POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Executive Summary of the Thesis

Robustness in Multi-Agent Pickup and Delivery with Delays

Laurea Magistrale in Computer Science and Engineering - Ingegneria Informatica

Author: Giacomo Lodigiani

Advisor: Prof. Francesco Amigoni

Co-advisor: Prof. Nicola Basilico

Academic year: 2020-2021

1. Introduction

In Multi-Agent Pickup and Delivery (MAPD) [6], a set of agents must jointly plan collision-free paths to serve pickup-delivery tasks that are submitted at run-time. MAPD combines the resolution of a task-assignment problem, where agents must be assigned to pickup-delivery pairs of locations, with Multi-Agent Path Finding (MAPF) [10], where paths for completing the assigned tasks must be computed. A particularly challenging feature of MAPD problems is their long-term and dynamic nature that allows for new tasks to be submitted at any time and location in the environment.

Despite studied only recently, MAPD has a great relevance for a number of real-world application domains. Automated warehouses, where robots continuously fulfill new orders, have arguably the most significant industrial deployments [14]. Beyond logistics, MAPD applications include also the coordination of teams of service robots [13] or fleets of autonomous cars, and the automated control of non-player characters in video games [9].

Recently, the MAPF community has focused on *robustness* [1, 2, 6], generally understood as a property of solutions that can withstand real-world-induced relaxations of some idealistic assumptions made by the models. A typical example is represented by the assumption that paths are executed without errors. In reality, however, paths execution is subject to delays and other issues that can hinder some properties (e.g., the absence of collisions) of a solution. In contrast, robustness in the long-term setting of MAPD has not been yet consistently studied.

In this paper, we study the robustness of MAPD to the occurrence of *delays* by defining a variant of the problem that we call *MAPD-d* (*MAPD with delays*). In this variant, agents, like in standard MAPD, are assigned to tasks (pickup-delivery locations pairs), which may continuously appear at any time step, and paths to accomplish those tasks avoiding collisions are computed. During path execution, delays can occur at arbitrary times, causing one or more agents to halt at some time steps, thus slowing down the execution of their planned paths. We devise a set of algorithms to compute robust solutions for MAPD-d. The first one is based on a decentralized MAPD algorithm, Token Passing (TP), to which we added some recovery routines that provide replanning in case collisions caused by delays are detected. TP is able to solve well-formed MAPD problem instances [8], and we show that, under some assumptions, the introduction of delays in MAPD-d does not affect well-formedness. We then propose two new algorithms, k-TP and p-TP, which adopt the approach of robust planning, computing paths that limit the risk of collisions caused by potential delays. k-TP returns solutions with deterministic guarantees about robustness in face of delays (k-robustness), while solutions returned by p-TP have probabilistic robustness guarantees (p-robustness). We compare these algorithms by running experiments in simulated environments and we evaluate the trade-offs offered by different levels of robustness.

In summary, the main contributions of this paper are: the introduction of the MAPD-d problem and the study of some of its properties (Section 3), the definition of two algorithms (k-TP and p-TP) for solving MAPD-d problems with robustness guarantees (Section 4), and their experimental evaluation that provides insights about how robustness and solution cost can be balanced (Section 5).

2. Preliminaries and Related Work

In this section, we present the MAPD problem, we discuss the different concepts of robustness from the MAPF and MAPD literature, and finally we illustrate some algorithms for MAPD problems.

2.1. MAPD

Informally, a MAPF problem [10] involves a set of agents, each one with a starting vertex and a target vertex in a graph representing the environment, and asks for a set of paths, one for each agent, such that, when the agents follows such paths they reach their targets without collisions. The paths could be also required to minimize some cost function.

A MAPD problem [6] consists of:

- A finite connected undirected graph $G = (V, E)$, whose vertices V represent locations and whose edges E represent connections between locations that the agents can traverse.
- A set of ℓ agents $A = \{a_1, a_2, \ldots, a_\ell\}$.
- A task set $\mathcal{T}$ that contains the unexecuted tasks in the system. The task set changes dynamically as, at each time step, new tasks can be added to the system. Each task $\tau_j \in \mathcal{T}$ is characterized by a pickup vertex $s_j \in V$ and a delivery vertex $g_j \in V$ and is added to the system at an unpredictable (finite) time step. A task is known to the agents and can thus be executed from the time step at which it is added to $\mathcal{T}$.

Time is discrete and starts from time step 0. At time step 0, each agent starts from an initial vertex. Initial vertices are all different.

Agents move in the environment represented by G along paths.

Definition 1. *A path $\pi_i = \langle \pi_{i,t}, \pi_{i,t+1}, ..., \pi_{i,t+n} \rangle$ for agent a_i starting at time step t and ending at time step $t + n$ is a finite sequence of vertices $\pi_{i,h} \in V$ that satisfies the following condition: the agent either moves to an adjacent vertex or does not move, that is, for any vertex $\pi_{i,h}$ in π_i, $(\pi_{i,h}, \pi_{i,h+1}) \in E$ or $\pi_{i,h+1} = \pi_i$.*

An agent is called *free* when it is currently not executing any task. Otherwise, it is called *occupied* when it is assigned to a task. If an agent is free, it can be assigned to any task $\tau_j \in \mathcal{T}$ (thus becoming occupied), with the constraint that a task can be assigned to only one agent. When a task is assigned to an agent, it is removed from $\mathcal{T}$. To execute a task τ_j, the assigned agent has to plan and follow paths to move first from its current location to the pickup vertex s_j of the task and then from there to the delivery vertex g_j. When the agent arrives at the delivery vertex g_j, the task is completed and the agent becomes free.

