Multi-Agent Pickup and Delivery with Task Deadlines

Paper Number: 334

ABSTRACT

We study the Multi-Agent Pickup and Delivery (MAPD) problem with task deadlines, where a team of agents execute a batch of tasks with individual deadlines to maximize the throughput. Existing approaches to MAPD 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 states of agents resulting from all the previous task assignments and path planning. For this purpose, we define metrics that allow us to effectively determine which agent ought to execute a given task and which task is most worth execution next, and propose a priority-based framework for joint task assignment and path planning. In our approach, a major challenge is frequently calling A^* search to compute the costs of potential paths. We leverage the brand and bound technique in the proposed framework to greatly improve the computational efficiency. We also refine the dummy path method for collision-free path planning. The effectiveness of the framework is validated by extensive experiments.

KEYWORDS

Multi-agent pickup and delivery, task assignment, path planning

1 INTRODUCTION

Multi-agent systems arise in many real-world applications, including warehouse management [1], aircraft towing [2] and mobile office service [3]. 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 called Multi-Agent Pickup-and-Delivery (MAPD) in the literature [4–8]. When planning paths for agents, some metric is to be optimized. Previous works have considered either makespan [4] or a common deadline for all tasks [9]; 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 operation 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 resource efficiency or the throughput (i.e., the number of tasks completed by their deadlines) given the resource constraints [9].

Real-time job scheduling has been studied extensively in computer systems [10]. 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 a result, tasks have different 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 task, which agent should be used to execute it? and (ii) given a set of unassigned tasks, which task is worth execution next? 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) [12] is a classical problem that aims to find collision-free paths for a group of agents to move from their current locations to their respective target locations with some metric optimized. 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 [9].

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 execute 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 [4–6]. 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. [4] construct a virtual complete graph among all tasks and agents and find a Hamiltonian cycle in the graph. The task sequence between two agents along the cycle is assigned to the agent at one end. Next, two approaches are 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 approach 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. [6] 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 several timesteps. Farinelli et al. [5] 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

Proc. of the 20th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2021), U. Endriss, A. Nowé, F. Dignum, A. Lomuscio (eds.), May 3–7, 2021, London, UK.

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 all the agents to execute the tasks previously assigned. To meet the deadline requirement, the favorable agent may either be one taking less timesteps to complete the next task or one being lightly loaded such that it can become available earlier to execute the next task. 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 execution next. Based on this metric, we propose a flexibility-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. This involves frequently calling A^* search to compute the potential paths of the agents for new tasks. We leverage the brand and bound technique [13] in our framework to greatly improve the computational efficiency. A state-of-the-art method to avoid collisions in path planning is to reserve for every agent a dummy path that starts from the agent's current location to its parking location whenever the agent finishes one task [4]. 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, which can also give rise to a better computational efficiency.

Finally, we note that the MAPD problems share many features with the well-studied queueing problems for service science. The concept of "agents" in MAPD corresponds to the "servers" in queueing problems as both of them need to dedicate a specific time period to process each task. Both problems aim to use the available server/agents to finish tasks fast. To this end, it is desirable to avoid load imbalance in assigning tasks to servers/agents where some servers/agents need much longer time to complete their assigned tasks than other servers/agents. The token-passing scheme in [5] is in essence the static round-robin rule in queueing theory. Such static policies do not consider the states of the servers. In contrast, the priority-based scheme of this paper corresponds to dynamic policies in queueing theory such as Join-the-Shortest-Queue, which observe the states of the servers to improve the task completion times [15–17].

2 PROBLEM DEFINITION

Consider an undirected connected graph $\mathcal{G} = (V, E)$ where the nodes in V correspond to locations and an edge in E corresponds to a connection between two locations along which agents can move. There are a set of M agents $\mathcal{A} = \{a_1, \cdots, a_M\}$, and a set of N tasks $\mathcal{T} = \{t_1, \cdots, t_N\}$. 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 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 throughput, i.e., the number of tasks completed by their deadlines. At each timestep, an agent can execute either a move action to move 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 the path planning process: (i) two agents cannot occupy the same location at the same timestep, and (ii) two agents cannot traverse the same edge in opposite directions at the same timestep. 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 path $\mathcal{P}_i$ is a sequence of locations specifying where the agent is located at each timestep. The paths of different agents are collision-free. Finding the optimal MAPD-TD solution to maximize the number of tasks completed by their deadlines is computationally expensive since it involves (a) searching all possible partitions of the tasks $\mathcal{T}$ among the agents $\mathcal{A}$, (b) searching all possible permutations of the tasks assigned to each agent, and (c) 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 [4, 8]. 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 [4]. In such instances, agents can always stay in their parking locations for a long enough period 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 SOLUTION

In this section, we propose a priority-based framework to perform task assignment and path planning in an integrated manner. Each assignment decision is made based on the paths already planned for the agents. Once a task gets assigned, the path for executing the task is planned immediately.

