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 $A = \{a_1, a_2, \dots, a_\ell\}$.
- 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 ⟨loc(a_i)⟩ 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 𝒯;
5      while agent a_i exists that requests token do
6          /* system sends token to a_i and a_i
             executes now */;
7          𝒯' ← {τ_j ∈ 𝒯 | no path in token ends in s_j
             or in g_j};
8          if 𝒯' ≠ {} then
9              τ ← arg min_{τ_j ∈ 𝒯'} h(loc(a_i), s_j);
10             assign a_i to τ;
11             remove τ from 𝒯;
12             update a_i's path in token with the path
                 returned by PathPlanner(a_i, τ, token);
13         else if no task τ_j ∈ 𝒯 exists with
             g_j = loc(a_i) then
14             update a_i's path in token with the path
                 ⟨loc(a_i)⟩;
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 ⟨loc(a_i)⟩ for
     each agent a_i;
3  while true do
4      add new tasks, if any, to the task set 𝒯;
5      ℛ ← CheckCollisions(token);
6      foreach agent a_i in ℛ do
7          retrieve task τ assigned to a_i;
8          π_i ← 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 $\{\pi_{i,t-k}, \dots, \pi_{i,t-1}, \pi_{i,t}, \pi_{i,t+1}, \dots, \pi_{i,t+k}\}$ 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, $P\{X_{t+1} = x_{t+1} \mid X_0 = x_0, \dots, X_{t-1} = x_{t-1}, X_t = x_t\} = P\{X_{t+1} = x_{t+1} \mid X_t = x_t\}$. 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

	k or p	ℓ=4			ℓ=8		
		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 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 2: Results of experiments in small warehouse with task frequency λ=3 and 10 delays per agent

	k or p	ℓ=4			ℓ=8		
		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

	k or p	ℓ=12			ℓ=24		
		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

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

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

	k or p	ℓ=12			ℓ=24		
		tot. cost	# replans	runtime	tot. cost	# replans	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.

POLITECNICO DI MILANO

School of Industrial and Information Engineering

Department of Electronics, Information and Bioengineering

Master of Science Degree in Computer Science and Engineering

Robustness in Multi-Agent Pickup and Delivery with Delays

Supervisor: Prof. Francesco Amigoni
Co-Supervisor: Prof. Nicola Basilico

Author:
Giacomo Lodigiani, 946455

Academic Year 2020-2021

Alla mia famiglia, ai miei amici, e a tutti quelli
che hanno sempre creduto in me.

Abstract

Multi-Agent Pickup and Delivery (MAPD) is the problem of computing collision-free paths for a group of agents such that they can safely reach delivery locations from pickup ones. These locations are provided at runtime, making MAPD a combination between classical Multi-Agent Path Finding (MAPF) and online task assignment. Current algorithms for MAPD do not consider many of the practical issues encountered in real applications: real agents often do not follow the planned paths perfectly, and may be subject to delays and failures. The objectives of this thesis are to study the problem of MAPD with *delays*, and to present solution approaches that provide robustness guarantees by planning paths that limit the effects of imperfect execution. In particular, two algorithms are introduced, k-TP and p-TP, both based on a decentralized algorithm typically used to solve MAPD, Token Passing (TP), which offer deterministic and probabilistic guarantees, respectively. Experimentally, these algorithms are compared against a version of TP enriched with recovery routines. k-TP and p-TP, planning robust solutions, are able to significantly reduce the number of replans caused by delays, with little or no increase in solution cost and running time.

Estratto in Lingua Italiana

In un problema di Multi-Agent Pickup and Delivery (MAPD) [19], un gruppo di agenti deve congiuntamente pianificare percorsi senza collisioni per svolgere compiti di raccolta-consegna che appaiono a tempo di esecuzione. Il MAPD combina la soluzione del problema di assegnamento dei compiti, dove gli agenti devono essere assegnati a coppie di posizioni di raccolta-consegna, col Multi-Agent Path Finding (MAPF) [30], dove bisogna calcolare i percorsi per completare i compiti assegnati. Una caratteristica particolarmente impegnativa dei problemi di MAPD è la loro natura a lungo termine e dinamica, che permette a nuovi compiti di apparire in ogni momento e in ogni posizione nell'ambiente.

Motivazione

Il problema MAPD ha una grande importanza in un numero considerevole di applicazioni reali. I magazzini automatizzati, dove i robot soddisfano continuamente nuovi ordini, ne costituiscono sicuramente il più significativo impiego industriale; un esempio è il sistema sviluppato da Amazon Robotics impiegato con successo da Amazon [43]. Oltre alla logistica, le applicazioni del problema MAPD includono anche il coordinamento di squadre di robot di servizio [40] o di flotte di auto a guida autonoma, e il controllo automatico di personaggi non giocanti nei videogiochi [23]. Altri esempi includono veicoli per il traino automatico di aeromobili [25], la gestione automatica degli incroci [7], i robot per il trasporto di oggetti [24], i robot per attività di pattuglia [1], i robot di ricerca e salvataggio [14], e le flotte di robot a trasmissione differenziale e quadricotteri [13].

Obiettivi

Recentemente, la comunità MAPF si è focalizzata sulla *robustezza* [2,3,19], generalmente intesa come una proprietà delle soluzioni di resistere a rilassamenti, suggeriti dalle applicazioni concrete, di alcune assunzioni fatte dai modelli. Un tipico esempio è rappresentato dall'assunzione che i percorsi vengano eseguiti senza errori. In realtà l'esecuzione dei percorsi è soggetta a ritardi e ad altri problemi che possono compromettere alcune proprietà (ad esempio l'assenza di collisioni) della soluzione. La robustezza nel contesto a lungo termine del problema MAPD, invece, non è stata ancora consistentemente studiata. Questa tesi ha l'obiettivo di contribuire a colmare questo divario studiando l'impatto dei ritardi sul problema MAPD proponendo soluzioni applicabili in contesti realistici.

Contributi

In questa tesi, la robustezza del problema MAPD in caso di *ritardi* viene studiata definendo una variante del problema chiamata *MAPD-d* (*MAPD with delays*). In questa variante, gli agenti, come nel problema MAPD classico, vengono assegnati a compiti (coppie di posizioni di raccolta-consegna), i quali possono continuamente comparire ad ogni istante di tempo, e successivamente vengono calcolati i percorsi per completare tali compiti evitando collisioni. Durante l'esecuzione dei percorsi, dei ritardi possono accadere in istanti arbitrari, causando l'arresto di uno o più agenti in alcuni istanti di tempo, rallentando quindi l'esecuzione dei loro percorsi pianificati. In questo lavoro vengono proposti diversi algoritmi per calcolare soluzioni robuste al problema MAPD-d. Il primo è basato su un algoritmo MAPD decentralizzato, chiamato Token Passing (TP), al quale sono state aggiunte alcune procedure di recupero che permettono di ripianificare quando vengono individuate collisioni causate da ritardi. Il TP è in grado di risolvere istanze MAPD ben-formate [22], e viene dimostrato che, con alcune assunzioni, l'introduzione dei ritardi nel problema MAPD-d non intacca la proprietà di istanza ben-formata. In seguito, vengono presentati due nuovi algoritmi, k-TP e p-TP, che adottano l'approccio della pianificazione robusta, calcolando percorsi che limitano il rischio di collisioni causate da potenziali ritardi. L'algoritmo k-TP ritorna soluzioni con garanzie deterministiche sulla robustezza in caso di ritardi (k-robustezza), mentre le soluzioni calcolate dall'algoritmo p-TP hanno garanzie di robustezza probabilistiche (p-robustezza). Infine, questi algoritmi sono comparati con esperimenti eseguiti in ambienti simulati e vengono valutati i compromessi offerti da diversi

livelli di robustezza.

Struttura della tesi

I contenuti della tesi sono organizzati nei seguenti capitoli.
Il Capitolo 1 introduce il contesto, le motivazioni, gli obiettivi e i contributi della ricerca svolta.
Nel Capitolo 2 viene introdotto il problema MAPF con le sue proprietà e vengono presentati vari algoritmi, ottimi e sub-ottimi, usati per risolverlo con i loro vari compromessi. In seguito sono discusse le limitazioni del problema MAPF, le quali hanno portato all'introduzione di nuove varianti del problema. In particolare, viene studiato nel dettaglio il problema MAPD (struttura teorica, algoritmi, assunzioni), una versione a lungo termine del problema MAPF che riesce a modellare fedelmente ciò che accade in contesti applicativi. Infine vengono descritte brevemente altre direzioni seguite dalla attuale ricerca per avvicinare il problema MAPF alle applicazioni, come considerare i vincoli cinematici o ritardi nell'esecuzione.
Nel Capitolo 3 viene introdotto il problema *MAPD-d* (*MAPD with delays*), che estende il concetto di ritardi al contesto a lungo termine del MAPD, vengono studiate le sue proprietà teoriche ed è proposto un algoritmo come base di riferimento, TP con procedure di recupero, che permette di risolvere un significativo sottoinsieme di istanze di problemi MAPD-d.
Il Capitolo 4 raccoglie il principale contributo algoritmico di questa tesi, che consiste in due algoritmi, k-TP e p-TP, i quali, considerando la possibilità di ritardi a tempo di pianificazione, permettono di ridurre significativamente l'impatto dei ritardi sull'esecuzione. Questi algoritmi seguono rispettivamente un approccio deterministico e probabilistico.
Nel Capitolo 6, k-TP and p-TP sono confrontati con l'algoritmo di rifermento TP con procedure di recupero in diversi ambienti che rappresentano magazzini simulati. Vengono analizzati i compromessi dei diversi livelli di robustezza e l'impatto degli algoritmi nel ridurre ripianificazioni causate da ritardi in diversi ambienti con diverso numero di robot, di ritardi, e diversa frequenza degli incarichi.
Nel Capitolo 7 sono riassunti i risultati ottenuti i questa tesi, sottolineando punti di forza e limiti dei metodi proposti. Infine, vengono suggeriti possibili miglioramenti e future direzioni per la ricerca.
L'Appendice A arricchisce gli esperimenti riportati nel Capitolo 6 presentando ulteriori simulazioni effettuate con differenti parametri e dettagli sulle deviazioni standard.
Nell'Appendice B un breve manuale utente mostra al lettore come usare

il codice fornito per eseguire le simulazioni e gli esperimenti presentati in questa tesi.

Ringraziamenti

Prima di tutto, desidero porgere un sentito ringraziamento al mio relatore, il Prof. Francesco Amigoni, per avermi affidato un progetto di notevole interesse ed essersi prestato, con pazienza e dedizione, a farmi da guida in questo lungo percorso. Voglio inoltre ringraziare il mio correlatore, il Prof. Nicola Basilico, per tutto il tempo dedicatomi, per la cordialità, e per il supporto che non ha mai fatto mancare nelle nostre innumerevoli discussioni. Grazie al loro impegno e alla loro disponibilità, non solo porterò con me importanti lezioni per la futura vita professionale, ma anche un bel ricordo a conclusione della mia carriera accademica.