We call *plan* the current set of paths computed by the agents. Paths in a plan are required to be be *collision-free*, namely two (or more) agents, when following their paths, cannot be in the same vertex or traverse the same edge at the same time step. Solving a MAPD problem means finding collision-free paths that complete all the tasks in $\mathcal{T}$. Due to the dynamic and online nature of MAPD, the paths cannot be fully planned in advance, but they are planned as soon as the tasks appear. The quality of a solution for a MAPD problem is measured according to service time or to makespan.

Definition 2. *The service time is the average number of time steps needed to complete each task, measured from the time step it is added to $\mathcal{T}$.*

Definition 3. *The makespan is the earliest time step when all tasks are completed.*

Since MAPD is a generalization of MAPF, and MAPF is NP-hard to solve optimally [11, 15], also MAPD is NP-hard to solve optimally, either using the service time or makespan objective function.

2.2. Robustness

In the context of MAPF, a robust solution allows to follow all the paths even when some unexpected event forces a deviation of the execution from what originally expected. In real applications, this behaviour is typically caused by delays afflicting agents' executions of planned paths. When an agent, following its path, intends to move to an adjacent vertex, a *delay* leaves the agent at its current vertex, thus slowing down the execution of the path. In the MAPF setting, different kinds of robustness have been considered.

The idea of k-robustness, introduced by Atzmon et al. [2], is defined as follows.

Definition 4. *A plan is k-robust iff it is collision-free and remains collision-free when at most k delays for each agent occur.*

To create k-robust plans, an algorithm should ensure that, when an agent leaves a vertex, that vertex is not occupied by another agent before k time steps. In this way, even if the first agent delays k times, no collision occurs.

The concept of p-robustness [1] is an alternative to k-robustness. Assume to know the delay probability p_d, which is the probability that an agent is delayed at a given time step. Assume also that delays are independent of each other and that the delay probability is fixed across all agents, locations, and time steps (this last assumption can be easily generalized). Then, p-robustness is defined as follows.

Definition 5. *A plan is p-robust iff the probability that it will be executed without a collision is at least p.*

Note that p-robustness offers a probabilistic guarantee on the absence of collisions in presence of delays, while k-robustness offers a deterministic guarantee.

Robustness for MAPD has been less studied. A first result comes from the fact that not all MAPD problem instances are solvable. According to Ma et al. [8] some characteristics of the problem environment, summarized under the term *well-formedness*, are a sufficient condition to enable *long-term robustness*, that is the guarantee to complete a finite number of tasks in a finite time. The underlying idea is that agents could be forced to idle only at specific vertices, called (non-task) *endpoints*, where they do not block other agents. A MAPD problem instance is well-formed when:

1. the number of tasks is finite;
2. the agents are less or equal than the endpoints (arbitrary vertices designated as rest locations);
3. for any two endpoints, there exists a path between them that traverses no other endpoints.

In this paper, we contribute to the study of robustness for MAPD by extending the concepts of k- and p-robustness from MAPF to the long-term setting of MAPD.

2.3. MAPD Algorithms

Some algorithms have been proposed to address the MAPD problem. Given the dynamic and online nature of the problem, they interleave planning and execution. Ma et al. [8] illustrate different algorithms able to solve well-formed MAPD problem instances, divided in two categories: decentralized (where each agent assigns itself to tasks and computes its own collision-free paths given some global information) and centralized. From experimental results [8], centralized algorithms offer better results in terms of service time and makespan, but require higher computational costs. A decentralized algorithm, Token Passing, proves instead suitable for real-time long-term operations.

Token Passing (TP, Algorithm 1) is based on a *token*, a synchronized shared block of memory that contains the current paths π_i of all agents, the current task set $\mathcal{T}$, and the current assignment of tasks to the agents. When the algorithm starts, the token is initialized with trivial paths in which agents rest at their initial locations (line 2). At each time step, any task that enters the setting is added to the task set $\mathcal{T}$ (line 4). When an agent has reached the end of its path in the token, it requests the token (at most once per time step). The system then sends the token to each requesting agent, in turn (line 5). The agent with the token can assign itself (line 10) to the task τ in $\mathcal{T}$ whose pickup vertex is closest to its current location (line 9, in experiments we use Manhattan distance as $h()$), provided that no other path already planned (and stored in the token) ends at the pickup or delivery vertex of such task (line 7). The agent then finds a collision-free path from its current position to the pickup vertex and then to the delivery vertex of the task and it eventually rests at the delivery vertex (line 12). Finally, the agent returns the token to the system and moves one step along its path in the token (lines 18 and 20). If it cannot find a feasible path it stays where it is or calls function *Idle* to compute a path to an endpoint (see Section 2.2) in order to avoid deadlocks and ensure long-term robustness (lines 14 and 16).

