximize the effectiveness of the bound b_j, we sort all the agents by their completion times for t_j derived in the previous task assignment (line 5). The agents are examined in the increasing order of these completion times. In this way, agents likely to achieve earlier completion times in the next task assignment are examined first to produce a tighter bound b_j. For the agents where the A* algorithm ended due to $f(n^*) > b_j$ in the previous task assignment, we do not have their completion times. In this case, we use $\tau_i + d(u_i, s_j) + d(s_j, g_j)$ as an alternative reference for the sorting purpose, where u_i and τ_i are the location and timestep when agent a_i becomes available, and $d(x, y)$ is the shortest-path distance between two locations x and y. Note that in the first task assignment where $\tau_i = 0$ for all agents, the agents are simply sorted according to $d(p_i, s_j) + d(s_j, g_j)$, where p_i is the parking location of agent a_i. The shortest-path distances between all pairs of endpoints (pickup, delivery and parking locations) can be precomputed before the task assignment process.

Pruning Across Tasks. Recall that in each task assignment, we would like to find the task with the least flexibility. Thus, we also apply a pruning technique across tasks. Suppose the flexibility f_k of a task t_k has been computed. When examining another task t_j, if we find that a particular agent a_i can achieve a completion time $c_{i,j}$ satisfying $d_j - c_{i,j} > f_k$, it implies that the flexibility f_j of task t_j satisfies $f_j = d_j - \min_{a_i \in \mathcal{A}} c_{i,j} \geq d_j - c_{i,j} > f_k$. Thus, t_j cannot be the task with the least flexibility. Hence, we can skip the path computations of the remaining agents for task t_j. To implement this idea, we maintain the least flexibility of all the tasks that have been examined, denoted by B (lines 1, 18–19). For each unassigned task t_j, we stop examining the agents once an agent is found to achieve a completion time earlier than $d_j - B$ (lines 12–13). To maximize the effectiveness of the bound B, we also sort all the unassigned tasks by their flexibility values in the previous task assignment (line 2). The tasks are examined in the increasing order of these flexibility values. As a result, tasks likely to have lower flexibility values are examined first to produce a tighter bound B. In the first task assignment, the tasks can be arranged in any order.

Even with the above bound B, for each unassigned task, we still need to compute the completion time by at least one agent. To further enhance computational efficiency, we establish an upper bound on the completion time of each unassigned task without any

A* call. Note that τ_i is the timestep when agent a_i becomes available. If we do not assign any new task to each agent when it becomes available, then by time $\max_{a_i \in \mathcal{A}} \tau_i$, all the agents would be available. At this time, if we would like to assign a task t_j to an agent and complete the task fastest, we should assign t_j to the agent with the shortest distance of $d^*(u_i, s_j) + d^*(s_j, g_j)$, where u_i is the location where agent a_i becomes available, and $d^*(x, y)$ is the shortest-path distance between two endpoints x and y without traversing any other endpoints (recall that in well-formed instances, there exists at least one such path). Thus, $\max_{a_i \in \mathcal{A}} \tau_i + \min_{a_i \in \mathcal{A}} d^*(u_i, s_j) + d^*(s_j, g_j)$ gives an upper bound on the completion time of task t_j. If this upper bound is earlier than $d_j - B$, we can safely discard task t_j from consideration without evaluating its completion time by any agent (line 6).

Reusing Task Completion Times. As mentioned, the changes to the node and edge access constraints for planning new paths between successive task assignments are usually minor. Thus, we can also skip recomputing the paths for unassigned tasks by reusing the computed paths from the previous task assignment. Specifically, if the path computed for an agent a_i to execute an unassigned task t_j does not collide with the path to execute the task selected in the previous task assignment, the path and hence the completion time $c_{i,j}$ for a_i to execute t_j remain valid in the next task assignment. Thus, $c_{i,j}$ from the previous task assignment can serve as an upper bound on the completion time of t_j in the next task assignment. Similar to "Pruning Across Tasks" above, if $c_{i,j}$ from the previous task assignment is earlier than $d_j - B$, there is no need to recompute the completion time for a_i to execute t_j. Therefore, we recompute the completion time for a_i to execute t_j only when the path giving $c_{i,j}$ in the previous task assignment collides with the path to execute the task selected in the previous task assignment or $c_{i,j} > d_j - B$ (line 8).