3.1 Priority Definition

First, we define metrics to answer the following questions:

(i) Given a task, which agent should be used to execute it?
(ii) Given a set of unassigned tasks, which task is most worth execution next?

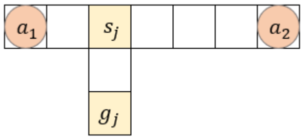

Figure 1: Choosing an agent according to its location and the timestep when it becomes available.

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, a_i arrives at the delivery location of the last assigned task at timestep τ_i and then a_i becomes available for executing other 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 $c_{i,j} - \tau_i$.

The agents differ in the timesteps when they become available and the locations where they become available. Thus, they can have different times and 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:

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

This is illustrated in Figure 1. The task t_j has a deadline $d_j = 10$. The timesteps required by agent a_1 to execute t_j is 4, and the timesteps required by agent a_2 to execute t_j is 6. In the case that both agents become available at the same timestep 3 (i.e., $\tau_1 = \tau_2 = 3$), agent a_1 will be chosen to execute t_j, since $c_{1,j} = 3 + 4 = 7$ and $c_{2,j} = 3 + 6 = 9$ are both less than d_j, and a_1 has a lower cost to execute t_j than a_2. In the case that agents a_1 and a_2 are available at timesteps 7 and 3 respectively (i.e., $\tau_1 = 7$ and $\tau_2 = 3$), agent a_2 will be chosen to execute t_j, since $c_{1,j} = 7 + 4 = 11$ is beyond d_j.

Choosing a Task. To decide which task to execute next, we define a metric called the *flexibility* for each task. 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. A higher flexibility value implies that there is more time buffer and the task is less 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 execute next:

$$j^* = \operatorname*{argmin}_{f_j \geq 0} f_j. \tag{3}$$

This rule is referred to as *Least Flexibility First* (LFF).

Algorithm 1: Integrated Task Assignment and Path Planning

1 $\mathcal{T}' \leftarrow \mathcal{T}$; // $\mathcal{T}'$ represents the set of unassigned tasks
2 **for** *each agent* $a_i \in \mathcal{A}$ **do**
3     $\tau_i \leftarrow 0$; // τ_i records the time when a_i is available
4     $u_i \leftarrow p_i$; // u_i records the location of a_i at time τ_i
5     $\mathcal{P}_i \leftarrow \emptyset$; // $\mathcal{P}_i$ records the path for a_i
6 **while** $\mathcal{T}' \neq \emptyset$ **do**
7   For every pair $(t_j, a_i) \in (\mathcal{T}', \mathcal{A})$, 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 $t_j \in \mathcal{T}'$ according to (2);
9   Remove from $\mathcal{T}'$ all tasks t_j where $f_j < 0$;
10   Select the task t_{j^*} satisfying (3); remove it from $\mathcal{T}'$;
11   Assign t_{j^*} to a_{i^*}, where a_{i^*} is the agent satisfying (1);
12   Plan a path $\mathcal{P}_{i^*,j^*}$ for a_{i^*} to execute t_{j^*} by calling Path-Planning(t_{j^*}, a_{i^*}) (presented in Algorithm 3);
13   Append $\mathcal{P}_{i^*,j^*}$ to $\mathcal{P}_{i^*}$;
14   Update τ_{i^*} and u_{i^*};
15 **for** *each agent* $a_i \in \mathcal{A}$ **do**
16   Plan a path for a_i to move from u_i to the parking location p_i and append the path to $\mathcal{P}_i$;

3.2 Prioritized Task Assignment

In this subsection, we base the task assignment process on the analysis in Section 3.1 and give a framework that assigns tasks to agents. We will later detail the path planning process at each task assignment.

The task assignment process is presented in the lines 1-14 of Algorithm 1. Its high-level idea is as follows. Let $\mathcal{T}'$ denote the set of unassigned tasks. Initially, $\mathcal{T}' = \mathcal{T}$ (line 1). Tasks are considered and assigned to agents one at a time. In each task assignment, the algorithm first computes the cost and completion time of executing each unassigned task by each agent (line 7) and then derive 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 a path for a_{i^*} to execute t_{j^*} (line 12) and append it to $\mathcal{P}_{i^*}$ (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).

In Algorithm 1, lines 7 and 12 involve planning a path for an agent to execute a task. We make use of the multi-label A* algorithm [11] 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.