In this paper, we propose two new algorithms, based on TP, able to produce solutions to MAPD problems, that are not only long-term robust, but also robust to delays.

Algorithm 1: Token Passing

1 /* system executes now */;
2 initialize *token* with the (trivial) path $\langle loc(a_i)\rangle$ for each agent a_i ($loc(a_i)$ is the current location of a_i);
3 **while** *true* **do**
4   add new tasks, if any, to the task set $\mathcal{T}$;
5   **while** *agent a_i exists that requests token* **do**
6   /* system sends *token* to a_i and a_i executes now */;
7     $\mathcal{T}' \leftarrow \{\tau_j \in \mathcal{T} \mid$ no path in *token* ends in s_j or in $g_j\}$;
8   **if** $\mathcal{T}' \neq \{\}$ **then**
9       $\tau \leftarrow \arg\min_{\tau_j \in \mathcal{T}'} h(loc(a_i), s_j)$;
10   assign a_i to τ;
11   remove τ from $\mathcal{T}$;
12   update a_i's path in *token* with the path returned by *PathPlanner*$(a_i, \tau, token)$;
13   **else if** *no task $\tau_j \in \mathcal{T}$ exists with $g_j = loc(a_i)$* **then**
14   update a_i's path in *token* with the path $\langle loc(a_i)\rangle$;
15   **else**
16   update a_i's path in *token* with *Idle*$(a_i, token)$;
17   **end**
18   /* a_i returns *token* to system, which executes now */;
19   **end**
20   agents move along their paths in *token* for one time step;
21   /* system advances to the next time step */;
22 **end**

3. MAPD with Delays

In this section, we first introduce the problem of MAPD with delays, then we discuss the conditions for its well-formedness, and finally we present a simple variation of Token Passing able to guarantee robustness to delays in a mostly reactive way, during execution. In the next section, we propose our algorithms that address delays when planning.

3.1. MAPD-d

Delays are typical problems in real applications of MAPF and MAPD and may have multiple causes. For example, robots can slow down when following paths due to some errors occurring in sensors used for localization and coordination [4]. Moreover, real robots are subject to physical constraints, like minimum turning radius, maximum velocity, and maximum acceleration, and, although algorithms exists to convert time-discrete MAPD plans into plans executable by real robots [7], small differences between models and actual agents may still cause delays. Another source of delays is represented by anomalies occurring during path execution and caused, for example, by partial or temporary failures of some agent [3].

We define the problem of *MAPD with delays* (*MAPD-d*) as a MAPD problem (see Section 2.1) where the execution of the computed paths π_i can be affected, at any time step t, by delays represented by a time-varying set $\mathcal{D}(t) \subseteq A$. Given a time step t, $\mathcal{D}(t)$ specifies the subset of agents that will delay the execution of their paths (lingering at their currently occupied vertex) during time step t. An agent could be delayed for several consecutive time steps (but not for indefinitely long to preserve well-formedness, see next section). The temporal realization of $\mathcal{D}(t)$ is unknown, so a MAPD-d instance is formulated as a MAPD one: no other information is available at planning time. The difference lies in how the solution is searched: in MAPD-d we compute solution accounting for robustness to delays that might happen.

More formally, delays affect each agent's execution trace. Agent a_i's *execution trace* $e_i = \langle e_{i,0}, e_{i,1}, ..., e_{i,m}\rangle$[1] for a given path $\pi_i = \langle \pi_{i,0}, \pi_{i,1}, ..., \pi_{i,n}\rangle$ corresponds to the actual sequence of m ($m \geq n$) vertices traversed by a_i while following π_i and accounting for possible delays. Let us call $idx(e_{i,t})$ the index of $e_{i,t}$ (the vertex occupied by a_i at time step t) in π_i. Given that $e_{i,0} = \pi_{i,0}$, the

[1] For simplicity, we consider a path and a corresponding execution trace starting from time step 0. The generalization to paths starting at a generic time step t is intuitive, but requires a more complex notation and is not reported here.

execution trace is defined, for $t > 0$, as:

$$e_{i,t} = \begin{cases} e_{i,t-1} & \text{if } a_i \in \mathcal{D}(t) \\ \pi_{i,h} \mid h = idx(e_{i,t-1}) + 1 & \text{otherwise} \end{cases}$$

An execution trace terminates when $e_{i,m} = \pi_{i,n}$ for some m.
Notice that, if no delays are present (that is, $\mathcal{D}(t) = \{\}$ for all t) then the execution trace e_i exactly mirrors the path π_i and, in case this is guaranteed in advance, the MAPD-d problem becomes *de facto* a regular MAPD problem. In general, such a guarantee is not given and solving a MAPD-d problem opens the issue of computing collision-free tasks-fulfilling MAPD paths (optimizing service time or makespan) characterized by some level of robustness to delays.
The MAPD-d problem reduces to the MAPD problem as a special case, so the MAPD-d problem is NP-hard.