3.5 Collision-free Path Planning

Now we detail line 12 of Algorithm 1. Collisions may have to arise if the path planning of each newly assigned task is simply based on the access constraints formed by the paths already planned for the previously assigned tasks and no additional constraints are taken into account.

Example. Figure 1 (left) shows a graph with some locations marked. Consider the following case in Figure 1 (middle):

- In its planned path, agent a_1 arrives at location v_4 at timestep 4 upon completing its last assigned task. If no additional constraints are imposed, this location v_4 is forbidden to be accessed by other agents only at timestep 4.
- Subsequently, agents a_2 and a_3 have their new paths planned. In their planned paths, a_2 starts to move from timestep 2 along the path $v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4$, and a_3 starts to move from timestep 3 along the path $v_7 \rightarrow v_6 \rightarrow v_5$.

In this case, between timesteps 4 and 5, a_1 cannot move to v_3 and v_5 due to collisions with the paths of a_2 and a_3 (a_2 is moving from v_3 to v_4, and a_3 is moving from v_6 to v_5). In addition, if a_1 continues to stay at v_4 at timestep 5, a collision occurs with a_2. As a result, a collision is inevitable.

To avoid the collision above, some additional access permissions must be granted to a_1 after it completes the last assigned task, which correspond to additional access constraints for other agents. One possible method is to grant a_1 access to location v_4 infinitely after timestep 4, before a new path is planned for a_1. This, however, would imply that no agent can move from the left half of the graph to the right half, which can significantly degrade the efficiency of following task assignments and execution.

The state-of-the-art method reserves a dummy path for every agent $a_i \in \mathcal{A}$ whenever a_i gets assigned a new task [5]. Suppose an agent a_i gets assigned a new task t_j and will arrive at t_j's delivery location g_j at timestep $c_{i,j}$ according to the planned path. *A dummy path* is defined as a path for a_i to move from g_j to its parking location p_i starting from timestep $c_{i,j}$, denoted by $\mathcal{P}(c_{i,j}, g_j, p_i)$.

In the example of Figure 1, after planning the path for a_1 to arrive at v_4 at timestep 4, we can immediately plan a dummy path $v_4 \rightarrow v_5 \rightarrow v_6 \rightarrow v_8 \rightarrow p_1$ for a_1 to return to its parking location p_1 starting from timestep 4, illustrated by the orange line in Figure 1 (right). We reserve this dummy path for a_1 until a_1 is assigned the next task, and impose this dummy path as access constraints for other agents such as a_2 and a_3. Then, in the subsequent path planning, the movement of a_3 from v_6 to v_5 between timesteps 4 and 5 will not be allowed due to the collision with a_1's dummy path. As a result, a_3 can only access v_6 after a_1's dummy path.

If every agent is reserved a dummy path after completing every task, it can bring about plenty of access constraints on the nodes and edges, degrading the performance of path planning for new task execution. In the above example, collisions occur because (i) a_1 cannot visit all its adjacent nodes (v_3 and v_5) at a timestep after reaching location v_4; and (ii) there exists another agent a_2 to visit v_4 at that timestep. In the following, we refine the dummy path method by identifying the conditions under which reserving a dummy path is necessary.

A Refined Approach to Reserving Dummy Paths. Suppose that a task t_{j^*} is being considered for assignment to agent a_{i^*} (line 12, Algorithm 1). Let $\mathcal{P}_{i^*,j^*}$ denote the path planned for a_{i^*} to execute t_{j^*}. Let c_{i^*,j^*} denote the timestep at which a_{i^*} arrives at the delivery location g_{j^*} of t_{j^*}. We identify some conflict-of-interest (COI) conditions between a_{i^*} and other agents. Recall that for each agent a_i, u_i records the delivery location of a_i's last assigned task and τ_i records the timestep when a_i arrives at u_i.