3.3 Branch and Bound

Computational efficiency is an important consideration for MAPD solutions. In this subsection, we propose a realization of the branch and bound paradigm in our solution by adapting the A* search algorithm.

To make a task assignment decision, Algorithm 1 computes an optimal path using A* search to derive the completion time $c_{i,j}$ for each pair of agent a_i and unassigned task t_j (line 3). Then, the flexibility f_j of each task t_j is computed based on the earliest completion time among all agents, and the task with the least flexibility is chosen for assignment. 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) and these operations incur a high computational cost.

The branch-and-bound paradigm executes a systematic search of the candidate solutions [13]. The set of candidate solutions is expressed as a tree and the algorithm explores the branches of this tree, which represent subsets of the candidate solution set. There is a lower or upper estimated bound on the optimal solution. Before searching the candidate solutions in a branch, the branch is checked against the estimated bound to see whether the branch possibly contains a better solution; if not, the branch is discarded and we do not need to evaluate every individual candidate solution in this branch. The branch and bound paradigm can help discard sub-optimal candidate solutions before too many actions are taken for evaluating these solutions, thus reducing the computational time of an algorithm. We adopt this principle to improve the computational efficiency of our task assignment and path planning algorithm.

Given an unassigned task t_j, we denote by b_j as an upper bound of its completion time.

- Before computing the completion time of t_j by any agent, we can initially set b_j to $+\infty$, which implies that t_j cannot be completed if no agent is going to execute it.
- After each subsequent computation of the completion time $c_{i,j}$ of t_j by a particular agent a_i, we can improve the upper bound b_j by setting $b_j = c_{i,j}$, if $c_{i,j} < b_j$.

The computation of each $c_{i,j}$ involves a call to the A* search algorithm. The A* algorithm maintains two lists *CLOSED* and *OPEN* to record the states already searched and to be searched respectively. An estimated lower bound $f(n)$ of the complete time is associated with each state n in *CLOSED* or *OPEN* indicating the timestep when task t_j can be completed. Initially, *OPEN* contains only the start state, while *CLOSED* is empty. In each iteration, a state n^* with the least cost is chosen from *OPEN* and added to *CLOSED*. Then, this state is expanded by adding all its successor states *OPEN*. The search terminates when a goal state is chosen from *OPEN*.

Adapted A*. Normally, the A* search algorithm will exit when it reaches a goal state or all reachable states have been searched. 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 searching too many states. We add the following exit condition to the A* algorithm:

An additional exit condition. The A* algorithm will also exit when the chosen state n^* 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. Thus, 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$^*(u_i, \tau_i, t_j, b_j)$, 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 exits with a goal state found, it implies that t_j has an earlier completion time by the current agent a_i. Then, we set $b_j = c_{i,j}$ and use this new bound in the call to the A* algorithm for the next agent.

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 new exit condition, we further sort all the agents by their completion times for t_j derived in the previous task assignment. The agents are examined in the increasing order of these completion times. In this way, agents that are 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 exits due to the new condition above, 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, $d(u_i, s_j)$ is the shortest-path distance from u_j to the pickup location s_j, and $d(s_j, g_j)$ is the shortest-path distance from s_j to the delivery location g_j. The shortest-path distances between all pairs of endpoints (pickup, delivery and parking locations) can be precomputed before the task assignment and path planning process. In the first task assignment, all 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.

Recall that in each task assignment, we would like to find the task with the least flexibility. Thus, we can also apply the branch and bound paradigm 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. For each unassigned task t_j, We stop examining the agents once an agent is found to achieve a completion time earlier than $d_j - B$. To further enhance the computational efficiency, we also sort all the unassigned tasks by their flexibility values in the previous task assignment. The tasks are examined in the increasing order of these flexibility values. As a result, tasks that are 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 any order.

Algorithm 2 summarizes the improved process for identifying t_{j^*} in each task assignment (in place of lines 7-10 of Algorithm 1).