Cari mamma e papà: non so da dove cominciare a ringraziarvi. Potrei iniziare ringraziandovi per avermi sostenuto ogni giorno, anche quando forse non lo meritavo, e per avermi incoraggiato a seguire le mie inclinazioni. Potrei ringraziarvi per essere qui oggi, alla mia laurea, e lo faccio, con tutto il cuore, ma sento che è ancora troppo poco. Questa laurea è anche vostra, e spero oggi possiate essere felici. Un ringraziamento speciale va poi a mia nonna Rosa, il cui supporto ha per me un valore inestimabile.

Infine, ci tengo a ringraziare i miei amici, quelli di sempre e quelli incontrati lungo il percorso. Grazie per esserci sempre stati, nei momenti felici e in quelli bui. Grazie a chi ha condiviso con me sia spensierate ed interminabili chiacchierate fino all'alba, che mesti viaggi di ritorno dopo un'estenuante giornata di università. Grazie per essere le stelle che, nelle notti della vita, illuminano questo cammino.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

In Multi-Agent Pickup and Delivery (MAPD) [19], 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) [30], where paths for completing the assigned tasks must be computed. A particularly challenging feature of MAPD problems is their time-extension and dynamic nature that allows for new tasks to be submitted at any time and location in the environment.

Motivation

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; an example is the successful Amazon Robotics system employed by Amazon [43]. Beyond logistics, MAPD applications include also the coordination of teams of service robots [40] or fleets of autonomous cars, and the automated control of non-player characters in video games [23]. Others examples include autonomous aircraft-towing vehicles [25], autonomous intersection management [7], object-transportation robots [24], patrolling robots [1], search-and-rescue robots [14], and swarms of differential-drive robots and quadcopters [13].

Goal

Recently, the MAPF community has focused on *robustness* [2, 3, 19], 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 time-extended setting of MAPD has not been yet consistently studied. The goal of this thesis is to contribute to bridge this gap by studying the impact of delays in the MAPD problem and by proposing solutions applicable in real applications.

Contribution

In this thesis, the robustness of MAPD to the occurrence of *delays* is studied by defining a variant of the problem called *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. Three algorithms are devised to compute robust solutions for MAPD-d. The first one is based on a decentralized MAPD algorithm, Token Passing (TP), to which some recovery routines are added to provide replanning in case collisions caused by delays are detected. TP is able to solve well-formed MAPD problem instances [22], and it is shown that, under some assumptions, the introduction of delays in MAPD-d does not affect well-formedness. Then, two new algorithms are proposed, 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). These algorithms are compared by running experiments in simulated environments and the trade-offs offered by different levels of robustness are evaluated.

Outline

The contents of this thesis are organized in the following chapters.
In Chapter 2 the MAPF problem is introduced, its proprieties are analyzed along with various algorithms, either optimal and sub-optimal, used to solve it with various trade-offs. Then the discussion focuses on the limitations of MAPF, which led to the introductions of new variations of the problem. In particular, MAPD, a time-extended version of MAPF which manages to model closely what happens in real applications, is studied in detail (theoretical framework, algorithms, assumptions). Finally other research directions to bring MAPF closer to reality are presented, like considering kinematic constraints of robots or delays in execution.
Chapter 3 presents the problem of *MAPD-d* (*MAPD with delays*), which extends the concept of delays to the MAPD setting, and its theoretical properties are analyzed; then, a baseline algorithm, TP with recovery routines, is proposed to solve a relevant subset of MAPD-d problem instances.
Chapter 4 collects the main algorithmic contribution of this thesis, which consists if two algorithms, k-TP and p-TP, that, by considering the possibility of delays at planning time, allow to significantly reduce the impact of delays on execution. These algorithms follow a deterministic and probabilistic approach, respectively.
In Chapter 6 k-TP and p-TP are evaluated against the baseline algorithm TP with recovery routines in several realistic warehouse environments. The evaluation is performed by analyzing the trade-offs of different level of robustness and impact of the algorithms in reducing delay-induced replans in different settings with a varying number of robots, task frequencies and delays.
In Chapter 7 the results obtained in this thesis are summarized, underling strengths and limitations of the proposed methods. Finally, possible improvements and future directions for research are suggested.
Appendix A enriches the experimental evaluation performed in Chapter 6 with additional simulation settings and details about the standard deviations.
In Appendix B a brief user guide instructs the reader on how to use the given code to run the simulations and the experiments presented in this thesis.

Chapter 2

Preliminaries and Related Work

In the domain of Multi-Agent Systems many path finding problems have been formulated to solve variations of the general problem of coordinating multiple agents to reach a set of locations. The original and most studied problem, Multi-Agent Path Finding (MAPF), 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 (paths could be also required to minimize some cost function). Variations of this problem have been introduced to make it more general: Anonymous MAPF [31] allows the freedom of assigning targets to agents, combining the problem of path planning and target assignment; Target Assignment and Path Finding (TAPF) [21], still combines the problem of path planning and target assignment, but agents are divided in teams and an agent can only be assigned a target if that target is given to its team (allocation of targets to teams is predetermined and fixed). All these problems, however, are one-shot, meaning they relies on the notion that all tasks (start-target pairs of locations) are known a priori and the mission is ended when all targets are reached, which is not true in many applications. To solve this issue, the theoretical framework of Multi-Agent Pickup and Delivery (MAPD) [19] has been developed: MAPD combines path planning and target assignment but, differently from previous problems, it is long-term and dynamic, meaning that new tasks may enter at any time step. The one-shot assumption is not the only limitation of MAPF, but also other problems exists in the practical domain [38]; to address them, some solutions have been proposed in the

MAPF setting to consider the kinematic constraints of real robots [20] and to deal with imperfect execution caused by delays [2, 3].

Chapter Outline

The chapter is organized as follows. Section 2.1 introduces the problem of MAPF, its theoretical proprieties, and different algorithms, optimal and sub-optimal, used to solve it. Section 2.2 presents instead the problem of MAPD, its theoretical framework and the stat of the art regarding MAPD algorithms. Section 2.3 describes MAPF with kinematic constraints, one of the directions followed by research to extend MAPF to address problems of real applications. Finally, in Section 2.4 the different concepts of robustness in MAPF and MAPD setting are introduced.

2.1 MAPF

To better understand the topic and the next problem formulations, some basic definitions are needed.

Definition 2.1. An *undirected graph* $G = (V, E)$ is a mathematical structure that consists of a set of vertices, or nodes, V, and a set of edges $E \subseteq V^2$, which are non ordered pairs of vertices.

Definition 2.2. A vertex v_i is said to be neighbor, or *adjacent*, of a vertex v_j in some graph G if v_i is connected to v_j through an edge.

2.1.1 Problem Definition

A MAPF problem instance consists of:

- A given finite connected undirected graph $G = (V, E)$, whose vertices V correspond to locations and whose edges E correspond to connections between locations that the agents can move along.
- A given set of ℓ agents $\{a_i | i \in [\ell]\}$[1]. Each agent a_i has a start vertex $s_i \in V$ and a goal vertex $g_i \in V$ (that represents the preassigned target). All start vertices are pairwise different. All goal vertices are also pairwise different.

Time is discrete and starts from time step 0. At time step 0, each agent starts from its initial vertex.

[1] We let $[\ell]$ denote the positive integer set $\{1, ..., \ell\}$, representing the number of agents.

Definition 2.3. 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 conditions:

1. The agent starts at its start vertex, that is, $\pi_{i,0} = s_i$.
2. The agent ends at its goal vertex at the *arrival time* T_i, which is the minimal time step T_i such that, for all time steps $t = T_i, ..., \infty$, $\pi_{i,t} = g_i$.
3. 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 $\pi_{i,h+1} = \pi_i$.

Definition 2.4. A *vertex collision* is a tuple $\langle a_i, a_j, v, t \rangle$, where agents a_i and a_j occupy the same vertex $v = \pi_{i,t} = \pi_{j,t}$ at the same time step t.

Definition 2.5. An *edge collision* is a tuple $\langle a_i, a_j, u, v, t \rangle$, where agents a_i and a_j traverse the same edge (u, v), where $u = \pi_{i,t} = \pi_{j,t+1}$ and $v = \pi_{j,t} = \pi_{i,t+1}$, in opposite directions between time steps t and $t + 1$.

We call *plan* the 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.

Definition 2.6. The *makespan* $max_{i \in [\ell]} T_i$ of a MAPF plan is the maximum of the arrival times of all agents at their goal vertices.

Definition 2.7. The *flowtime* $\sum_{i \in [\ell]} T_i$ of a MAPF plan is the sum of the arrival times of all agents at their goal vertices.

The problem of MAPF is to find a solution with the smallest makespan or flowtime.

2.1.2 Some theoretical results for MAPF

While single agent path finding is tractable (Dijkstra [6]), MAPF is NP-hard to solve optimally for both makespan minimization [33] and flowtime minimization [45]. The optimal makespan and optimal flowtime of any MAPF problem instance are both bounded by $O(|V|^3)$ [46], however, as stated, it is NP-hard to find a solution with the minimum makespan.

2.1.3 MAPF Algorithms

MAPF algorithms can be divided in four classes based on their methodologies: reduction based, rule-based, search-based, and hierarchical algorithms;

algorithms differ for completeness (find a solution, if one exists, for all MAPF problem instances, for some MAPF problem instances on graphs with special properties, or offer no guarantee) and optimality (optimal, bounded-suboptimal, or suboptimal).

Reduction-Based Algorithms Reduction-Based Algorithms work by reducing MAPF to other well-studied combinatorial problems, such as Integer Linear Programming (ILP) [44], Boolean Satisfiability [34], and Answer Set Programming [8]. The idea behind these algorithms in to use variables and constraints to construct an explicit representation of the state space of a MAPF problem instance up to some value of the makespan, try to solve the problem for this value of makespan and, if no solution exists, increase the value. These algorithms are complete for solving MAPF and minimizing makespan. Variation exists to to solve MAPF with other objectives optimally [36] [44], bounded-suboptimally (in this case, the level of suboptimality can be chosen) [37], and suboptimally [35].
An example of optimal ILP-based algorithm is the one proposed in [44], where MAPF is reduced to the integer *multi commodity flow* problem on a time-expanded flow network and then this reduction is used to solve the problem optimally for makespan minimization with ILP techniques. Informally, the MAPF instance graph $G = (V, E)$ is expanded up to T time steps (a single vertex $v \in V$ is expanded to multiple vertices in the time-expanded flow network representing it at beginning and at the end of time step t, new vertices are connected by $\mathcal{E}$ edges with unit capacities), and for each agent a_i a supply of one at start vertex at time 0 and a demand of one at goal vertex at time t is set for commodity time i. An optimal solution can be found by starting with a lower bound on T and iteratively checking if, for increasing values of T, a feasible integer multi-commodity flow of ℓ (number of agents) units can be found in the corresponding T-step time-expanded flow network (which is an NP-hard problem), until an upper bound on T is reached. A MAPF solution with the smallest makespan corresponds to a feasible integer multi-commodity flow for the smallest value of T.

Rule-Based Algorithms Rule-based algorithms are based on a set of primitives used to choose agent actions in different situations. They usually guarantee completeness for a restricted class of MAPF problem instances. Rule-based algorithms are often very efficient since they limit choices to a predefined set of operations, but provide no optimality guarantee (lots of redundant actions in solution, very inefficient). One of the most known algorithm in this category is Push and Swap [18]; its extension Push and

Rotate [5] is complete for MAPF problem instances on graphs with at most $\ell = |V| - 2$ agents. Another algorithm, BIBOX [32], is complete for MAPF problem instances on bi-connected graphs with at most $\ell = |V| - 2$ agents. Finally, some algorithms like FAR [41] and MAPP [42] try to find patterns to combine paths of individual agents, using both primitive operations and search.