3.2. Well-formedness of MAPD-d

The fact that delays only affect execution does not harm long-term robustness (namely, the guarantee that a finite number of tasks will be completed in a finite time), since the property is guaranteed by well-formedness that depends mostly on the environment (see Section 2.2). The only possible exception is when an agent cannot move anymore (namely when $e_{i,t+1} = e_{i,t}$ for all $t \geq T$ or, equivalently, when the agent is in $\mathcal{D}(t)$ for all $t \geq T$). In this case, the agent becomes a new obstacle in the environment, potentially blocking a path critical for preserving the well-formedness of the environment. In a real context, this problem can be solved by removing or repairing the blocked agent. So it is reasonable to add the following assumption: if an agent fails permanently, it will be removed (in this case its incomplete task will return in the task set) or repaired after a finite number of time steps. This guarantees that the well-formedness of a problem instance is preserved (or, more precisely, that it is restored after a time interval).
Hence, an instance of the MAPD-d problem is well-formed and, consequently, long-time robust when, in addition to conditions (1)-(3) from Section 2.2, we have:

(4) any agent that cannot move anymore is removed or repaired after a finite number of time steps; if the agent is removed, at least one agent survives in the system (e.g., $\ell \geq 1$).

In what follows, we implicitly consider well-formed instances of MAPD-d problems.

3.3. TP with Recovery Routines

From the previous discussion it follows that algorithms able to solve well-formed MAPD problems, like Token Passing (TP), are in principle able to solve well-formed MAPD-d problems as well. The only issue is that these algorithms return paths that do not consider possible delays occurring during execution. Delays cause planned paths to possibly collide, although they did not at the time they have been created. Note that, according to our assumptions, when an agent is delayed at time step t, there is no way to know for how long it will be delayed. In the original TP algorithm (Section 2.3), only agents that have reached the end of their paths in the token can request the token to plan again. To address the presence of delays, we add a simple recovery routine to the TP algorithm such that, when a collision is detected between agents following their paths in the token, it assigns the token to one of the colliding agents to allow replanning of a new collision-free path. This TP with recovery routines algorithm (Algorithm 2) will be a baseline for experimentally evaluating the algorithms we propose in the next section. In addition to the other information (Section 2.3), we also store in the token the current execution traces of the agents. The algorithm checks if there will be a collision at the current time step using the function *CheckCollisions* in line 5: a collision occurs at time step t if the path π_i of an agent a_i that is not delayed ($a_i \notin \mathcal{D}(t)$) tells a_i to move to a vertex occupied by a delayed agent a_j ($a_j \in \mathcal{D}(t)$). The function returns the set $\mathcal{R}$ of non-delayed colliding agents that try to plan new collision-free paths (line 8). Note that *PathPlanner* considers as constraints the current paths of other agents in the token.
A problem may happen when multiple delays occur at the same time; in particular situations, two or more delayed agent may prevent each other to follow the only paths to complete their tasks. In this case, the algorithm recognizes the situation and implements a deadlock recovery routine. In particular, although with our assumptions agents cannot be delayed forever, we plan short collision-free random walks for the involved agents in order to speedup the deadlock resolution (line 12).

Algorithm 2: TP with recovery routines

1 /* system executes now */;
2 initialize *token* with the (trivial) path $\langle loc(a_i) \rangle$ for each agent a_i;
3 **while** *true* **do**
4   add new tasks, if any, to the task set $\mathcal{T}$;
5     $\mathcal{R} \leftarrow$ *CheckCollisions*(*token*);
6   **foreach** *agent* a_i *in* $\mathcal{R}$ **do**
7   retrieve task τ assigned to a_i;
8         $\pi_i \leftarrow$ *PathPlanner*(a_i, τ, *token*);
9   **if** π_i *is not null* **then**
10   update a_i's path in *token* with π_i;
11   **else**
12   recovery from deadlocks;
13   **end**
14   **end**
15   **while** *agent* a_i *exists that requests token* **do**
16   proceed like in Algorithm 1 (lines 6 - 18);
17   **end**
18   agents move along their paths in *token* for one time step (or stay at their current position if delayed);
19   /* system advances to the next time step*/;
20 **end**

4. MAPD-d Algorithms

In this section we present two algorithms, k-TP and p-TP, able to plan paths that solve MAPD-d problem instances with some guaranteed degree of robustness in face of delays. In particular, k-TP provides a deterministic degree of robustness, while p-TP provides a probabilistic degree of robustness. For developing these two algorithms, we took some inspiration from the corresponding concepts of k- and p-robustness proposed for MAPF (see

Section 2).

4.1. k-TP Algorithm

As we have discussed in Section 3, TP with recovery routines just reacts to the occurrence of delays, ensuring that long-term robustness is preserved. The k-TP algorithm proposed here, instead, plans considering that delays may occur, reducing the need of replanning during execution.