Algorithm 2: Branch and Bound

$B \leftarrow +\infty$;
Sort all unassigned tasks $t_j \in \mathcal{T}'$ in increasing order of f_j;
for *each task* $t_j \in \mathcal{T}'$ **do**
  $b_j \leftarrow +\infty$;
  Sort all agents $a_i \in \mathcal{A}$ in increasing order of $c_{i,j}$ (or $\tau_i + d(u_i, s_j) + d(s_j, g_j)$);
  for *each agent* $a_i \in \mathcal{A}$ **do**
    $c_{i,j} \leftarrow$ Truncated-A$^*(u_i, \tau_i, t_j, b_j)$;
    if $c_{i,j} \neq \perp$ **then**
      $b_j \leftarrow c_{i,j}$;
    if $c_{i,j} \leq d_j - B$ **then**
      break;
  $f_j \leftarrow d_j - b_j$;
  if $f_j < 0$ **then**
    remove t_j from $\mathcal{T}'$;
  else
    if $f_j < B$ **then**
      $B \leftarrow f_j$;
      $j^* \leftarrow j$;

3.4 Collision-free Path Planning

In this subsection, 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. We give an example to illustrate this. Figure 2 (left) shows a graph with some key locations marked. As illustrated in Figure 2 (middle), we consider the following case:

- In its planned path, agent a_1 arrives at location v_4 at timestep 4 upon completion of 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 at timestep 2 along the path $v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4$, and a_3 starts to move at timestep 3 along the path $v_7 \rightarrow v_6 \rightarrow v_5$.

In this case, at timestep 5, a_1 cannot move to v_3 and v_5 due to collisions with the paths of a_2 and a_3. In addition, if a_1 continues to stay at v_4 at timestep 5, a collision occurs with the agent 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 a path is planned to finish its last assigned task, which corresponds to additional access constraints for other agents. One possible method to grant a_1 access to location v_4 infinitely before a new path is planned for a_1 after timestep 4, i.e., to add an access constraint that forbids v_4 to be accessed by other agents infinitely. 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 assignment and execution. The state-of-the-art method associates a dummy path with every agent $a_i \in \mathcal{A}$ whenever a_i gets assigned a new task. Below, we formalize the concept of dummy paths that are first introduced in [4].

Definition 3.1. Suppose an agent $a_i \in \mathcal{A}$ 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 at timestep $c_{i,j}$, denoted by $\mathcal{P}(c_{i,j}, g_j, p_i)$.

In the example of Figure 2, after planning the path for a_1 to arrive at v_1 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 at timestep 4, illustrated by the blue arrow in Figure 2 (right). We reserve the access permissions to these locations at corresponding timesteps for agent a_1 until a_1 is assigned the next task, and thus impose this dummy path as access constraints on 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 at timestep 4 will not be allowed due to the collision with a_1's dummy path. In the future path planning for a_1, it can move along some or all nodes of the dummy path at the predefined timesteps or completely discard the dummy path.

A dummy path of an agent $a_i \in \mathcal{A}$ represents its exclusive access permission to some nodes at specific timesteps. If every agent is associated with 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 executions. In the above example, collisions occur only under the conditions that (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) that is to visit v_4 at that timestep. Collisions would not occur if any of these conditions is not satisfied. This implies that associating an agent with a dummy path is only necessary under limited conditions. In the following, we refine the method of [4] by identifying the conditions under which planning a dummy path is necessary.

A Refined Approach to Associating Dummy Paths. Suppose that a task t_{j^*} is being considered to be assigned to agent a_{i^*} in an assignment (line 12, Algorithm 1). The path for executing t_{j^*} by a_{i^*}, denoted by $\mathcal{P}_{i^*,j^*}$, is first computed. Let c_{i^*,j^*} denote the timestep at which a_{i^*} arrives at the delivery location g_{j^*} of t_{j^*}. At the delivery location g_{j^*}, we identify some critical temporal relationships of a_{i^*} to other agents, referred to as conflict-of-interest conditions.

Definition 3.2. While considering assigning t_{j^*} to a_{i^*}, we say that a *conflict-of-interest condition* arises if either of the following holds:

(a) There exists another agent, denoted by a_{i_1}, whose planned path will pass g_{j^*} at a timestep later than c_{i^*,j^*} (here g_{j^*} can be any node on the path of a_{i_1}).
(b) There exists another agent, denoted by a_{i_2}, such that g_{j^*} is also the delivery location of the last assigned task of a_{i_2}, and a_{i_2} will finish its last assigned task at a timestep earlier than c_{i^*,j^*}.

When there exists a conflict-of-interest condition, a dummy path may be needed so that the assignment of t_{j^*} will not lead to collisions. Let $\mathcal{P}_i^d$ denote the dummy path associated with each

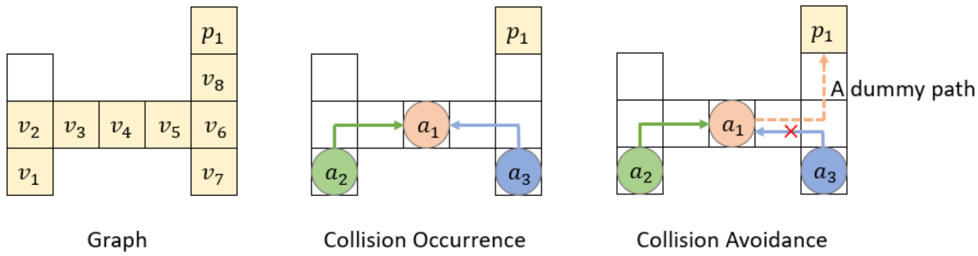

Figure 2: Limited Necessity of Dummy Paths.