Definition 1. *While assigning t_{j^*} to a_{i^*}, a* conflict-of-interest (COI) condition *arises with another agent a_i if either of the following holds:*

(a) *The planned path of a_i will access g_{j^*} at a timestep later than c_{i^*,j^*}.*
(b) *The path $\mathcal{P}_{i^*,j^*}$ will access the last delivery location u_i of a_i at a timestep later than τ_i.*

When there is a COI condition, a dummy path is needed so that the assignment of t_{j^*} will not lead to collisions. Let $\mathcal{P}_i^d$ denote the dummy path reserved for each agent $a_i \in \mathcal{A}$. Initially, the dummy paths of all agents are empty. Algorithm 3 shows how dummy paths are generated or updated to cope with a COI condition:

- Under a type-(a) COI condition, after t_{j^*} is assigned, a_{i^*} can only stay at g_{j^*} for a limited number of timesteps. We generate a dummy path $\mathcal{P}(c_{i^*,j^*}, g_{j^*}, p_{i^*})$ for a_{i^*} to move from g_{j^*} to its parking location p_{i^*} starting from timestep c_{i^*,j^*}.

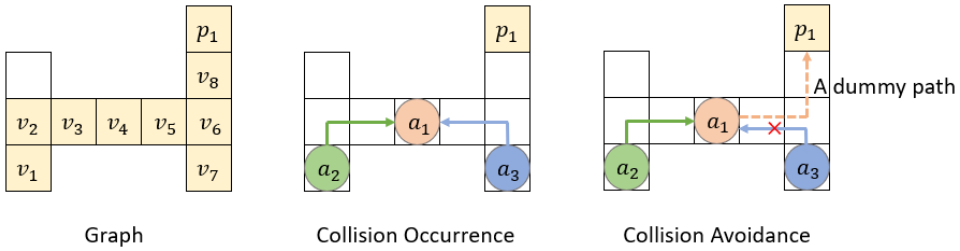

Figure 1: Necessity of dummy path.

Algorithm 3: Dummy-Path-Planning(t_{j^*}, a_{i^*})

1 Let $\mathcal{P}_{i^*,j^*}$ be the path planned for a_{i^*} to execute t_{j^*};
2 $\mathcal{P}^d_{i^*} \leftarrow \emptyset$;
3 **if** *a type-(a) COI condition holds* **then**
4 $\quad \mathcal{P}^d_{i^*} \leftarrow$ Plan a dummy path $\mathcal{P}(c_{i^*,j^*}, g_{j^*}, p_{i^*})$;
5 **foreach** *type-(b) COI condition holds* **do**
6 $\quad$ **if** $\mathcal{P}^d_i = \emptyset$ **then**
7 $\quad\quad \mathcal{P}^d_i \leftarrow$ Plan a dummy path $\mathcal{P}(\tau_i, u_i, p_i)$;

This new path overwrites the dummy path reserved for a_{i^*} before the assignment of t_{j^*} if any (lines 2–4, Algorithm 3).

- Under a type-(b) COI condition, after t_{j^*} is assigned, a_i can only stay at its last delivery location u_i for a limited number of timesteps. If no dummy path is reserved for a_i yet, we generate a dummy path $\mathcal{P}(\tau_i, u_i, p_i)$ for a_i to move from u_i to its parking location p_i starting from timestep τ_i. There may exist multiple type-(b) COI conditions for different agents a_i (lines 5–7, Algorithm 3).

Our proposed method reserves a dummy path for an agent only when a COI condition holds, and thus can greatly reduce the number of dummy paths needed.

Constraints and Path Planning. In general, when planning a path for an agent a_i, we need to respect the planned paths and the dummy paths (if any) of all the agents other than a_i, i.e., $\left\{\mathcal{P}_k \cup \mathcal{P}^d_k\right\}_{a_k \neq a_i}$. In line 4 of Algorithm 3, the path planning respects $\left\{\mathcal{P}_k \cup \mathcal{P}^d_k\right\}_{a_k \neq a_{i^*}}$. In line 7 of Algorithm 3, the path planning respects $\left\{\mathcal{P}_k \cup \mathcal{P}^d_k\right\}_{a_k \neq a_i}$. In case any planning of the dummy path fails in Algorithm 3, we skip the agent a_{i^*}, revert any changes made in the algorithm, and try assigning task t_{j^*} to the next agent satisfying (1). This continues until the path planning in Algorithm 3 succeeds. If all the agents are exhausted for t_{j^*}, we remove t_{j^*} from $\mathcal{T}'$.