Search-Based Algorithms The problem associated with solving MAPF optimally is that the number of possible states of a MAPF problem instance is exponential in the number of agents: each joint state is given by the Cartesian product of the vertices of all the agents, and each joint action is the Cartesian product of the actions of all the agents. Search-based algorithms try to reduce the exponential size of the state space; they are often efficient but provide no optimality or even completeness guarantee. A technique used to achieve this result is *decoupled planning*: the main idea is to plan for each agent separately. Many of these algorithms are based on Prioritized Planning [4] that plans a path for a single agent at a time, selected by following predefined priorities. Cooperative A* and Hierarchical Cooperative A* (HCA*) [28] are prioritized planning algorithms that use a space-time A* search [10] to plan a path for one agent at a time avoiding collisions with agents that have already planned. An extension of HCA*, Windowed-HCA* [28], uses a limited time window when considering paths of other agents for planning and assigns priority for agents dynamically.

Hierarchical Algorithms Hierarchical algorithms are two-level algorithms: they decouple MAPF into one-shot single-agent path-planning problems on the low level and dynamically couple the resulting single-agent paths on the high level. They are complete and optimal for all MAPF problem instances, and the two most notable examples are *Conflict-Based Search* (CBS) [26] and *Increasing Cost Tree Search* (ICTS) [27].
CBS is a two-level complete and optimal MAPF algorithm that does not convert the problem into a single "joint agent" model. At the high level, a search is performed on a Conflict Tree (CT) which is a tree based on conflicts between individual agents. Each node in the CT represents a set of constraints on the motion of the agents. At the low level, fast single-agent searches are performed to satisfy the constraints imposed by the high level CT node. In many cases this two-level formulation enables CBS to examine fewer states than a classical algorithm based on A^* search over the joint space of agents while still maintaining optimality. CBS minimizes either the makespan or the flowtime.

ICTS is another example of complete and optimal two-level search algorithm. The high-level phase of ICTS searches the increasing cost tree for a set of costs (one cost per agent). The low-level phase of ICTS searches for a valid path for every agent that is constrained to have the same cost as given by the high-level phase. A compact data-structure called multi-value decision diagram (MDD) [29] is involved to store all single-agent paths of a certain length, for each agent. ICTS minimizes the flowtime.

2.2 MAPD

A MAPD problem [19] 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. Differently from the MAPF setting, agents are not preassigned to a task. Agents move in the environment represented by G along paths.

Definition 2.8. 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 2.9. 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 2.10. The *makespan* is the earliest time step when all tasks are completed.

2.2.1 Some theoretical results for MAPD

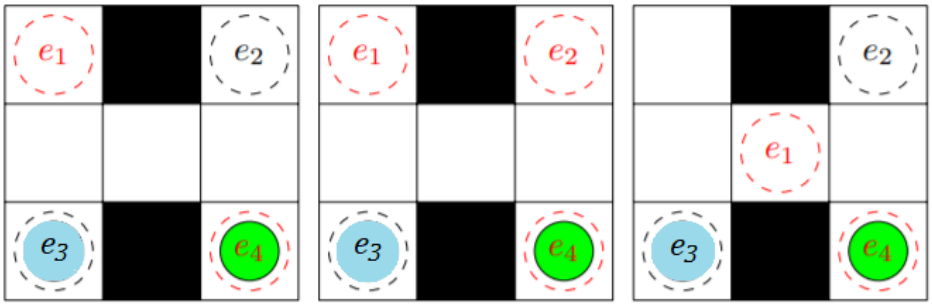

Figure 2.1: Three MAPD problem instances.

Not all MAPD problem instances are solvable. According to Ma et al. [22] 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.

Figure 2.1 shows three MAPD problem instances. Black cells are blocked. Blue and green circles are the initial vertices of agents. Red dashed circles are task endpoints. Black dashed circles are non-task endpoints. We assume each of the MAPD problem instances has finitely many tasks. The MAPD problem instance on the left is well-formed. The MAPD problem instance in the center is not well-formed because there are more agents than non-task endpoint (2 agents, 1 non-task endpoint). The MAPD problem instance on the right is not well-formed because all paths between endpoints e_3 and e_4 (or e_2 and e_3) traverse endpoint e_1.

Since MAPD is a generalization of MAPF, and MAPF is NP-hard to solve optimally (see Section 2.1.2), also MAPD is NP-hard to solve optimally, either using the service time or makespan objective function.

2.2.2 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. [22] 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.

Token Passing (TP, Algorithm 2.1) This decoupled algorithm 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 our 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 2.4) in order to avoid deadlocks and ensure long-term robustness (lines 14 and 16).

Token Passing with Task Swaps (TPTS) TPTS is a decoupled algorithm similar to TP except that its task set now contains all unexecuted tasks, rather than only unassigned tasks. This means that an agent with the token can assign itself not only a task that is not yet assigned to any agent but also a task that is already assigned to another agent as long as that agent is still moving to reach the pickup vertex of the task. This might be beneficial when the former agent can move to the pickup vertex of the task in fewer time steps than the latter agent. The latter agent is then no longer assigned the task and no longer needs to execute it. The former agent thus sends the token to the latter agent so that the latter agent can try to assign itself a new task.

Central A centralized algorithm that makes decisions for multiple agents at a time is also proposed in [22]. Similar to TPTS, *Central* allows agents that have just become free to consider not only unassigned tasks but also all unexecuted tasks, thus including the ones that have been assigned to agents, in the task set. Unlike TPTS, *Central* uses a centralized target-assignment algorithm, the Hungarian method [16], to assign (or reassign) tasks to agents and allows all free agents to consider tasks that have just been added to the system. It uses the centralized MAPF algorithm CBS to plan paths for multiple agents.

From experimental results [22], centralized algorithms offer better results in terms of service time and makespan, but require higher computational costs. A decentralized algorithm, Token Passing, proves instead most suitable for real-time long-term operations.

2.3 Kinematic Constraints

So far we have considered agents as abstract entities; in reality they are often robots and, as such, they are subject to physical constraints like minimum turning radius, maximum velocity, maximum acceleration, etc.; to introduce these physical limitations in the formulation of the problem, some solutions

Algorithm 2.1: Token Passing

```
1  /* system executes now */;
2  initialize token with the (trivial) path ⟨loc(a_i)⟩ 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 𝒯;
5      while agent a_i exists that requests token do
6          /* system sends token to a_i and a_i executes now */;
7          𝒯' ← {τ_j ∈ 𝒯 | no path in token ends in s_j or in g_j};
8          if 𝒯' ≠ {} then
9              τ ← arg min_{τ_j ∈ 𝒯'} h(loc(a_i), s_j);
10             assign a_i to τ;
11             remove τ from 𝒯;
12             update a_i's path in token with the path returned by
                 PathPlanner(a_i, τ, token);
13         else if no task τ_j ∈ 𝒯 exists with g_j = loc(a_i) then
14             update a_i's path in token with the path ⟨loc(a_i)⟩;
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
```

have been proposed.

Hoenig et al. [12] presented MAPF-POST, an approach that makes use of a simple temporal network to post-process a MAPF plan in polynomial time to create a plan-execution schedule that works on non-holonomic (not all degrees of freedom can be controlled at the same time) robots, takes their maximum translational and rotational velocities into account, provides a guaranteed safety distance between them, and exploits slack (defined as the difference of the latest and earliest entry times of locations) to absorb imperfect plan executions and avoid time intensive replanning in many cases. A similar approach was later followed by Hoenig et al. [11]: their solution exploits a particular type of graph, called Action Dependency Graph, that captures the action precedence relationships of a MAPF solution and can be

used to enforce these relationships on real robots with higher-order dynamics. More recently, Ma et al. [20] proposed, for the MAPD problem, an improved version of TP, called TP-SIPPwRT, keeping into account kinematic constraints. TP is made more effective using a novel combinatorial search algorithm, called Safe Interval Path Planning with Reservation Table (SIPPwRT) for single-agent path planning. SIPPwRT uses an advanced data structure that allows for fast updates and lookups of the current paths of all agents in an online setting. The resulting MAPD algorithm TP-SIPPwRT takes kinematic constraints of real robots into account directly during planning, computes continuous agent movements with given velocities that work on non-holonomic robots rather than discrete agent movements with uniform velocity, and is complete for well-formed MAPD instances.

2.4 Robustness

In the context of MAPF, a robust solution allows to follow the same 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. [3], is defined as follows.

Definition 2.11. 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 [2] 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 2.12. 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 it is not possible to solve all MAPD problem instances. As we have discussed in Section 2.2.1 some characteristics of the problem environment, summarized under the term *well-formedness*, are a sufficient condition to enable *long-term robustness*, meaning that we have the guarantee to complete a finite number of tasks in a finite time. It is important to notice that here the term robustness, despite coming form literature, can be misleading: it actually describes a feasibility condition.

In this theis, 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. We also 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.

Chapter 3

MAPD with Delays

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 (encoders, gyroscopes, accelerometers) used for localization and coordination [15]. 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 [20], 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 [9].

Chapter Outline

The chapter is organized as follows. Section 3.1 presents the problem of *MAPD with delays* (*MAPD-d*), which introduces delays in a long term setting. Section 3.2 is devoted to the analysis of the property of well-formedness in MAPD-d. Finally, Section 3.3 introduces a baseline algorithm to solve MAPD-d.

3.1 MAPD-d

We define the problem of *MAPD with delays* (*MAPD-d*) as a MAPD problem (see Section 2.2) 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 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 2.4). The only possible exception is when an agent cannot move anymore (namely when $e_{i,t+1} = e_{i,t}$

[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.

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 2.4, we have:

(4) there is not any agent that belongs to $\mathcal{D}(t)$ for $t \geq T$.

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 2.2.2), 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 3.1) will be a baseline for experimentally evaluating the algorithms we propose in the next section. In addition to the other information (Section 2.2.2), 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).

Figure 3.1 shows hot TP with recovery routines work. The setting is a grid environment, with two agents and two tasks, at different time steps. Initially, the agents plan their paths without collisions (top). At time steps 6 and 7, a_2 is delayed (middle) and at time step 7 a collision is detected in the token. Then, a_1 regains the token and replans (bottom).