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

/* t_{j^*} is assigned to a_{i^*} */

1 Use A* search to plan a path $\mathcal{P}_{i^*,j^*}$ for a_{i^*} to execute t_{j^*};

2 **if** *a conflict-of-interest condition of type (a) holds* **then**

3   Plan a dummy path $\mathcal{P}(c_{i^*,j^*}, g_{j^*}, p_{i^*})$;

4     $\mathcal{P}^d_{i^*} \leftarrow \mathcal{P}(c_{i^*,j^*}, g_{j^*}, p_{i^*})$;

5 **if** *a conflict-of-interest condition of type (b) holds* **then**

6   **if** $\mathcal{P}^d_{i_2} = \emptyset$ **then**

7   Plan a dummy path $\mathcal{P}(\tau_{i_2}, g_{j^*}, p_{i_2})$;

8         $\mathcal{P}^d_{i_2} \leftarrow \mathcal{P}(\tau_{i_2}, g_{j^*}, p_{i_2})$;

---

agent $a_i \in \mathcal{A}$. Initially, the dummy paths of all agents are empty. We present in Algorithm 3 how dummy paths are generated or updated to cope with a conflict-of-interest condition:

- Under a conflict-of-interest condition of type (a), after t_{j^*} is assigned, a_{i^*} can only stay at g_{j^*} for a limited number of timesteps, due to the existence of a_{i_1}. We generate a dummy path $\mathcal{P}(c_{i^*,j^*}, g_{j^*}, p_{i^*})$ for a_{i^*} to move from g_{j^*} to the parking location p_{i^*} starting at timestep c_{i^*,j^*}. This new path overwrites the dummy path associated with a_{i^*} before the assignment of t_{j^*} if any (lines 2-4, Algorithm 3).
- Under a conflict-of-interest condition of type (b), the assignment of t_{j^*} has an effect on agent a_{i_2} such that a_{i_2} can only stay at the delivery location $g_{j_{i_2}}$ of its last task for a limited number of timesteps. If there is no dummy path associated with a_{i_2} yet, we generate a dummy path $\mathcal{P}(\tau_{i_2}, g_{j^*}, p_{i_2})$ for a_{i_2} (lines 5-8, Algorithm 3).

The method above associates a dummy path with an agent only when a conflict-of-interest condition holds. In the other cases, no dummy paths are associated with agents. These cases include (i) no planned path for any task assigned before t_{j^*} will pass the delivery location g_{j^*} of t_{j^*}, and (ii) there exists another agent, denoted by a_{i_3}, such that a_{i_3} will pass g_{j^*} at a timestep earlier than c_{i^*,j^*}, and g_{j^*} is not the delivery location of a_{i_3}'s last assigned task, and . As a result, the proposed method can greatly reduce the number of dummy paths needed and thus improve the efficiency and performance of path planning.

Constraints and Path Planning. In general, When planning a path for an agent $a_i \in \mathcal{A}$, we need to respect the planned paths of all agents and the dummy paths of the agents other than a_i (if any), i.e., $\{\mathcal{P}_{i'}\}_{a_{i'} \in \mathcal{A}} \cup \left\{\mathcal{P}^d_{i'}\right\}_{a_{i'} \neq a_i}$. In this way, whenever any agent $a_i \in \mathcal{A}$ is associated with a dummy path, the path planning of other agents will still reserve the access permissions for a_i. As a result, we can always guarantee for any agent that, when it is assigned a new task in the future, there exists a feasible path for the agent to execute the task starting from its current location.