Since it is not a one-shot problem, a k-robust solution for MAPD-d is a plan which is long-term robust and avoids collisions due to at most k consecutive delays for each agent, not only considering the paths already planned but also those planned in the future. This is what our proposed k-TP algorithm does (see full thesis for pseudocode). The basic structure is similar to TP with recovery routines, but the path planning is subject to additional constraints. A new path π_i, before being added to the token, is used to generate the constraints (the k-extension of the path, also added to the token) representing that, at any time step t, any vertex in $\{\pi_{i,t-k}, \ldots, \pi_{i,t-1}, \pi_{i,t}, \pi_{i,t+1}, \ldots, \pi_{i,t+k}\}$ should be considered as an obstacle (at time step t) by agents planning later. In this way, even if agent a_i or agent a_j planning later are delayed up to k times, no collision will occur. For example, if $\pi_i = \langle v_1, v_2, v_3 \rangle$, the 1-extension constraints will forbid any other agent to be in $\{v_1, v_2\}$ at the first time step, in $\{v_1, v_2, v_3\}$ at the second time step, in $\{v_2, v_3\}$ at the third time step, and in $\{v_3\}$ at the fourth time step.

The path of an agent added to the token ends at the delivery vertex of the task assigned to the agent, so the space requested in the token to store the path and the corresponding k-extension constraints is finite, for finite k. Note that, especially for large values of k, it may happen that a sufficiently robust path for an agent a_i cannot be found at some time step; in this case, a_i simply returns the token and tries to replan at the next time step. The idea is that, as other agents advance along their paths, the setting becomes less constrained and a path can be found more easily. Since delays that affect the execution are not known beforehand and an agent could be delayed more than k consecutive time steps, recovery routines are still necessary.

Note that k-TP is an extension of TP with recovery routines, so it is able to solve all well-formed MAPD-d problem instances.

4.2. p-TP Algorithm

The idea of k-robustness considers a fixed value k for the guarantee, which could be hard to set: if k is too low, plans may not be robust enough and the number of replans could be high, while if k is too high, it will increase the total cost of the solution with no extra benefit (see Section 5 for numerical data supporting these claims).

An alternative approach is to resort to the concept of p-robustness (Section 2). A p-robust plan guarantees long-term robustness and keeps collision probability below a certain threshold p ($0 \leq p \leq 1$). In a MAPD setting, where tasks are not known in advance, the planner could quickly reach the threshold with just first few paths planned, so that no other path can be added to the plan until the current paths have been executed. Our solution to avoid this problem is to impose that only the collision probability of individual paths should remain below the threshold p, not the whole plan.

We thus need a way to calculate collision probability for a given path: in the p-TP algorithm (see full thesis for pseudocode) we use Markov chains, a tool typically employed to model the future states of systems when transitions are defined in term of probability [5]. A sequence of states $\{X_t, t \geq 0\}$ is said to be a *Markov chain* if, for all state values x_i, $P\{X_{t+1} = x_{t+1} \mid X_0 = x_0, \ldots, X_{t-1} = x_{t-1}, X_t = x_t\} = P\{X_{t+1} = x_{t+1} \mid X_t = x_t\}$. In fact, p-TP assumes that the set of possible execution traces $\{e_i\}$ corresponding to a path π_i of an agent a_i is compactly represented as a Markov chain, where we have a probability p_d of remaining on the current vertex (probability of being delayed) and a probability $1 - p_d$ of advancing along π_i. Our model assumes that transitions along chains of different agents are independent.

p-TP inherits the structure of TP with recovery routines but, before inserting a new path π_i in the token, a Markov chain associated to the path is derived (states of the Markov chain are the vertices composing the path and transitions of the Markov chain are defined according to p_d, as explained before) and the collision probability $cprob_{\pi_i}$ between path π_i and paths already in the token is calculated. Let us show the procedure in detail. The properties of Markov chains [5] allows to calculate the probability that an agent occupies a vertex at a time step as follows. The probability distribution for the vertex occupied by an agent a_i at the beginning of a path $\pi_i = \langle \pi_{i,t}, \pi_{i,t+1}, ..., \pi_{i,t+n} \rangle$ is given by a (row) vector s_0 with length n that has every element set to 0 except that corresponding to the vertex $\pi_{i,t}$, which is 1. The probability distribution for the location of an agent at time step $t+j$ is given by $s_0 P^j$, where P is the matrix describing transition probabilities constructed considering that an agent has probability $1 - p_d$ of advancing one step in the path. Hence, for any vertex traversed by the path π_i, we calculate its collision probability as 1 minus the probability that all the other agents are not in that vertex at that time step (i.e., the probability that at least one of the other agents is in that vertex at that time step) multiplied by the probability that the agent is actually at that vertex in that given time step. All the probabilities of the steps along the path are summed to obtain the collision probability $cprob_{\pi_i}$ for the path π_i. If this probability is above the threshold p , the path is rejected and a new one is calculated. If an enough robust path is not found after a fixed number of rejections *itermax*, the token is returned to the system and the agent will try to replan at the next time step (as other agents advance along their paths, chances of collisions could decrease).

Also for p-TP, since the delays are not known beforehand, recovery routines are still necessary because p-TP provides only a probabilistic guarantee that collisions won't occur. Moreover, we need to set the value of p_d, with which we build that guarantee, according to the specific application setting. Finally, notice that, since p-TP is an extension of TP with recovery routines, it is able to solve all the well-formed MAPD-d problem instances.