Proposition 1. *The proposed method for reserving dummy paths guarantees that the planned paths $\mathcal{P}_1, \cdots, \mathcal{P}_M$ of all agents are collision-free when Algorithm 1 ends.*

Proof. While assigning t_{j^*} to a_{i^*}, the path $\mathcal{P}_{i^*,j^*}$ is planned based on the current constraints of $\left\{\mathcal{P}_k \cup \mathcal{P}^d_k\right\}_{a_k \neq a_{i^*}}$. Thus, $\mathcal{P}_{i^*,j^*}$ will not collide with the paths of other agents planned before t_{j^*}. After the assignment of t_{j^*}, when any other agent $a_{i^\diamond}$ gets assigned a new task $t_{j^\diamond}$, the path $\mathcal{P}_{i^\diamond,j^\diamond}$ for $t_{j^\diamond}$ is planned based on the constraints of $\left\{\mathcal{P}_k \cup \mathcal{P}^d_k\right\}_{a_k \neq a_{i^\diamond}}$ at that moment. Thus, collisions will not occur between $\mathcal{P}_{i^\diamond,j^\diamond}$ and $\mathcal{P}_{i^*,j^*}$. Therefore, the path $\mathcal{P}_{i^*,j^*}$ will not collide with any other agent, no matter whether the latter is executing a task assigned before or after t_{j^*}.

If t_{j^*} is the last task of a_{i^*}, we need to see whether there exists a collision-free path for a_{i^*} to move from g_{j^*} to the parking location p_{i^*}. If a type-(a) COI condition holds when assigning t_{j^*} to a_{i^*}, by Algorithm 3, a dummy path has been planned for a_{i^*} to move to p_{i^*}. Otherwise, if no other agent will access the delivery location g_{j^*} of t_{j^*} after timestep c_{i^*,j^*}, a_{i^*} can stay at g_{j^*} infinitely from timestep c_{i^*,j^*} onward. Thus, a collision-free path from g_{j^*} to p_{i^*} can always be found, e.g., a_{i^*} can wait at g_{j^*} until the other agents complete all their tasks and then a_{i^*} can find an endpoint-free path to p_{i^*} due to well-formed instances. □

4 EXPERIMENTAL RESULTS

To experimentally evaluate the proposed framework, we make use of two simulated warehouse environments of different sizes [5] as shown in Figure 2. We assume there are M agents and N tasks. The parking locations of agents are randomly chosen from the orange cells. The pickup and delivery locations of each task are randomly chosen from the blue cells.

To generate task deadlines, we run a simple algorithm to construct M hypothetical streams of tasks. Each task stream is headed by the parking location of one agent. We scan all the tasks and append each task to the stream with the least load, where the load of a hypothetical task stream is defined as the time required by an agent to complete all the tasks in sequence assuming that there is no conflict in the environment. That is, if a task stream is headed by a parking location p_i and has a sequence of tasks (s_1, g_1), (s_2, g_2), $\ldots$, (s_m, g_m), its load is given by $d(p_i, s_1) + d(s_1, g_1) + d(g_1, s_2) + d(s_2, g_2) + \cdots + d(g_{m-1}, s_m) + d(s_m, g_m)$, where $d(x, y)$ is the shortest-path distance between two locations x and y. Then, we set the deadline of each task (s_j, g_j) in the stream to $(1 + \phi) \cdot (d(p_i, s_1) + d(s_1, g_1) + d(g_1, s_2) + d(s_2, g_2) + \cdots + d(g_{j-1}, s_j) + d(s_j, g_j))$, where ϕ is a parameter for controlling the tightness of the deadline setting. If $\phi = 0$, it implies that a_i can just complete all the tasks in the stream by their deadlines in the ideal case that there is no conflict with any other agent. The larger the value of ϕ, the looser the task deadlines. Note that the construction of hypothetical task streams is for the purpose of setting task deadlines only. Agent a_i is not necessarily assigned the tasks in the stream headed by p_i by our assignment and planning algorithms. The above method for setting