In lines 1 and 3 of Algorithm 3, the path planning respects $\{\mathcal{P}_{i'}\}_{a_{i'} \in \mathcal{A}} \cup \left\{\mathcal{P}^d_{i'}\right\}_{a_{i'} \neq a_{i^*}}$. In line 7 of Algorithm 3, the path planning respects $\{\mathcal{P}_{i'}\}_{a_{i'} \in \mathcal{A}} \cup \left\{\mathcal{P}^d_{i'}\right\}_{a_{i'} \neq a_{i_2}}$. In case any planning of the dummy path fails in Algorithm 3, we skip the agent a_{i^*} 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 3.3. *The proposed method for associating 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. It suffices to show that, while each agent $a_i \in \mathcal{A}$ moves along its planned path to sequentially complete its assigned tasks and return to its parking location, it will not collide with any other agent. This can be decomposed to analyze the cases where a_i executes each of its tasks or returns to its parking location. Without loss of generality, we will show for each assigned task t_{j^*} that, upon completion of Algorithm 1, while a_{i^*} is moving along the planned path $\mathcal{P}_{i^*,j^*}$, it will not collide with any other agent. If t_{j^*} is the last task of a_{i^*}, it will subsequently return to the parking location p_{i^*} and we will also show that no collision occurs in this process.

While assigning t_{j^*}, the planning of the path $\mathcal{P}_{i^*,j^*}$ is based on the current constraints of $\{\mathcal{P}_{i'}\}_{a_{i'} \in \mathcal{A}} \cup \left\{\mathcal{P}^d_{i'}\right\}_{a_{i'} \neq a_{i^*}}$. 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 $t_{j^\diamond}$, a subpath $\mathcal{P}_{i^\diamond,j^\diamond}$ for $t_{j^\diamond}$ is planned based on the constraints of $\{\mathcal{P}_{i'}\}_{a_{i'} \in \mathcal{A}} \cup \left\{\mathcal{P}^d_{i'}\right\}_{a_{i'} \neq a_{i^\diamond}}$ at that moment. Thus, collisions will not occur while $a_{i^\diamond}$ and a_{i^*} are moving along $\mathcal{P}_{i^\diamond,j^\diamond}$ and $\mathcal{P}_{i^*,j^*}$ respectively. Therefore, while a_{i^*} is moving along the planned path $\mathcal{P}_{i^*,j^*}$, it 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^*}, let c_{i^*,j^*} denote the timestep at which a_{i^*} arrives at the delivery location g_{j^*}. We need to observe the following cases to see whether there exists a collision-free path for a_{i^*} to move from g_{j^*} to the parking location p_{i^*}. The first case is that a conflict-of-interest condition of type (a) in Definition 3.2 holds. Then, from the above (lines 2-4, Algorithm 3), we know that a dummy path $\mathcal{P}(c_{i^*,j^*}, g_{j^*}, p_{i^*})$ for a_{i^*} would be planned. The second case is that no other tasks will pass the delivery location g_{j^*} of t_{j^*} after timestep c_{i^*,j^*}. Then, the agent a_{i^*} can stay at g_{j^*} infinitely from the timestep c_{i^*,j^*} onward. Thus, we can find a collision-free path from g_{j^*} to p_{i^*}, e.g., a_{i^*} can wait at g_{j^*} until the other agents complete all their tasks and then it can find a path to p_{i^*} due to well-formed instances. □

4 EXPERIMENTAL RESULTS

The performance of the proposed framework is experimentally evaluated in three aspects. The first aspect is the success rate, which is defined as the ratio of the number of tasks completed by their deadlines to the total number of tasks. The other two aspects are the computational time improvements brought respectively by the proposed branch and bound technique and the refined approach to associating dummy paths.

We make use of two typical simulated warehouse environments of different sizes [4, 8] as shown in Figure 3. We assume there are M agents and $N = k \cdot M$ tasks, where k is a parameter indicating the average number of tasks to execute by an agent. The agent parking locations and tasks are generated as follows. Given a setting of M and k, we generate M streams of locations:

(i) Each stream i contains $2k+1$ locations $o_{i,1}, o_{i,2}, \cdots, o_{i,2k+1}$. The first location $o_{i,1}$ is randomly chosen from the orange circles. The other $2k$ locations are randomly chosen from the blue cells.

(ii) $o_{i,1}$ is used as the parking location of agent a_i.

(iii) Each pair of successive locations $o_{i,2j}$ and $o_{i,2j+1}$ ($j \in [1,k]$) is used to generate one task. The pickup and delivery locations of the task are set to $o_{i,2j}$ and $o_{i,2j+1}$ respectively. The deadline of the task is set to $\left\lceil (1+\phi) \cdot \sum_{h=1}^{2j} d(o_{i,h}, o_{i,h+1}) \right\rceil$, where $d(x,y)$ is the shortest-path distance between two locations x and y.