5. Experimental Results

Table 1: Results of experiments in small warehouse with task frequency λ=0.5 and 10 delays per agent

		ℓ=4			ℓ=8		
	k or p	tot. cost	# replans	runtime	tot. cost	#replans	runtime
k-TP	0	**1459.52**	7.26	**0.85**	**1876.72**	16.04	**2.11**
	1	1497.92	1.4	0.91	1925.52	3.85	2.27
	2	1563.28	0.1	1.16	1929.12	0.73	2.15
	3	1644.36	0.01	1.59	2075.04	0.09	3.12
	4	1744.48	**0.0**	2.0	2226.64	**0.04**	4.49
p-TP, p_d=.1	1	**1459.52**	7.26	1.14	**1876.72**	16.04	2.63
	0.5	1478.0	6.29	1.81	1898.16	12.59	5.0
	0.25	1580.28	4.29	2.88	2041.68	5.63	6.11
	0.1	1636.68	2.9	3.16	2151.92	3.23	6.32
	0.05	1714.56	2.93	3.42	2234.08	2.76	6.48
p-TP, p_d=.02	0.5	1466.88	7.34	1.29	1910.64	12.81	3.87
	0.25	1513.68	6.8	1.57	1889.68	10.21	4.38
	0.1	1566.52	4.53	2.37	2003.12	6.73	5.57
	0.05	1622.12	3.51	2.66	2049.92	4.25	5.34

5.1. Setting

Our experiments are conducted on a 3.2 GHz Intel Core i7 8700H laptop with 16 GB of RAM. We tested our algorithms in two warehouse 4-connected grid environments in which effects of delays can be significant: a small one, 15×13, with 4 and 8 agents, and a large one, 25×17, with 12 and 24 agents. (Environments of similar size have been used in [8].) We create a sequence of 50 tasks choosing the pickup and delivery vertices uniformly at random among a set of predefined vertices. The arrival time of each task is determined according to a Poisson distribution [12]. We test 3 different arrival frequencies λ for the tasks: 0.5, 1, and 3 (since, as discussed later, the impact of λ on robustness is not relevant, we do not show results for all values of λ). At the beginning, the agents are located at the endpoints selected for well-formedness (Section 2.2).

We measure the total cost of a solution as the sum of the lengths of all the paths in a run (total cost is strictly related to service time), the number of replans performed during execution, and the total runtime of a simulation (in s). Results are averaged over 100 runs. During each run, 10 delays per agent are randomly inserted. A run ends when all the tasks have been completed.

We test both k-TP and p-TP against the baseline TP with recovery routines (to the best of our knowledge, we are not aware of any other algorithm for finding robust solutions to MAPD-d). For p-TP we use two different values for the parameter p_d, 0.02 and 0.1, modeling a low and a higher probability of delay, respectively. (Note that this is the expected delay probability used to calculate the robustness of a path and could not match with the delays actually observed.) For planning paths of individual agents (*PathPlanner* in the algorithms), we use an A* path planner with Manhattan distance as heuristic. All algorithms are implemented in Python[2].

5.2. Results

Results relative to small warehouse are shown in Tables 1 and 2 and those relative to large warehouse are shown in Tables 3 and 4. To keep readability, we do not report the standard deviation in tables. Standard deviation values do not present any evident oddity and support the conclusions about the trends that are reported below.

The baseline algorithm, TP with recovery routines, appears two times in each table: as k-TP with $k = 0$ (that

[2] *Link to GitHub code:* `https://github.com/Lodz97/Multi-Agent_Pickup_and_Delivery.git`

Table 2: Results of experiments in small warehouse with task frequency λ=3 and 10 delays per agent

		ℓ=4			ℓ=8		
	k or p	tot. cost	# replans	runtime	tot. cost	# replans	runtime
k-TP	0	**1419.08**	8.3	**0.6**	**1742.32**	14.67	1.93
	1	1452.88	1.47	0.77	1758.96	4.01	**1.81**
	2	1534.36	0.2	0.95	1814.0	0.58	1.89
	3	1603.08	0.01	1.33	2001.84	0.12	3.02
	4	1716.48	**0.0**	1.68	2107.76	**0.01**	4.32
p-TP, p_d=.1	1	**1419.08**	8.3	0.86	**1742.32**	14.67	2.53
	0.5	1441.16	6.7	1.45	1794.48	11.06	4.93
	0.25	1527.92	5.12	2.3	1961.92	6.46	5.83
	0.1	1619.68	2.93	2.81	2011.36	3.55	5.66
	0.05	1668.16	2.65	3.05	2101.84	3.65	6.11
p-TP, p_d=.02	0.5	1432.56	8.05	1.25	1756.64	13.19	3.61
	0.25	1491.68	7.02	1.57	1826.0	10.93	3.77
	0.1	1521.24	4.41	2.12	1871.76	6.89	4.65
	0.05	1574.2	3.45	2.5	1956.96	4.81	4.98

Table 3: Results of experiments in large warehouse with task frequency λ=0.5 and 10 delays per agent