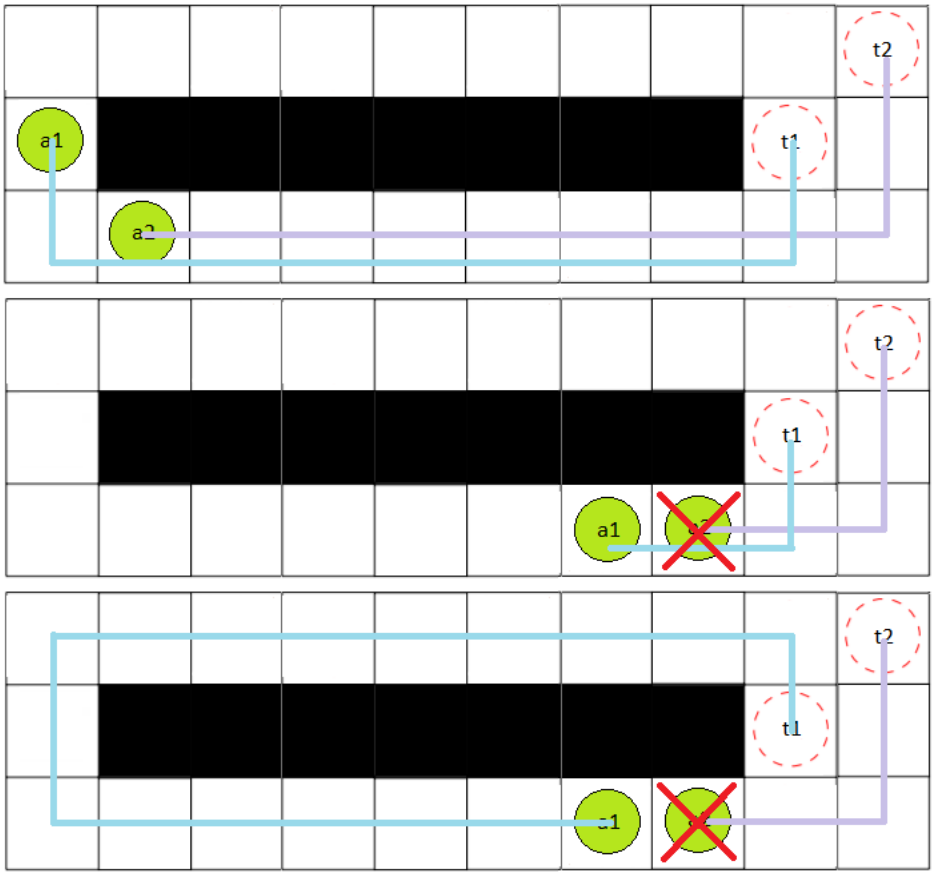

Figure 3.1: Example of how TP with recovery routines works.

Algorithm 3.1: TP with recovery routines

```
1  /* system executes now */;
2  initialize token with the (trivial) path ⟨loc(a_i)⟩ for each agent a_i;
3  while true do
4      add new tasks, if any, to the task set 𝒯;
5      ℛ ← CheckCollisions(token);
6      foreach agent a_i in ℛ do
7          retrieve task τ assigned to a_i;
8          π_i ← 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 2.1 (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
```

Chapter 4

MAPD-d Algorithms

As we have discussed in Section 3.1, TP with recovery routines just reacts to the occurrence of delays, ensuring that long-term robustness is preserved. The algorithms proposed here, instead, plans considering that delays may occur, reducing the need of replanning during execution.

Chapter Outline

The chapter is organized as follows. Section 4.1 presents k-TP, an algorithm able to produce robust solutions to the MAPD-d problem following a deterministic approach. Section 4.2 presents p-TP, an algorithm able to produce robust solutions to the MAPD-d problem following a probabilistic approach.

4.1 k-TP Algorithm

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, shown as Algorithm 4.1, does. 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, lines 18 and 24) 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 Chapter 6 for numerical data supporting these claims).

An alternative approach is to resort to the concept of p-robustness (Chapter 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 we use Markov chains, a tool typically employed to model the future states of systems when transitions are defined in term of probability [17]. 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

Algorithm 4.1: k-TP

```
1  /* system executes now */;
2  initialize token with the (trivial) path ⟨loc(a_i)⟩ for each agent a_i;
3  while true do
4      add new tasks, if any, to the task set 𝒯;
5      ℛ ← CheckCollisions(token);
6      foreach agent a_i in ℛ do
7          proceed like in Algorithm 3.1 (lines 7 - 13);
8      end
9      while agent a_i exists that requests token do
10         /* system sends token to a_i and a_i executes now */;
11         𝒯' ← {τ_j ∈ 𝒯 | no path in token ends in s_j or in g_j};
12         if 𝒯' ≠ {} then
13             τ ← arg min_{τ_j ∈ 𝒯'} h(loc(a_i), s_j);
14             assign a_i to τ;
15             remove τ from 𝒯;
16             π_i ← PathPlanner(a_i, τ, token);
17             if π_i is not null then
18                 update token with k-extension(π_i, k);
19         else if no task τ_j ∈ 𝒯 exists with g_j = loc(a_i) then
20             update a_i's path in token with the path ⟨loc(a_i)⟩;
21         else
22             π_i ← Idle(a_i, token);
23             if π_i is not null then
24                 update token with k-extension(π_i, k);
25         end
26         /* a_i returns token to system, which executes now */;
27     end
28     agents move along their paths in token for one time step (or stay
         at their current position if delayed);
29     /* system advances to the next time step*/;
30 end
```

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.

From Algorithm 4.2, we can see that 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 (lines 20 and 33). Let us show the procedure in detail. The properties of Markov chains [17] 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 (lines 20 and 32), 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.

Algorithm 4.2: p-TP

```
1  /* system executes now */;
2  initialize token with the (trivial) path ⟨loc(a_i)⟩ for each agent a_i;
3  while true do
4      add new tasks, if any, to the task set 𝒯;
5      ℛ ← CheckCollisions(token);
6      foreach agent a_i in ℛ do
7          proceed like in Algorithm 3.1 (lines 7 - 13);
8      end
9      while agent a_i exists that requests token do
10         /* system sends token to a_i and a_i executes now */;
11         𝒯' ← {τ_j ∈ 𝒯 | no path in token ends in s_j or in g_j};
12         if 𝒯' ≠ {} then
13             τ ← arg min_{τ_j ∈ 𝒯'} h(loc(a_i), s_j);
14             assign a_i to τ;
15             remove τ from 𝒯;
16             j ← 0;
17             while j ¡ itermax do
18                 π_i ← PathPlanner(a_i, τ, token);
19                 cprob_{π_i} ← MarkovChain(π_i, token);
20                 if cprob_{π_i} < p then
21                     update a_i's path in token with π_i;
22                     break
23                 j ← j + 1;
24             end
25         else if no task τ_j ∈ 𝒯 exists with g_j = loc(a_i) then
26             update a_i's path in token with the path ⟨loc(a_i)⟩;
27         else
28             j ← 0;
29             while j ¡ itermax do
30                 π_i ← Idle(a_i, token);
31                 cprob_{π_i} ← MarkovChain(π_i, token);
32                 if cprob_{π_i} < p then
33                     update a_i's path in token with π_i;
34                     break
35                 j ← j + 1;
36             end
37         end
38         /* a_i returns token and system executes now */;
39     end
40     agents move along their paths in token for one time step (or stay
         at their current position if delayed);
41     /* system advances to the next time step*/;
42 end
```

Chapter 5

Implementation Details

To compare the proposed algorithms and evaluate the trade-offs offered by the different levels of robustness, we developed a simulation pipeline and algorithm implementations using Python. All the code used in this thesis can be found at `https://github.com/Lodz97/Multi-Agent_Pickup_and_Delivery.git`, MIT Licence.

Chapter Outline

The chapter is organized as follows. Section 5.1 describes how environments are represented. Section 5.2 presents the core code structure of both the simulation pipeline and the algorithm implementations. Section 5.3 focuses on how visualization is performed.

5.1 Environments

All the experiments have been run in several simulated warehouse environments created following typical warehouse representations described in the literature [22, 43]. An environment is completely defined by a yaml file which contains information regarding environment size, obstacles, starting positions of the agents, task and non-task endpoints. Information regarding tasks (pickup location, goal location, time of appearance) and delays (agent involved, time of appearance) is also present in the yaml file, and can either be fixed or change in different simulations. In this latter case, a yaml file containing just the number of tasks to insert and the number of delays per agent to insert is processed and transformed into a new yaml file containing complete information about tasks (inserted randomly following a Poisson

distribution [39]) and delays (inserted randomly in a given interval). An example of a yaml file where tasks and dealys are not fixed is presented in Listing 5.1.

```yaml
agents:
    -  start: [0, 1]
       name: agent0
    -  start: [0, 5]
       name: agent1
map:
    dimensions: [10, 10]
    obstacles:
    - !!python/tuple [3, 2]
    - !!python/tuple [4, 2]
    - !!python/tuple [5, 2]
    - !!python/tuple [6, 2]
    non_task_endpoints:
    - !!python/tuple [0, 1]
    - !!python/tuple [0, 5]
    start_locations:
    -  [4, 1]
    -  [6, 1]
    -  [8, 1]
    goal_locations:
    -  [0, 0]
    -  [0, 2]
    -  [0, 3]
n_tasks: 50
task_freq: 1
n_delays_per_agent: 10
```

Listing 5.1: Example of a yaml file.

5.2 Simulation Code Structure

Our simulation pipeline relies on the interaction of few key objects.
In Figure 5.1 we can see a representation of such pipeline. Green rectangles represents the input and the output of the simulation. Blue rectangles constitute the core components of the simulation. The orange rectangular outline symbolizes the loop performed by advancing the simulation of one

time step until all the tasks are completed. Finally, the purple rectangular outline represents the loop performed at any time step on all the agents that can be assigned to a task or that need a replan.

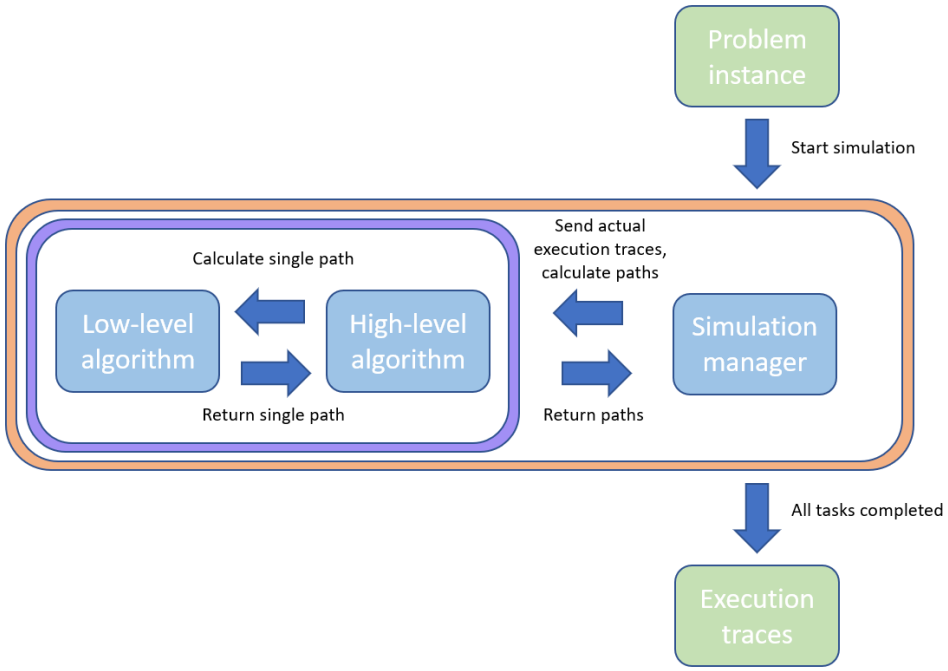

Figure 5.1: High level view of the simulation pipeline.