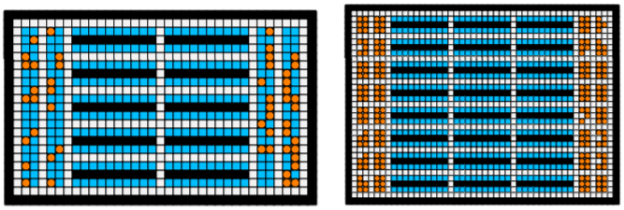

Figure 2: Two simulated warehouses of 4-neighbor grids [5]. Black cells are blocked, blue cells represent potential pickup or delivery locations of tasks, and orange cells represent potential parking locations of agents.

Table 1: Average success rate

small warehouse						
ϕ	N \ M	10	20	30	40	50
0.0	$10 \times M$	0.9360	0.9230	0.8573	0.8190	0.7978
0.1	$10 \times M$	0.9660	0.9635	0.9560	0.9342	0.8964
0.25	$10 \times M$	0.9950	0.9920	0.9950	0.9900	0.9880
large warehouse						
ϕ	N \ M	60	90	120	150	180
0.0	$10 \times M$	0.8778	0.8172	0.7737	0.7179	0.6836
0.1	$10 \times M$	0.9708	0.9198	0.8576	0.8141	0.7578
0.25	$10 \times M$	0.9977	0.9943	0.9842	0.9621	0.8948

task deadlines allows us to tune the parameter ϕ and construct problem instances in which it is possible to meet nearly all the task deadlines, which we believe is of most interest in practice.

We run extensive experiments with different settings. We set the number of agents $M = 10, 20, 30, 40, 50$ for the small warehouse and set $M = 60, 90, 120, 150, 180$ for the large warehouse. We set the number of tasks $N = 10 \times M$ and $\phi = 0, 0.1, 0.25$. For each setting of (M, N, ϕ), we randomly generate 10 problem instances and present the average performance over these instances. We implement the algorithms in C++ and run the experiments on a machine with Intel Core i9-9820X 3.30GHz CPU and 64GB memory.

Success Rate. Table 1 shows the average success rate of the 10 problem instances for each (M, N, ϕ) setting by our algorithms, where the success rate is defined as the ratio of the number of tasks completed by their deadlines to the total number of tasks. As expected, the tightness of the deadline setting is a main factor that affects the success rate. The success rate achieved by our algorithms increases with the ϕ value. It can also be seen that the success rate decreases with increasing number of agents M. This is because a larger number of agents give rise to potentially more conflicts in a given environment, making it harder to meet task deadlines. For comparison purposes, Table 2 shows the average success rate of a baseline algorithm that, for each task to assign, simply chooses the agent giving rise to the task flexibility (2) (i.e., completing the task earliest) rather than the agent with the lowest cost to execute the task (as given by (1)). It can be seen that choosing the agent with the lowest cost can improve the success rate by up to 10%.

Table 2: Average success rate for the baseline algorithm

small warehouse						
ϕ	N \ M	10	20	30	40	50
0.0	$10 \times M$	0.8670	0.8090	0.7760	0.7378	0.7216
0.1	$10 \times M$	0.9380	0.8920	0.8580	0.8198	0.7906
0.25	$10 \times M$	0.9890	0.9810	0.9630	0.9402	0.9116
large warehouse						
ϕ	N \ M	60	90	120	150	180
0.0	$10 \times M$	0.7867	0.7376	0.7022	0.6645	0.6271
0.1	$10 \times M$	0.8742	0.8124	0.7725	0.7261	0.6886
0.25	$10 \times M$	0.9792	0.9377	0.8933	0.8369	0.7918