		ℓ=12			ℓ=24		
	k or p	tot. cost	# replans	runtime	tot. cost	# replans	runtime
k-TP	0	3403.44	17.18	**2.8**	6462.0	20.71	8.32
	1	**3320.4**	3.88	3.27	**6359.04**	5.37	**5.78**
	2	3423.84	1.18	4.89	6611.52	1.62	9.54
	3	3648.6	0.24	7.54	7213.2	0.4	15.55
	4	3727.08	**0.01**	10.9	7210.8	**0.1**	22.11
p-TP, p_d=.1	1	3403.44	17.18	4.12	6462.0	20.71	11.2
	0.5	3443.4	10.02	11.3	7002.72	17.09	38.61
	0.25	3661.56	5.38	17.26	7527.12	9.59	58.95
	0.1	3966.96	4.51	19.6	7734.24	4.51	54.92
	0.05	4047.96	3.56	20.27	8373.36	3.89	57.24
p-TP, p_d=.02	0.5	3478.32	14.51	7.41	6961.2	20.3	28.74
	0.25	3452.64	9.92	10.19	6882.48	14.15	39.47
	0.1	3732.0	6.53	13.76	7301.76	8.94	49.04
	0.05	3760.56	6.41	14.91	7394.4	7.02	49.96

is the basic implementation as in Algorithm 2) and as p-TP with $p_d = 0.1$ and $p = 1$ (which accepts all paths). The two versions of the baseline return the same results in terms of total cost and number of replans (we use the same random seed initialization for runs with different algorithms), but the total runtime is larger in the case of p-TP, due to the overhead of calculating the Markov chains and the collision probability for each path.

Looking at robustness, which is the goal of our algorithms, we can see that, in all settings, both k-TP and p-TP significantly reduce the number of replans with respect to the baseline. For k-TP, increasing k leads to increasingly more robust solutions with less replans, and the same happens for p-TP when the threshold probability p is reduced. However, increasing k shows a more evident effect on the number of replans than reducing p. More robust solutions, as expected, tend to have a larger total cost, but the first levels of robustness ($k = 1$, $p = 0.5$) manage to reduce significantly the number of replans with a small or no increase in total cost. For instance, in Table 4, k-TP with $k = 1$ decreases the number of replans of more than 75% with an increase in total cost of less than 2%, with respect to the baseline. Pushing towards higher degrees of robustness (i.e., increasing k or decreasing p) tends to increase cost significantly with diminishing returns in terms of number of replans, especially for k-TP.

Comparing k-TP and p-TP, it is clear that solutions produced by k-TP tend to be more robust at similar total costs (e.g., see k-TP with $k = 1$ and p-TP with $p_d = .1$ and $p = 0.5$ in Table 1), and decreasing p may sometimes lead to relevant increases in costs. This suggests that our implementation of p-TP has margins for improvement: if

Table 4: Results of experiments in large warehouse with task frequency λ=3 and 10 delays per agent

	k or p	ℓ=12 tot. cost	ℓ=12 # replans	ℓ=12 runtime	ℓ=24 tot. cost	ℓ=24 # replans	ℓ=24 runtime
k-TP	0	3182.76	18.96	**2.91**	6203.76	30.83	**8.12**
	1	3237.36	4.22	3.28	6109.44	8.98	9.81
	2	3297.36	1.19	4.75	6271.2	1.71	12.03
	3	3348.24	0.18	7.31	6565.44	0.59	19.43
	4	3487.08	**0.04**	10.76	6769.68	**0.17**	30.91
p-TP, p_d=.1	1	3182.76	18.96	4.16	6203.76	30.83	10.78
	0.5	3224.88	11.31	9.04	6183.36	17.21	36.74
	0.25	3576.12	7.39	14.58	6906.0	9.96	48.14
	0.1	3820.44	5.3	16.33	7451.04	6.32	47.11
	0.05	3973.2	3.83	16.83	8017.44	4.42	47.62
p-TP, p_d=.02	0.5	**3115.68**	12.47	7.22	**5946.24**	20.47	26.21
	0.25	3477.0	12.05	9.23	6350.4	15.72	39.68
	0.1	3360.84	6.78	11.59	6975.6	9.88	42.76
	0.05	3580.08	6.21	12.98	7048.32	8.81	42.23

the computed path exceeds the threshold p we wait the next time step to replan, without storing any collision information extracted from the Markov chains; finding ways to exploit this information may lead to an enhanced version of p-TP (this investigation is left as future work). It is also interesting to notice the effect of p_d in p-TP: a higher p_d (which, in our experiments, amounts to overestimating the actual delay probability that, considering that runs last on average about 300 time steps and there are 10 delays per agent, is equal to $\frac{10}{300} = 0.03$) leads to solutions requiring less replans, but with a noticeable increase in total cost.

Considering runtimes, k-TP and p-TP are quite different. For k-TP, we see a trend similar to that observed for total cost: a low value of k ($k = 1$) often corresponds to a slight increase in runtime with respect to the baseline (sometimes even a decrease), while for larger values of k the runtime may be much longer than the baseline. Instead, p-TP shows a big increase in runtime with respect to the baseline, but then it does not change too much, at least for low values of p ($p = 0.1$, $p = 0.05$). Finally, we can see how different task frequencies λ have no significant impact on our algorithms, but higher frequencies have the global effect of reducing total costs since tasks (which are always 50 per run) are available earlier.