Simulation Manger Class This class manages the simulation. It receives in input a yaml files containing all the information regarding obstacles, agents, tasks, delays, and starts the simulation. At every time step, the simulation class interacts with the high-level algorithm class and receives the planned paths for each agent. Then, following this computed path, agents are moved forward by one step. Agents may not move if delays occur or an imminent collision caused by a delay is detected (simulating local obstacle avoidance). Information about these agents whose execution trace (see Chapter 3) differs from the planned path is passed to the high-level algorithm class at the next time step to possibly replan paths. From experiments, we found that replanning only when obstacles are detected on the path (for agents blocked by a simple delay just reuse the already computed path shifted forward by one time step) gives the best results, especially in terms of algorithm running time. When all the tasks are completed, the simulation class outputs a yaml file containing the execution traces of all the agents.

High-Level Algorithms Class This class can be initialized with different parameters to implement k-TP or p-TP with different levels of robustness, and can even be initialized without robustness guarantees to implement TP with recovery routines. The main duty of this class is to manage the token object, which contains information about tasks, agents, paths. In principle the token should be a block of shared memory requested by one agent at a time; to keep implementation simple we did not explicitly model an agent object receiving the token and the returning it to the system, but we just keep track in the token of free agents or agents needing replan. When an agent is selected, he is assigned to the closest task according to Manhattan distance (in case of a replan task, assignment is already done) and then the low-level algorithm classes are called to compute a path from the start location of the agent to the pickup vertex of task (in case of a replan, if an agent has already reached the pickup vertex this first path is empty) and then a path from the the pickup vertex to the goal vertex of the task. These paths are then combined together and added to the token. In the case of p-TP, before adding a path to the token, another class calculates the Markov chains and collision probability, and if the probability is above threshold the path is rejected. If one of this two paths is not found for several time steps, deadlock recovery procedures like planning random movements around the starting position of the agent are performed. When all the agents are not free or there are no more available tasks in the token, the control is returned to the simulation class.
In this class we can set the various parameters for k-TP and p-TP, like the level of robustness, represented by k and p, respectively, that present the trade-offs that will be discussed in Chapter 6. Another parameter for p-TP is the number of times a new path can be recalculated if the one calculated before exceeds the probability threshold; in our experiments we set it to 1 since it's difficult to impose constraints such that the paths computed next will have a strictly lower collision probability. For example, avoiding locations where the Markov chains suggest conflicts are more likely, ensures that, if a path still exists, it will be different form the one computed previously, but gives no guarantees on the new collision probability. Finally, for p-TP, we can also set p_d, which is the probability of an agent of being delayed at any time step.

Low-Level Algorithm Classes All the proposed algorithms (TP with recovery routines, k-TP, and p-TP), employ as low-level search algorithm an implementation of space-time A*. The original algorithm (`https://github.com/atb033/multi_agent_path_planning.git`, MIT Licence) has

been modified to take into account dynamic obstacles, which are the agents that have already planned. In our implementation the high-level class does not call directly space-time A*, but another object implementing CBS (see Chapter 2). Then the CBS algorithm is not actually used, since space-time A* is called at the root node of CBS and no other node is expanded (we cannot modify the paths of agents that have already planned).
The space-time A* class has an important parameter, the maximum number of states to be explored. In this variation of A*, states not only differ for their location, but also for the time step at which they are visited; this means that a maximum number of states to be visited is needed in case no solution exists, otherwise the algorithm will keep exploring forward in time. For example, a solution may not exist in a particular time step because the robustness requirements are too tight at that time step. This parameter should be tuned according to the specific environment: if it is set too low, no solution will be found even if one exists; if it is set too high, lots of time will be wasted searching for a solution when none exists.

5.3 Visualization

Visualization is performed through a script that exploits the animation capabilities of the Python matplotlib package. Information found in the environment yaml file and in the simulation output yaml file are used to draw the environment, the tasks, and move the agents according to the time step of their execution traces.
Parameters allow to change the speed of the visualization and to save the output in various video formats.

Chapter 6

Experimental Results

Since the problem of delays in the long term setting of MAPD has never been studied (to the best of our knowledge), no specific algorithms exists for a direct comparison. This is why a baseline algorithm, TP with recovery routines, has been developed trying to resemble as close as possible a current state of the art MAPD algorithm, TP. Morover, since the focus is on real applications, the experimental evaluation of the proposed algorithms is performed in simulated warehouse environments.

Chapter Outline

The chapter is organized as follows. Section 6.1 describes the hardware on which simulations are run, as well as the differences among simulation settings. Section 6.2 presents the results of the conducted experiments, with a particular focus on the trade-offs offered by different levels of robustness, impact of the number of delays, and scalability.

6.1 Setting

The experiments are conducted on a laptop equipped with 1 x CPU Intel(R) Core(TM) i7 8750H @ 3.30 GHz (6 cores, 12 threads, 9 MB cache) and 16 GB RAM.
We test 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 (Figure 6.1), and a large one, 25×17, with 12 and 24 agents (Figure 6.2). (Environments of similar size have been used in [22].) 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 [39]. 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 in this section results for all values of λ). The complete results can be found in Appendix A. At the beginning, the agents are located at the endpoints selected for well-formedness (Section 2.4).

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 (as said, 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). Implementation details about these algorithms are discussed in Section 5.

6.2 Results

Results relative to small warehouse are shown in Tables 6.1 and 6.2 and those relative to large warehouse are shown in Tables 6.3 and 6.4. To keep readability, we do not report here the standard deviation in tables. All the standard deviation tables can be found in Appendix A. 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 is the basic implementation as in Algorithm 3.1) 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.

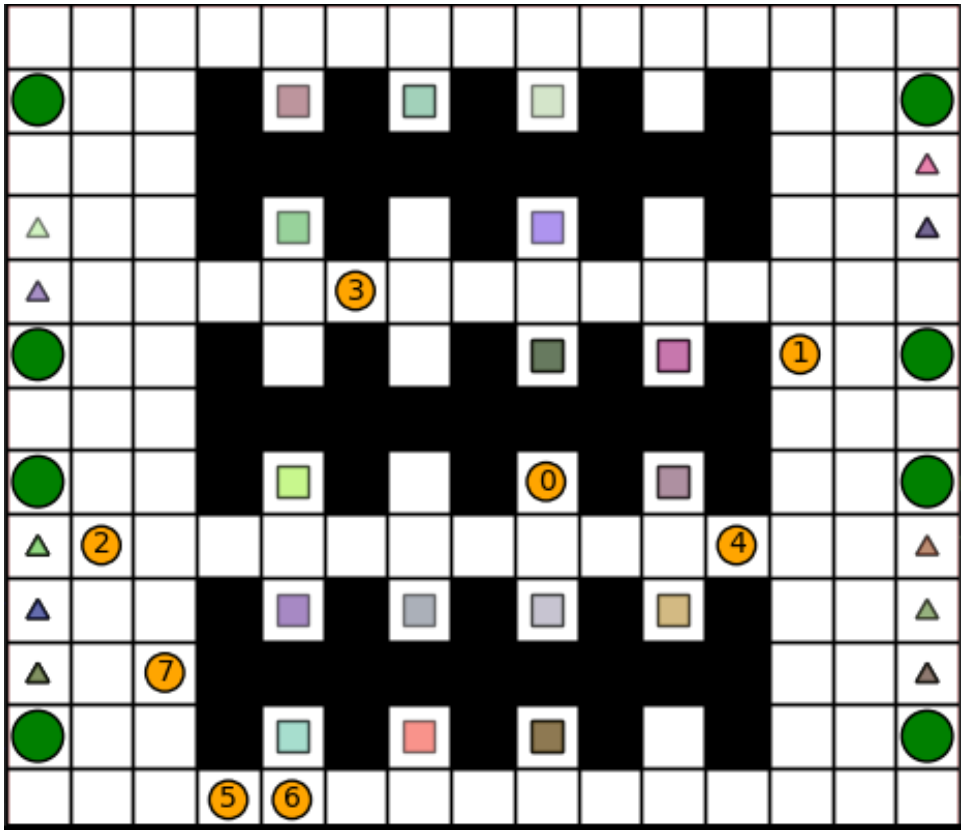

Figure 6.1: Small warehouse with 8 agents. Black cells are obstacles. Colored squares are task pickup vertices, triangles are task goal vertices. Green circles are endpoints.

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 6.4, 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

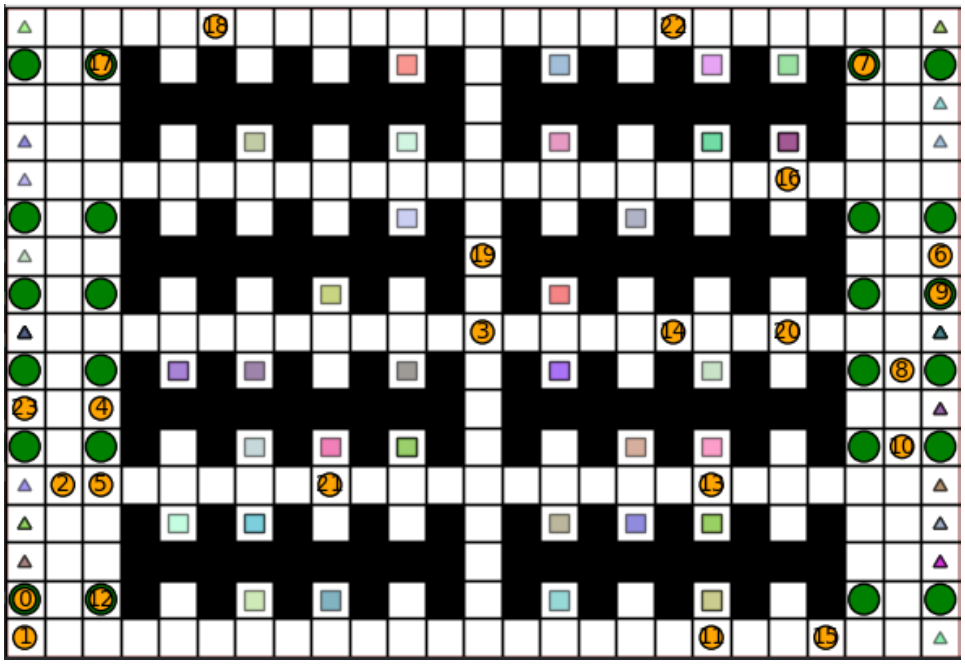

Figure 6.2: Large warehouse with 24 agents. Black cells are obstacles. Colored squares are task pickup vertices, triangles are task goal vertices. Green circles are endpoints.

Table 6.1: Results of experiments in small warehouse with task frequency $\lambda = 0.5$ and 10 delays per agent

k or p		$\ell = 4$			$\ell = 8$		
		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

Table 6.2: Results of experiments in small warehouse with task frequency $\lambda = 3$ and 10 delays per agent

		$\ell = 4$			$\ell = 8$		
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 6.3: Results of experiments in large warehouse with task frequency $\lambda = 0.5$ and 10 delays per agent

		$\ell = 12$			$\ell = 24$		
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

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 6.1), and decreasing p may sometimes

Table 6.4: Results of experiments in large warehouse with task frequency $\lambda = 3$ and 10 delays per agent

		$\ell = 12$			$\ell = 24$		
k or p		tot. cost	# replans	runtime	tot. cost	# replans	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

lead to relevant increases in costs. This suggests that our implementation of p-TP has margins for improvement: if 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.