Note that $\sum_{h=1}^{2j} d(o_{i,h}, o_{i,h+1})$ is the number of timesteps needed by agent a_i to visit the locations $o_{i,1}$ to $o_{i,2j+1}$ in sequence, without considering the existence of the other agents. Thus, it represents the completion time of the j-th task if a_i were to execute the tasks from stream i in sequence and it is the only agent in the environment. 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 k tasks in stream i 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. All the tasks generated from the M streams are put together to form the entire task set T for assignment and planning.[1] The task set generated in the above manner allows us to 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 a wide range of settings. we set $M = 10, 20, 30, 40, 50$ for the small warehouse and set $M = 60, 90, 120, 150, 180$ for the large warehouse. We set $k = 2, 5, 10$ and $\phi = -0.25, -0.1, 0, 0.1, 0.25$. For each setting of (M, k, ϕ), we randomly generate 10 problem instances. So for each warehouse, we test a total of 750 problem instances. We implement the algorithms in C++ and run the experiments on a 3.8GHz AMD Ryzen 3960x machine with 32 GB RAM.

[1] So, agent a_i is not necessarily assigned the tasks in sequence i by our algorithms.

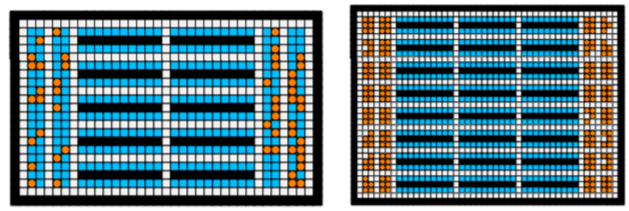

Figure 3: Two 4-neighbor grids that represent simulated warehouses [4]. Black cells are blocked, blue cells represent potential pickup or delivery locations of tasks, and orange circles represent potential parking locations of agents.

Success Rate. The tightness of the deadline setting is a main factor that affects the success rate. Table 1 shows, for each ϕ value, the average success rate of all the 150 problem instances over different (M, k) settings. Tables 2 and 3 detail the average success rate of the 10 problem instances for each (M, k) setting. As expected, the success rate achieved by our algorithms increases with the ϕ value. Recall that the task deadlines are rather tight when $\phi = 0$: task deadlines are set according to the exact shortest-path distances and the tasks in each stream can be completed just in time if they were executed by one agent without any conflicts in the environment. This scenario can be well handled by our framework. Our algorithms can achieve over 98% success rates for $\phi = 0$. When deadlines are looser ($\phi \geq 0.1$), our algorithms can achieve even higher success rates over 99%. Even for $\phi = -0.1$, our algorithms can achieve success rates near 95%. These results demonstrate the effectiveness of our algorithms in assigning and planning tasks to meet their deadlines.

Branch and Bound. To show the efficiency improvement brought by the proposed branch and bound technique, we normalize the computational time without applying the pruning and sorting by that applying pruning and sorting, and refer to it as the speedup ratio. Table 4 shows the average speedup ratio for the 10 problem instances of each (M, k) setting when $\phi = 0$ for the small warehouse. As can be seen, pruning and sorting can speed up the task assignment and planning by 4 – 26 times (the symbol × indicates that we are not able to complete the runs for this (M, k) setting). The speedup ratio generally increases with the number of agents and tasks. The results for other ϕ values show similar trends. The efficiency improvement is even more significant for the large warehouse in that we are not able to complete the runs for most (M, k) settings (and hence do not report the speedup ratios here).

Refined Dummy Path Association. To study the efficiency improvement brought by the refined approach to associating dummy paths, we also compute the speedup ratio, i.e., the computational time for associating dummy paths with agents for every task planning normalized by that for our refined approach. Tables 5 and

Table 1: Average Success Rates for the Small and Large Warehouses

	$\phi = -0.25$	$\phi = -0.1$	$\phi = 0$	$\phi = 0.1$	$\phi = 0.25$
Small Warehouse	0.8382	0.9418	0.9863	0.9948	0.9985
Large Warehouse	0.8832	0.9515	0.9856	0.9939	0.9941

Table 2: Average Success Rates for Different (M, k) settings for the Small Warehouse ($\phi = 0$)

k \ M	10	20	30	40	50
2	0.9800	0.9675	0.9800	0.9725	0.9680
5	0.9840	0.9950	0.9960	0.9925	0.9904
10	0.9950	0.9970	0.9937	0.9923	0.9912

Table 3: Average Success Rates for Different (M, k) settings for the Large Warehouse ($\phi = 0$)

k \ M	60	90	120	150	180
2	0.9958	0.9894	0.9875	0.9767	0.9650
5	0.9980	0.9960	0.9880	0.9861	0.9748
10	0.9982	0.9924	0.9867	0.9809	0.9681

Table 4: Speedup Ratio of Branch and Bound for the Small Warehouse ($\phi = 0$)

k \ M	10	20	30	40	50
2	4.307	9.955	16.59	21.62	26.39
5	6.367	11.97	18.658	×	×
10	6.968	12.61	×	×	×

6 show the average speed ratio for the 10 problem instances of each $(M.k)$ setting when $\phi = 0$ for the small and large warehouses respectively. As can be seen, our refined approach can reduce the computational time by up to 30% for the large warehouse, while it does not have a significant impact on the computational time for the small warehouse. This show the importance of refining dummy path computation in large-scale environments. The results for other ϕ values have similar trends and are not shown here due to space limitations.