We repeat the previous experiments increasing the number of random delays inserted in execution to 50 per agent, thus generating a scenario with multiple troubled agents. Both algorithms significantly reduce the number of replans with respect to the baseline, reinforcing the importance of addressing possible delays during planning and not only during execution, especially when the delays can dramatically affect the operations of the agents, like in this case. The k-TP algorithm performs better than the p-TP one, with trends similar to those discussed above.

Finally, we run simulations in a even larger warehouse 4-connected grid environment of size 25×37, with 50 agents, $\lambda = 1$, 100 tasks, and 10 delays per agent. The same qualitative trends discussed above are observed also in this case. For example, k-TP with $k = 2$ reduces the number of replans of 93% with an increase of total cost of 5% with respect to the baseline. The runtime of p-TP grows to hundreds of seconds, also with large values of p, suggesting that some improvements are needed. Full results are not reported here due to space constraints.

6. Conclusion

In this paper, we introduced a variation of the Multi-Agent Pickup and Delivery (MAPD) problem, called MAPD with delays (MAPD-d), which considers an important practical issue encountered in real applications: delays in execution. In a MAPD-d problem, agents must complete a set of incoming tasks (by moving to the pickup vertex of each task and then to the corresponding delivery vertex) even if they are affected by an unknown but finite number of delays during execution. We proposed two algorithms to solve MAPD-d, k-TP and p-TP, that are able to solve well-formed MAPD-d problem instances and provide deterministic and probabilistic robustness guarantees, respectively. Experimentally, we compared them against a baseline algorithm that reactively deals with delays during execution. Both k-TP and p-TP plan robust solutions, greatly reducing the number of replans needed with a small increase in solution cost and runtime. k-TP showed the best results in terms of robustness-cost trade-off, but p-TP still offers great opportunities for future improvements.

Future work will address the enhancement of p-TP according to what we outlined in Section 5.2 and the experimental testing of our algorithms in real-world settings.

References

[1] Dor Atzmon, Roni Stern, Ariel Felner, Nathan R. Sturtevant, and Sven Koenig. Probabilistic robust multi-agent path finding. In *Proc. ICAPS*, pages 29–37, 2020.

[2] Dor Atzmon, Roni Stern, Ariel Felner, Glenn Wagner, Roman Barták, and Neng-Fa Zhou. Robust multi-agent path finding. In *Proc. AAMAS*, page 1862–1864, 2018.

[3] Pinyao Guo, Hunmin Kim, Nurali Virani, Jun Xu, Minghui Zhu, and Peng Liu. Roboads: Anomaly detection against sensor and actuator misbehaviors in mobile robots. In *Proc. DSN*, pages 574–585, 2018.

[4] Eliahu Khalastchi and Meir Kalech. Fault detection and diagnosis in multi-robot systems: A survey. *Sensors*, 19(18):1–19, 2019.

[5] David A Levin and Yuval Peres. *Markov chains and mixing times*, volume 107. American Mathematical Soc., 2017.

[6] Hang Ma. *Target Assignment and Path Planning for Navigation Tasks with Teams of Agents*. PhD thesis, University of Southern California, Department of Computer Science, Los Angeles, CA, 2020.

[7] Hang Ma, Wolfgang Hönig, T. K. Satish Kumar, Nora Ayanian, and Sven Koenig. Lifelong path planning with kinematic constraints for multi-agent pickup and delivery. In *Proc. AAAI*, pages 7651–7658, 2019.

[8] Hang Ma, Jiaoyang Li, T.K. Satish Kumar, and Sven Koenig. Lifelong multi-agent path finding for online pickup and delivery tasks. In *Proc. AAMAS*, page 837–845, 2017.

[9] Hang Ma, Jingxing Yang, Liron Cohen, T. K. Kumar, and Sven Koenig. Feasibility study: Moving non-homogeneous teams in congested video game environments. *Proc. AIIDE*, pages 270–272, 2017.

[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, Roman Barták, and Eli Boyarski. Multi-agent pathfinding: Definitions, variants, and benchmarks. In *Proc. SoCS*, pages 151–159, 2019.

[11] Pavel Surynek. An optimization variant of multi-robot path planning is intractable. In *Proc. AAAI*, page 1261–1263, 2010.

[12] Kung-Kuen Tse. Some applications of the poisson process. *Appl. Math.*, 05:3011–3017, 2014.

[13] Manuela Veloso, Joydeep Biswas, Brian Coltin, and Stephanie Rosenthal. Cobots: Robust symbiotic autonomous mobile service robots. In *Proc. IJCAI*, page 4423–4429, 2015.

[14] Peter R. Wurman, Raffaello D'Andrea, and Mick Mountz. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. In *Proc. IAAI*, page 1752–1759, 2007.

[15] Jingjin Yu and Steven M. LaValle. Structure and intractability of optimal multi-robot path planning on graphs. In *Proc. AAAI*, page 1443–1449, 2013.