6.2.1 Effect of the Number of Delays

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. We show results for task frequency $\lambda = 1$ in Tables 6.5 and 6.6. 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.

Table 6.5: Results of experiments in small warehouse with task frequency $\lambda = 1$ and 50 delays per agent

		$\ell = 4$			$n\ell = 8$		
k or p		tot. cost	# replans	runtime	tot. cost	# replans	runtime
k-TP	0	1679.44	24.52	1.34	2267.36	44.27	4.37
	1	1696.4	8.77	**0.87**	2269.52	18.35	3.21
	2	1711.16	3.88	1.03	2239.28	8.28	**3.18**
	3	1782.08	1.27	1.46	2429.84	4.7	3.66
	4	1881.68	**0.53**	1.74	2462.08	**2.17**	4.63
p-TP, $p_d = .1$	1	1679.44	24.52	1.71	2267.36	44.27	5.64
	0.5	1659.16	16.18	1.64	2268.64	28.85	7.74
	0.25	1723.96	11.83	2.46	2359.76	15.42	8.03
	0.1	1794.0	6.82	2.81	2402.08	8.39	8.48
	0.05	1835.68	5.68	2.91	2472.24	5.77	7.38
p-TP, $p_d = .02$	0.5	**1629.16**	18.47	1.46	**2175.68**	32.15	5.44
	0.25	1670.08	16.62	1.69	2282.32	28.41	6.49
	0.1	1722.2	12.5	2.26	2325.76	17.75	7.14
	0.05	1759.8	7.83	2.41	2328.4	9.76	6.12

6.2.2 Scalability in Larger Environments

Finally, we run simulations in a even larger warehouse 4-connected grid environment of size 25×37, with 52 agents (Figure 6.3), $\lambda = 1$, 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.

Table 6.6: Results of experiments in large warehouse with task frequency $\lambda = 1$ and 50 delays per agent

k or p		$\ell = 12$			$\ell = 24$		
		tot. cost	# replans	runtime	tot. cost	#replans	runtime
k-TP	0	**3726.12**	42.8	4.83	7613.52	66.53	12.66
	1	3771.12	18.79	**4.7**	**7295.52**	26.76	**12.26**
	2	3853.56	9.43	5.98	7598.4	18.29	16.56
	3	3961.2	4.7	7.8	8000.4	7.61	22.42
	4	4151.16	**2.98**	11.26	8068.8	**4.7**	28.49
p-TP, $p_d = .1$	1	**3726.12**	42.8	9.71	7613.52	66.53	19.36
	0.5	3962.88	28.99	19.26	7665.36	38.59	48.64
	0.25	4055.88	17.06	23.28	8195.76	22.81	62.19
	0.1	4260.36	10.16	25.25	8834.4	13.19	63.77
	0.05	4456.08	7.23	25.21	8812.8	9.48	56.24
p-TP, $p_d = .02$	0.5	3887.28	35.19	9.66	7686.24	49.95	37.02
	0.25	3913.08	27.45	11.6	8154.96	40.87	56.83
	0.1	3971.16	15.9	13.6	8229.84	24.73	54.53
	0.05	4204.68	15.67	15.11	8298.96	20.17	55.37

Table 6.7: Results of experiments in larger warehouse (25x37) with task frequency $\lambda = 1$ and 10 delays per agent (100 tasks)

k or p		$\ell = 52$		
		tot. cost	# replans	runtime
k-TP	0	**19262.88**	58.98	**40.06**
	1	20146.36	16.16	51.54
	2	20246.72	3.7	95.96
	3	21718.84	**1.16**	204.25
	4	22564.88	1.87	454.91
p-TP, $p_d = .1$	1	**19262.88**	58.98	53.99
	0.5	21005.92	32.97	462.18
	0.25	23568.48	19.35	475.23
	0.1	26770.12	12.98	505.72
	0.05	28459.08	12.17	525.51
p-TP, $p_d = .02$	0.5	20742.8	44.63	337.64
	0.25	21577.92	28.63	414.17
	0.1	23047.96	18.57	454.63
	0.05	23861.76	17.36	464.08

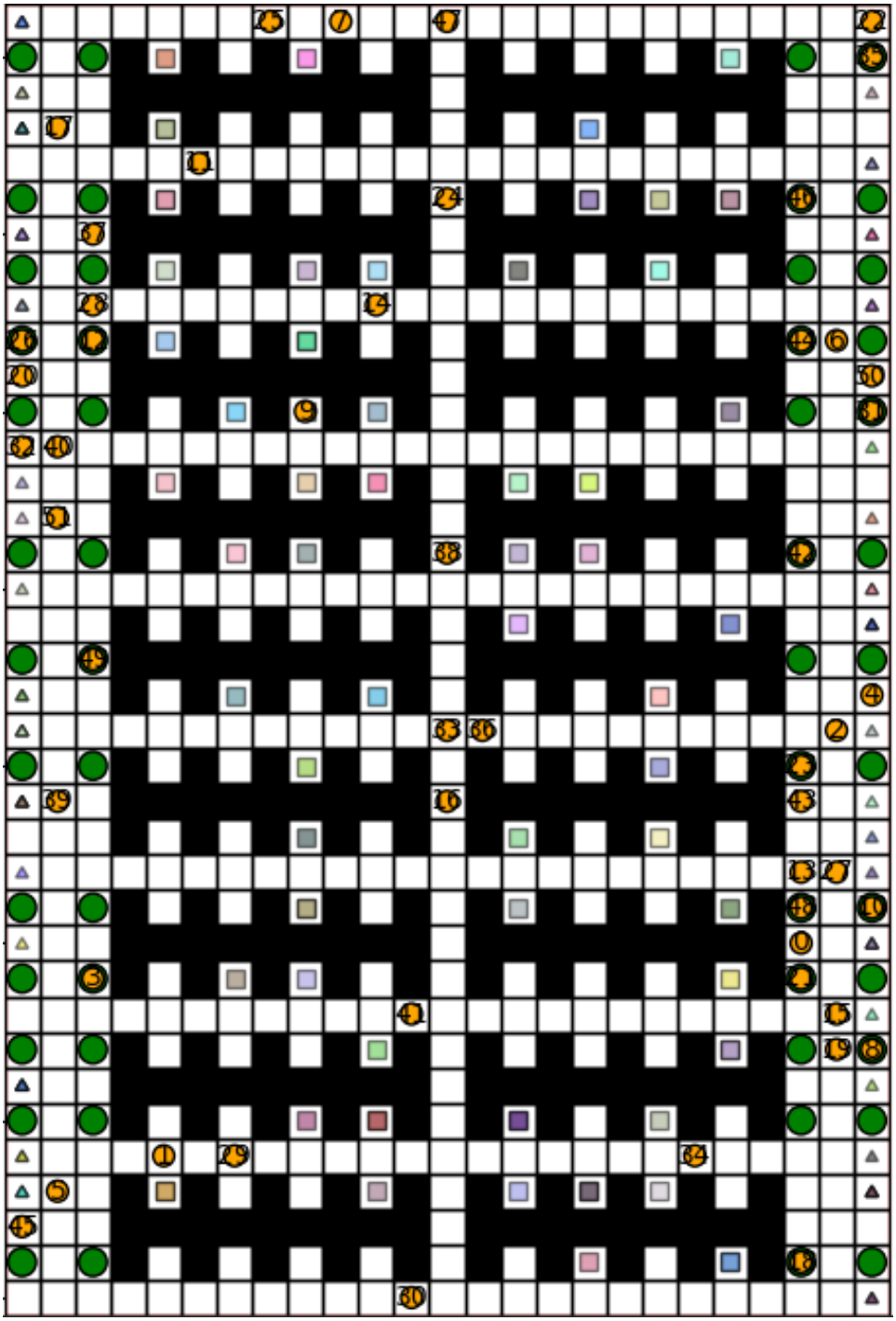

Figure 6.3: Larger warehouse with 52 agents. Black cells are obstacles. Colored squares are task pickup vertices, triangles are task goal vertices. Green circles are endpoints.

Chapter 7

Conclusions

This thesis provided a theoretical, algorithmic and experimental contribution to field of Multi-Agent Path Planning. In recent years, the research community has focused on bringing theoretical models closer to applications, due to the growing industrial deployment of autonomous multi-agent systems. This is why great importance has been given to robustness, understood as a property of solutions that can withstand real-world-induced relaxations of some idealistic assumptions made by the models. Differently from the one-shot MAPF problem, robustness in the long-term setting of Multi-Agent Pickup and Delivery (MAPD) has not been yet consistently studied, and the proposed work aims to bridge this gap.

We introduced a variation of the 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 presented two algorithms, k-TP and p-TP, to solve well-formed MAPD-d problem instances with deterministic and probabilistic robustness guarantees, respectively.

Experimentally, these algorithms have been compared 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 Chapter 6 and the experimental testing of our algorithms in more real-world settings. Another possible direction for future research is considering an adversarial environment: delays are no more random, but are selected by an adversary to cause the greatest possible damage.

Appendix A

Tables of Results

In this appendix, we report all the results of the experiments not presented in Chapter 6, including the standard deviations. The data appearing in this appendix are coherent with the considerations made in Chapter 6.

A.1 Low Number of Delays

In this section we report all the experiments done introducing 10 delays per agent and not already presented in Chapter 6.

Table A.1: Results of experiments in small warehouse with task frequency $\lambda = 1$ *and* 10 *delays per agent*

k or p		n. of agents = 4			n. of agents = 8		
		tot. cost	n. replans	runtime	tot. cost	n. replans	runtime
k-TP	0	**1404.4**	7.64	**0.64**	1815.68	18.12	2.69
	1	1478.76	1.38	0.73	**1780.88**	3.98	**1.71**
	2	1552.68	0.2	1.0	1911.76	0.75	1.94
	3	1644.96	0.01	1.39	2008.64	0.12	3.13
	4	1716.8	**0.0**	1.77	2204.16	**0.02**	4.44
p-TP, $p_d = .1$	1	**1404.4**	7.64	0.9	1815.68	18.12	3.6
	0.5	1482.64	7.44	1.59	1871.36	12.03	5.06
	0.25	1549.72	4.27	2.48	1934.72	6.47	5.97
	0.1	1628.16	3.07	2.86	2043.76	3.08	6.16
	0.05	1671.0	2.46	3.08	2125.28	2.86	6.29
p-TP, $p_d = .02$	0.5	1463.88	6.68	1.22	1818.72	13.65	4.49
	0.25	1474.44	6.53	1.58	1858.56	10.36	4.24
	0.1	1537.48	4.24	2.14	1947.44	6.26	5.17
	0.05	1603.36	2.96	2.49	1958.16	3.62	5.05

Table A.2: Results of experiments in large warehouse with task frequency $\lambda = 1$ *and* 10 *delays per agent*

k or p		n. of agents = 12			n. of agents = 24		
		tot. cost	n. replans	runtime	tot. cost	n. replans	runtime
k-TP	0	**3221.04**	15.4	**2.81**	**6175.44**	26.63	**6.87**
	1	3303.24	4.49	3.41	6350.16	7.11	8.03
	2	3416.4	1.28	4.98	6555.84	1.8	11.23
	3	3383.88	0.18	7.16	6651.36	0.52	17.59
	4	3616.08	**0.04**	10.62	7240.56	**0.21**	26.43
p-TP, $p_d = .1$	1	**3221.04**	15.4	3.97	**6175.44**	26.63	9.09
	0.5	3358.32	12.91	9.38	6970.8	19.06	50.29
	0.25	3572.52	6.14	14.74	6775.68	4.43	42.64
	0.1	3815.88	4.72	16.88	7823.28	6.23	57.98
	0.05	3999.12	3.81	18.68	7960.8	3.73	50.8
p-TP, $p_d = .02$	0.5	3270.0	12.7	6.76	6675.84	22.12	30.16
	0.25	3442.32	10.48	9.49	6703.44	14.5	39.05
	0.1	3549.48	6.96	11.76	6995.76	10.44	45.36
	0.05	3623.88	6.63	13.22	7257.84	7.94	47.98

A.2 Low Number of Delays (Standard Deviations)

In this section we report all the standard deviations of the experiments done introducing 10 delays per agent.