Table 3: Average running time in seconds

small warehouse						
ϕ	N \ M	10	20	30	40	50
0.0	$10 \times M$	0.2050	0.9618	1.936	4.053	7.628
0.1	$10 \times M$	0.2160	1.207	2.722	5.836	10.25
0.25	$10 \times M$	0.2492	1.315	3.184	7.530	14.24
large warehouse						
ϕ	N \ M	60	90	120	150	180
0.0	$10 \times M$	15.50	42.68	104.2	221.3	398.1
0.1	$10 \times M$	20.82	53.65	117.0	238.5	424.7
0.25	$10 \times M$	20.00	69.89	180.1	374.0	634.1

Table 4: Speedup of bounding and pruning for small warehouse

ϕ	N \ M	10	20	30	40	50
0.0	$10 \times M$	33.90	79.19	150.6	202.8	235.5
0.1	$10 \times M$	35.96	74.70	138.6	170.4	224.5
0.25	$10 \times M$	36.57	93.07	181.8	214.0	272.0

Running Time. Table 3 shows the average running time of our algorithms for each (M, N, ϕ) setting. As can be seen, the running time increases with the number of agents M, since path planning requires more effort to resolve conflicts among agents. Our algorithms can handle the largest instances tested (180 agents and 1800 tasks for the large warehouse) in about 10 minutes. Next, we study the efficiency improvements due to the bounding and pruning techniques and the refined approach for dummy path reservation.

Bounding and Pruning. We also run our framework without applying the bounding and pruning techniques of Algorithm 2. Table 4 shows the average speedup due to bounding and pruning for each (M, N, ϕ) setting for the small warehouse. As can be seen, the proposed bounding and pruning techniques can speed

Table 5: Reserve ratio of refined dummy path reservation

small warehouse						
ϕ	N \ M	10	20	30	40	50
0.0	$10 \times M$	0.1452	0.2156	0.2435	0.3334	0.3998
0.1	$10 \times M$	0.1513	0.2574	0.3052	0.3640	0.4289
0.25	$10 \times M$	0.2161	0.3120	0.3679	0.4468	0.4990
large warehouse						
ϕ	N \ M	60	90	120	150	180
0.0	$10 \times M$	0.2815	0.3915	0.5129	0.5907	0.6486
0.1	$10 \times M$	0.3313	0.4214	0.5148	0.6050	0.6645
0.25	$10 \times M$	0.3961	0.4749	0.5697	0.6330	0.7088

Table 6: Speedup of refined dummy path reservation

small warehouse						
ϕ	N \ M	10	20	30	40	50
0.0	$10 \times M$	1.490	1.490	1.664	1.605	1.446
0.1	$10 \times M$	1.475	1.328	1.539	1.363	1.342
0.25	$10 \times M$	1.387	1.218	1.307	1.168	1.062
large warehouse						
ϕ	N \ M	60	90	120	150	180
0.0	$10 \times M$	1.557	1.495	1.426	1.274	1.242
0.1	$10 \times M$	1.394	1.361	1.338	1.244	1.176
0.25	$10 \times M$	1.303	1.141	1.061	1.064	0.9919

up task assignment and planning by one to two orders of magnitude. The speedup generally increases with the number of agents. The efficiency improvement is even more significant for the large warehouse in that we are not able to complete the runs for most (M, N, ϕ) settings without applying bounding and pruning (and hence do not report the speedup here).

Refined Dummy Path Reservation. Recall that in the original dummy path method, one dummy path is reserved for every task planned. Table 5 shows the average ratio between the number of dummy paths reserved by our refined approach and the number of tasks planned. It can be seen that our refined approach can save a substantial amount of dummy path computation. The saving is larger when the number of agents is smaller since it is less likely for agents to access the delivery locations of tasks executed by other agents. To study the efficiency improvement brought by the refined approach to reserving dummy paths, we also run our framework by simply reserving a dummy path for every task planned. Table 6 shows the average speedup due to our refined approach. As can be seen, our refined approach can reduce the running time remarkably, which demonstrates the effectiveness of reserving dummy paths selectively.