5 CONCLUSIONS

We have studied the MAPD-TD problem. We have adopted an integrated approach to develop a joint task assignment and path planning framework. 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 easily extended to optimize other metrics such as makespan and sum-of-costs. For example, when an objective of makespan minimization is considered, we can adapt the choice standards (1) and (3) to choose the agent and task with the earliest completion times. The branch and bound technique can still work by using a upper bound of the task completion time. The refined approach to associating dummy paths can also reduce the number of dummy paths needed. In the future, we will extend the framework to optimize other metrics.

REFERENCES

[1] Peter R. Wurman, Raffaello D'Andrea, and Mick Mountz. "Coordinating hundreds of cooperative, autonomous vehicles in warehouses." AI magazine 29.1 (2008): 9-9.

[2] R. Morris, C. S. Pasareanu, K. Luckow, W. Malik, H. Ma, T. K. S. Kumar, and S. Koenig. "Planning, scheduling and monitoring for airport surface operations." In AAAI Workshop on Planning for Hybrid Systems, 2016.

[3] Manuela Veloso, Joydeep Biswas, Brian Coltin, and Stephanie Rosenthal. 2015. "CoBots: robust symbiotic autonomous mobile service robots." In Proceedings of the 24th International Conference on Artificial Intelligence (IJCAI'15). AAAI Press, 4423–4429.

[4] Minghua Liu, Hang Ma, Jiaoyang Li, and Sven Koenig. "Task and Path Planning for Multi-Agent Pickup and Delivery." In Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems, pp. 1152-1160. IFAAMAS, 2019.

[5] 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'20). IFAAMAS, 1843–1845.

[6] 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'20). IFAAMAS, 1898–1900.

[7] 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 AAAI Conference on Artificial Intelligence. Hilton Hawaiian Village, Honolulu, Hawaii, USA, 7651–7658.

[8] 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 Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017. Sao Paulo, Brazil, 837–845.

Table 5: Average Speedup Ratio of the Refined Approach to Associating Dummy Path for the Small Warehouse ($\phi = 0$)

k \ M	10	20	30	40	50
2	1.039	0.9160	1.003	1.004	0.9969
5	1.031	1.001	1.000	1.002	1.002
10	0.9952	1.002	0.9988	1.004	0.9982

Table 6: Average Speedup Ratio of the Refined Approach to Associating Dummy Path for the Large Warehouse ($\phi = 0$)

k \ M	60	90	120	150	180
2	1.422	1.365	1.262	1.220	1.208
5	1.218	1.088	1.002	0.9846	0.9881
10	1.018	1.066	1.004	0.8635	0.9514

[9] 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'18). AAAI Press, 417–423.

[10] D. Karger, C. Stein, and J. Wein. Scheduling algorithms. In CRC Handbook of Computer Science. 1997.

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

[12] Roni Stern. Multi-agent path finding-an overview. In Artificial Intelligence, pages 96–115. Springer, 2019.

[13] D. S. Nau, V. Kumar, and L. Kanal, "General branch and bound, and its relation to A* and AO*." Artificial Intelligence, vol. 23, pp. 29-58, 1984.

[14] Jingjin Yu and Steven M. LaValle. 2013. "Structure and intractability of optimal multi-robot path planning on graphs." In Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence (AAAI'13). AAAI Press, 1443–1449.

[15] M. van der Boor, S.C. Borst, J.S. van Leeuwaarden, D. Mukherjee, Scalable load balancing in networked systems: A survey of recent advances, arXiv preprint arXiv:1806.05444.

[16] Michael Mitzenmacher. 2001. The Power of Two Choices in Randomized Load Balancing. IEEE Trans. Parallel Distrib. Syst. 12, 10 (October 2001), 1094–1104.

[17] Yi Lu, Qiaomin Xie, Gabriel Kliot, Alan Geller, James R. Larus, and Albert Greenberg. 2011. Join-Idle-Queue: A novel load balancing algorithm for dynamically scalable web services. Perform. Eval. 68, 11 (November, 2011), 1056–1071.