Table A.3: Results of experiments in small warehouse with task frequency $\lambda = 0.5$ and 10 delays per agent (standard deviation)

k or p		$\ell = 12$			$\ell = 24$		
		tot. cost	# replans	runtime	tot. cost	# replans	runtime
k-TP	0	154.85	3.08	1.2	245.69	5.25	2.79
	1	124.39	1.43	0.69	245.43	2.53	3.34
	2	129.05	0.3	0.29	216.49	1.05	0.59
	3	107.42	0.1	0.5	242.7	0.32	1.15
	4	116.49	0.0	0.59	189.75	0.31	1.82
p-TP, $p_d = .1$	1	154.85	3.08	1.08	245.69	5.25	2.54
	0.5	129.1	2.69	1.05	240.61	4.86	3.54
	0.25	114.61	2.54	1.12	239.22	2.54	1.88
	0.1	134.02	1.87	0.94	235.89	2.14	2.28
	0.05	135.75	1.75	0.9	232.27	1.9	1.53
p-TP, $p_d = .02$	0.5	113.53	2.95	0.59	219.49	4.89	2.84
	0.25	115.99	2.71	0.43	239.62	4.47	2.92
	0.1	111.49	2.37	0.72	264.44	2.98	3.44
	0.05	121.11	2.22	0.88	237.56	2.36	1.73

Table A.4: Results of experiments in small warehouse with task frequency $\lambda = 1$ and 10 delays per agent (standard deviation)

		$\ell = 12$			$\ell = 24$		
k or p		tot. cost	# replans	runtime	tot. cost	# replans	runtime
k-TP	0	123.4	3.1	0.6	279.17	5.08	3.03
	1	133.31	1.54	0.15	190.58	2.46	2.41
	2	127.83	0.57	0.22	231.43	0.95	0.65
	3	133.14	0.1	0.32	208.94	0.38	0.92
	4	112.4	0.0	0.44	227.09	0.14	1.53
p-TP, $p_d = .1$	1	123.4	3.1	0.62	279.17	5.08	3.39
	0.5	139.93	2.69	0.78	259.2	4.36	3.99
	0.25	139.21	2.36	0.72	212.32	2.74	2.39
	0.1	138.63	2.0	0.78	212.73	1.99	2.34
	0.05	134.69	1.72	0.84	233.56	1.97	1.86
p-TP, $p_d = .02$	0.5	138.25	2.95	0.47	217.81	4.85	4.13
	0.25	127.48	2.9	0.76	271.18	3.82	3.07
	0.1	125.6	2.22	0.57	219.64	3.25	3.11
	0.05	131.42	2.32	0.95	210.85	2.24	1.52

Table A.5: Results of experiments in small warehouse with task frequency $\lambda = 3$ and 10 delays per agent (standard deviation)

		$\ell = 12$			$\ell = 24$		
k or p		tot. cost	# replans	runtime	tot. cost	# replans	runtime
k-TP	0	131.39	3.03	0.47	227.38	4.4	2.2
	1	126.53	1.31	0.59	207.95	2.57	2.38
	2	115.12	0.51	0.2	189.49	0.81	0.47
	3	94.93	0.1	0.27	214.41	0.45	1.58
	4	114.66	0.0	0.44	218.44	0.1	1.48
p-TP, $p_d = .1$	1	131.39	3.03	0.48	227.38	4.4	2.26
	0.5	106.88	2.37	0.53	232.65	3.61	4.17
	0.25	123.01	2.42	0.67	227.57	3.24	3.14
	0.1	134.56	2.18	0.78	233.48	2.57	1.72
	0.05	135.58	1.82	0.8	210.63	2.03	1.9
p-TP, $p_d = .02$	0.5	113.34	2.64	0.53	222.56	4.57	2.82
	0.25	122.1	2.82	0.49	251.94	3.86	2.53
	0.1	126.63	2.27	0.54	213.7	3.2	2.04
	0.05	126.7	1.89	0.76	196.12	2.82	2.52

Table A.6: Results of experiments in large warehouse with task frequency $\lambda = 0.5$ and 10 delays per agent (standard deviation)

		$\ell = 12$			$\ell = 24$		
k or p		tot. cost	# replans	runtime	tot. cost	# replans	runtime
k-TP	0	565.06	4.84	2.95	2011.61	6.9	33.43
	1	475.11	2.43	1.85	840.26	3.07	2.15
	2	474.37	1.23	1.3	897.15	1.73	3.23
	3	493.82	0.69	2.7	1010.84	0.77	5.75
	4	569.3	0.1	3.61	920.66	0.5	7.81
p-TP, $p_d = .1$	1	565.06	4.84	3.05	2011.61	6.9	35.8
	0.5	583.34	3.72	6.03	1069.48	6.47	21.58
	0.25	503.23	2.45	4.89	1916.98	4.33	40.99
	0.1	595.78	2.47	4.95	929.51	2.84	15.4
	0.05	509.47	1.83	5.78	1191.88	2.34	19.69
p-TP, $p_d = .02$	0.5	540.14	4.55	3.46	1976.54	6.31	23.46
	0.25	495.1	4.09	3.97	1017.08	5.63	21.84
	0.1	581.78	3.05	3.36	998.73	4.11	27.03
	0.05	474.97	2.96	4.87	804.32	3.56	26.41

Table A.7: Results of experiments in large warehouse with task frequency $\lambda = 1$ and 10 delays per agent (standard deviation)

		$\ell = 12$			$\ell = 24$		
k or p		tot. cost	# replans	runtime	tot. cost	# replans	runtime
k-TP	0	498.53	4.52	2.65	950.47	7.35	7.68
	1	453.91	2.86	2.55	991.87	3.51	5.51
	2	586.78	1.58	2.75	1001.62	1.66	9.99
	3	487.87	0.46	1.77	1006.99	0.89	5.68
	4	507.35	0.2	3.39	1068.54	0.68	21.0
p-TP, $p_d = .1$	1	498.53	4.52	2.74	950.47	7.35	8.08
	0.5	503.16	4.59	3.75	3099.79	6.68	54.35
	0.25	484.63	2.94	4.98	1077.99	2.91	11.9
	0.1	586.82	2.23	4.76	1115.29	3.93	56.32
	0.05	537.63	2.27	7.69	919.49	2.4	15.28
p-TP, $p_d = .02$	0.5	431.05	3.89	3.21	2020.55	6.98	31.11
	0.25	467.49	3.56	2.93	2057.01	5.43	36.51
	0.1	564.88	3.4	3.11	911.47	5.16	34.86
	0.05	498.76	3.3	4.25	1939.77	3.82	49.05

Table A.8: Results of experiments in large warehouse with task frequency $\lambda = 3$ and 10 delays per agent (standard deviation)

k or p		$\ell = 12$			$\ell = 24$		
		tot. cost	# replans	runtime	tot. cost	# replans	runtime
k-TP	0	454.4	5.39	3.3	968.96	8.48	8.29
	1	534.22	2.41	2.28	845.12	3.63	7.59
	2	559.54	1.35	1.24	919.81	1.59	3.03
	3	441.98	0.54	2.95	1044.64	0.85	9.55
	4	459.93	0.2	4.1	877.5	0.88	28.57
p-TP, $p_d = .1$	1	454.4	5.39	3.5	968.96	8.48	8.8
	0.5	493.97	3.78	5.01	1050.84	5.98	35.6
	0.25	455.77	3.31	6.15	1111.31	4.01	38.41
	0.1	505.76	2.45	4.14	979.04	3.91	15.67
	0.05	575.43	2.38	4.76	1192.25	2.32	10.54
p-TP, $p_d = .02$	0.5	446.21	4.79	3.99	930.86	6.61	13.76
	0.25	595.32	4.37	3.83	1075.07	5.93	48.45
	0.1	489.59	2.82	4.95	2588.22	3.61	48.88
	0.05	566.01	3.03	5.13	1030.78	3.78	25.26

Table A.9: Results of experiments in larger warehouse (25x37) with task frequency $\lambda = 1$ and 10 delays per agent (100 tasks, standard deviation)

k or p		$\ell = 52$		
		tot. cost	# replans	runtime
k-TP	0	2798.6	11.29	29.34
	1	3000.2	5.21	20.54
	2	2959.17	2.08	25.97
	3	4344.1	1.7	100.25
	4	2847.61	3.13	368.13
p-TP, $p_d = .1$	1	2798.6	11.29	30.28
	0.5	4466.36	10.83	322.6
	0.25	3889.32	5.74	302.5
	0.1	4969.69	6.86	427.1
	0.05	5107.51	7.33	430.99
p-TP, $p_d = .02$	0.5	5225.75	12.58	242.56
	0.25	5118.99	11.33	266.14
	0.1	3899.47	7.29	307.4
	0.05	3048.33	8.2	406.58

A.3 High Number of Delays

In this section we report all the experiments done introducing 50 delays per agent and not already presented in Chapter 6.