5 CONCLUSIONS

We have adopted an integrated approach to develop a joint task assignment and path planning framework for the MAPD-TD problem. We have also proposed a number of techniques to enhance the computational efficiency of the framework. In this paper, we have focused on the offline MAPD-TD problem. One direction for future work is to extend the framework of this paper to address the online MAPD-TD problem. Our framework can also be extended to optimize other metrics such as makespan and sum-of-costs.

6 ACKNOWLEDGMENTS

This study is supported under the RIE2020 Industry Alignment Fund – Industry Collaboration Projects (IAF-ICP) Funding Initiative, as well as cash and in-kind contribution from Singapore Telecommunications Limited (Singtel), through Singtel Cognitive and Artificial Intelligence Lab for Enterprises (SCALE@NTU).

REFERENCES

[1] Alessandro Farinelli, Antonello Contini, and Davide Zorzi. 2020. Decentralized Task Assignment for Multi-item Pickup and Delivery in Logistic Scenarios. In *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. 1843–1845.

[2] Florian Grenouilleau, Willem-Jan van Hoeve, and John N. Hooker. 2019. A Multi-Label A* Algorithm for Multi-Agent Pathfinding. In *Proceedings of the 29th International Conference on Automated Planning and Scheduling (ICAPS)*. 181–185.

[3] David Karger, Cliff Stein, and Joel Wein. 2010. Scheduling Algorithms. In *Algorithms and theory of computation handbook: special topics and techniques*. Chapman & Hall/CRC, 20:1 – 20:34.

[4] Jiaoyang Li, Andrew Tinka, Scott Kiesel, Joseph W. Durham, T. K. Satish Kumar, and Sven Koenig. 2020. Lifelong Multi-Agent Path Finding in Large-Scale Warehouses. In *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. 1898–1900.

[5] Minghua Liu, Hang Ma, Jiaoyang Li, and Sven Koenig. 2019. Task and Path Planning for Multi-Agent Pickup and Delivery. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. 1152–1160.

[6] Hang Ma, Wolfgang Hönig, T. K. Satish Kumar, Nora Ayanian, and Sven Koenig. 2019. Lifelong Path Planning with Kinematic Constraints for Multi-Agent Pickup and Delivery. In *Proceedings of the 33th AAAI Conference on Artificial Intelligence (AAAI)*. 7651–7658.

[7] Hang Ma, Jiaoyang Li, T. K. Satish Kumar, and Sven Koenig. 2017. Lifelong Multi-Agent Path Finding for Online Pickup and Delivery Tasks. In *Proceedings of the 16th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. 837–845.

[8] Hang Ma, Glenn Wagner, Ariel Felner, Jiaoyang Li, T. K. Satish Kumar, and Sven Koenig. 2018. Multi-Agent Path Finding with Deadlines. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI)*. 417–423.

[9] R. Morris, C. S. Pasareanu, K. Luckow, W. Malik, H. Ma, T. K. S. Kumar, and S. Koenig. 2016. Planning, Scheduling and Monitoring for Airport Surface Operations. In *Proceedings of the AAAI Workshop on Planning for Hybrid Systems*.

[10] Roni Stern, Nathan R. Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne T. Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, T. K. Satish Kumar, Eli Boyarski, and Roman Bartak. 2019. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. In *Proceedings of the 12th Annual Symposium on Combinatorial Search (SoCS)*. 151–158.

[11] Manuela Veloso, Joydeep Biswas, Brian Coltin, and Stephanie Rosenthal. 2015. CoBots: Robust Symbiotic Autonomous Mobile Service Robots. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI)*. 4423–4429.

[12] Xiaohu Wu, Yihao Liu, Xueyan Tang, Wentong Cai, Funing Bai, Gilbert Khonstantine, and Guopeng Zhao. 2021. Multi-Agent Pickup and Delivery with Task Deadlines (Extended Abstract). In *Proceedings of the 14th Annual Symposium on Combinatorial Search (SoCS)*. 2 pages.

[13] Peter R. Wurman, Raffaello D'Andrea, and Mick Mountz. 2008. Coordinating Hundreds of Cooperative, Autonomous Vehicles in Warehouses. *AI Magazine* 29, 1 (2008), 9–9.