Table A.10: Results of experiments in small warehouse with task frequency $\lambda = 0.5$ *and* 50 *delays per agent*

k or p		n. of agents = 12			n. of agents = 24		
		tot. cost	n. replans	runtime	tot. cost	n. replans	runtime
k-TP	0	**1673.6**	23.32	0.97	2322.64	43.41	3.67
	1	1714.0	8.52	**0.89**	2317.76	17.9	**3.13**
	2	1716.8	2.95	0.99	2352.24	9.05	**3.13**
	3	1785.64	0.92	1.31	2396.4	3.9	3.6
	4	1879.76	**0.48**	1.58	2511.84	**1.9**	4.54
p-TP, $p_d = .1$	1	**1673.6**	23.32	1.28	2322.64	43.41	4.55
	0.5	1719.2	17.39	1.75	2224.24	23.35	6.22
	0.25	1743.4	11.26	2.55	2423.76	14.81	7.85
	0.1	1816.6	7.23	2.77	2438.08	8.03	7.14
	0.05	1860.72	5.72	2.96	2525.36	6.18	7.35
p-TP, $p_d = .02$	0.5	1682.56	19.55	1.57	2319.2	32.48	7.32
	0.25	1716.56	17.27	1.88	**2221.12**	27.37	6.26
	0.1	1739.88	11.51	2.35	2256.96	15.49	6.58
	0.05	1788.28	8.19	2.65	2361.12	10.38	6.91

Table A.11: Results of experiments in small warehouse with task frequency $\lambda = 3$ and 50 delays per agent

		n. of agents = 12			n. of agents = 24		
k or p		tot. cost	n. replans	runtime	tot. cost	n. replans	runtime
k-TP	0	1625.44	24.04	0.96	2181.76	44.93	4.47
	1	**1618.16**	7.39	**0.79**	**2148.24**	18.11	**2.66**
	2	1682.08	3.23	0.96	2214.08	8.51	3.24
	3	1777.88	1.12	1.34	2320.16	3.88	3.71
	4	1866.0	**0.38**	1.6	2454.96	**2.34**	4.82
p-TP, $p_d = .1$	1	1625.44	24.04	1.38	2181.76	44.93	5.77
	0.5	1631.56	16.82	1.61	2204.16	27.45	7.04
	0.25	1734.76	12.49	2.51	2256.72	13.61	7.05
	0.1	1812.08	7.37	2.88	2347.76	8.36	7.38
	0.05	1837.12	5.92	3.01	2411.68	6.31	6.67
p-TP, $p_d = .02$	0.5	1693.44	21.13	1.69	2276.24	38.04	7.17
	0.25	1696.8	18.94	1.78	2255.44	29.35	6.58
	0.1	1716.28	12.56	2.32	2245.76	14.73	6.43
	0.05	1757.32	8.22	2.49	2327.84	11.22	6.88

Table A.12: Results of experiments in large warehouse with task frequency $\lambda = 0.5$ and 50 delays per agent

		n. of agents = 12			n. of agents = 24		
k or p		tot. cost	n. replans	runtime	tot. cost	n. replans	runtime
k-TP	0	4056.96	44.36	4.16	7785.36	64.16	**9.24**
	1	3879.6	16.54	**3.9**	7726.56	27.39	9.9
	2	3987.96	9.34	5.43	7919.28	15.62	13.99
	3	4087.56	5.61	7.2	7817.04	6.78	16.88
	4	4230.12	**2.83**	10.23	8136.48	**4.2**	24.46
p-TP, $p_d = .1$	1	4056.96	44.36	5.64	7785.36	64.16	12.57
	0.5	4042.68	27.49	11.95	8142.24	40.4	54.63
	0.25	4196.28	17.35	18.16	8849.76	21.67	62.63
	0.1	4393.2	10.34	18.26	9144.72	11.8	63.6
	0.05	4439.04	6.38	18.26	9117.36	9.22	62.05
p-TP, $p_d = .02$	0.5	**3840.48**	33.14	9.43	**7511.52**	45.29	42.98
	0.25	4083.0	26.76	11.42	8036.16	36.08	56.78
	0.1	4179.48	18.43	15.16	8525.76	24.17	72.01
	0.05	4239.72	13.69	16.56	8665.68	19.42	71.32

Table A.13: Results of experiments in large warehouse with task frequency $\lambda = 3$ and 50 delays per agent

k or p		n. of agents = 12			n. of agents = 24		
		tot. cost	n. replans	runtime	tot. cost	n. replans	runtime
k-TP	0	**3638.64**	38.87	**3.81**	7364.64	71.11	**13.7**
	1	3688.68	18.31	4.21	7635.84	33.04	13.89
	2	3781.44	9.01	5.59	7497.84	17.62	16.02
	3	3796.32	4.69	7.14	7864.56	10.67	25.12
	4	4003.08	**2.77**	10.25	7848.72	**5.49**	28.56
p-TP, $p_d = .1$	1	**3638.64**	38.87	5.25	7364.64	71.11	18.69
	0.5	3927.48	29.95	10.84	7758.0	41.43	54.17
	0.25	3967.68	16.8	15.58	8510.88	23.81	63.46
	0.1	4239.84	10.41	16.29	8532.48	13.29	65.92
	0.05	4351.08	7.95	16.9	8894.88	10.23	53.14
p-TP, $p_d = .02$	0.5	3761.28	34.9	9.11	7174.8	42.09	36.52
	0.25	3682.8	26.57	11.15	**6664.56**	22.49	33.21
	0.1	3893.88	15.89	13.0	8116.8	27.09	60.01
	0.05	4090.92	14.51	15.68	8319.6	22.43	58.28

A.4 High Number of Delays (Standard Deviations)

In this section we report all the standard deviations of the experiments done introducing 50 delays per agent.

Table A.14: Results of experiments in small warehouse with task frequency $\lambda = 0.5$ and 50 delays per agent (standard deviation)

		$\ell = 12$			$\ell = 24$		
k or p		tot. cost	# replans	runtime	tot. cost	# replans	runtime
k-TP	0	153.74	6.48	0.93	272.33	9.65	3.36
	1	165.77	3.34	0.54	296.61	5.45	3.36
	2	125.88	1.98	0.22	257.04	3.74	3.45
	3	129.92	1.08	0.32	261.23	2.41	2.75
	4	127.58	0.82	0.4	303.76	1.64	3.23
p-TP, $p_d = .1$	1	153.74	6.48	0.95	272.33	9.65	3.43
	0.5	140.66	4.29	0.77	278.62	6.91	3.56
	0.25	127.42	3.72	0.82	261.88	4.71	5.54
	0.1	163.94	3.08	0.67	234.72	3.56	3.37
	0.05	160.99	2.48	0.78	284.68	2.75	3.82
p-TP, $p_d = .02$	0.5	153.94	5.98	1.02	628.84	8.99	19.86
	0.25	159.82	4.85	0.95	232.12	7.2	3.75
	0.1	165.62	3.71	0.73	274.35	5.19	3.13
	0.05	126.83	3.45	0.86	237.78	3.85	3.1

Table A.15: Results of experiments in small warehouse with task frequency $\lambda = 1$ and 50 delays per agent (standard deviation)

		$\ell = 12$			$\ell = 24$		
k or p		tot. cost	# replans	runtime	tot. cost	# replans	runtime
k-TP	0	153.32	6.69	1.4	290.6	11.21	3.77
	1	144.03	3.26	0.55	300.03	5.69	3.48
	2	123.4	2.22	0.25	254.46	4.03	3.43
	3	125.77	1.52	0.72	245.01	2.59	2.78
	4	121.98	0.95	0.37	251.31	1.84	1.85
p-TP, $p_d = .1$	1	153.32	6.69	1.5	290.6	11.21	4.19
	0.5	157.48	4.49	0.73	267.01	7.58	4.88
	0.25	139.41	3.38	0.8	305.8	4.6	4.89
	0.1	151.91	2.98	0.91	252.67	3.26	5.09
	0.05	164.58	2.79	0.82	307.56	2.99	2.81
p-TP, $p_d = .02$	0.5	153.6	5.36	1.09	290.54	8.13	3.44
	0.25	134.54	4.61	0.72	255.19	8.51	4.2
	0.1	132.19	4.27	0.68	246.76	5.37	4.98
	0.05	130.28	3.46	0.62	283.12	4.06	2.54

Table A.16: Results of experiments in small warehouse with task frequency $\lambda = 3$ and 50 delays per agent (standard deviation)

		$\ell = 12$			$\ell = 24$		
k or p		tot. cost	# replans	runtime	tot. cost	# replans	runtime
k-TP	0	121.03	6.66	0.92	226.18	12.56	4.2
	1	148.42	2.88	0.51	235.5	6.13	3.06
	2	143.52	2.11	0.46	270.59	3.51	4.31
	3	137.23	1.27	1.57	270.97	2.36	4.17
	4	133.76	0.66	0.69	262.4	2.05	4.1
p-TP, $p_d = .1$	1	121.03	6.66	1.03	226.18	12.56	4.89
	0.5	138.19	4.73	0.68	246.5	7.49	4.58
	0.25	142.27	3.6	0.96	342.19	4.9	3.64
	0.1	136.53	3.01	0.74	249.39	3.42	4.68
	0.05	160.29	2.99	0.93	279.79	2.99	2.41
p-TP, $p_d = .02$	0.5	165.6	5.39	1.14	262.68	9.8	5.07
	0.25	133.96	5.84	0.87	248.14	8.17	4.17
	0.1	138.3	4.2	0.78	298.22	5.18	3.5
	0.05	131.03	3.38	0.72	280.29	4.19	4.06

Table A.17: Results of experiments in large warehouse with task frequency $\lambda = 0.5$ and 50 delays per agent (standard deviation)

		$\ell = 12$			$\ell = 24$		
k or p		tot. cost	# replans	runtime	tot. cost	# replans	runtime
k-TP	0	582.36	9.93	3.88	946.75	15.53	7.34
	1	547.2	4.91	2.18	1032.03	7.49	6.15
	2	555.84	3.44	2.71	1876.55	4.78	21.9
	3	569.64	2.96	3.6	1055.55	3.66	12.06
	4	490.53	1.85	4.04	1055.35	2.85	13.51
p-TP, $p_d = .1$	1	582.36	9.93	4.08	946.75	15.53	8.34
	0.5	671.03	6.6	5.34	1999.36	9.31	74.42
	0.25	494.52	5.53	7.05	2365.54	7.27	33.51
	0.1	500.15	3.73	6.17	2848.57	4.49	49.33
	0.05	543.8	3.22	4.98	1110.67	4.14	46.02
p-TP, $p_d = .02$	0.5	478.52	8.4	6.08	1110.35	10.99	41.31
	0.25	574.7	7.55	5.05	1181.37	8.76	30.66
	0.1	514.48	5.6	5.07	1918.32	7.46	57.26
	0.05	526.11	4.73	8.75	1885.51	7.02	52.72

Table A.18: Results of experiments in large warehouse with task frequency $\lambda = 1$ and 50 delays per agent (standard deviation)

		$\ell = 12$			$\ell = 24$		
k or p		tot. cost	# replans	runtime	tot. cost	# replans	runtime
k-TP	0	472.19	10.72	4.0	963.5	16.07	11.95
	1	540.22	6.22	3.11	1080.33	6.92	10.84
	2	553.33	3.64	4.13	947.18	5.5	11.15
	3	567.68	2.42	3.47	1188.09	3.71	20.89
	4	553.38	1.77	5.64	1311.17	2.55	13.13
p-TP, $p_d = .1$	1	472.19	10.72	7.05	963.5	16.07	16.0
	0.5	590.07	7.0	9.75	1465.16	9.25	40.65
	0.25	496.55	4.54	15.33	1076.06	6.17	38.21
	0.1	589.33	3.43	9.58	1079.22	5.07	33.89
	0.05	637.05	3.26	10.69	1056.31	4.11	34.83
p-TP, $p_d = .02$	0.5	541.5	8.61	5.83	1160.0	12.41	22.28
	0.25	567.13	6.99	5.43	2042.04	11.89	63.58
	0.1	511.58	5.14	5.4	1906.14	6.31	34.39
	0.05	559.58	4.9	6.24	1075.56	6.04	51.5

Table A.19: Results of experiments in large warehouse with task frequency $\lambda = 3$ and 50 delays per agent (standard deviation)

k or p		$\ell = 12$			$\ell = 24$		
		tot. cost	# replans	runtime	tot. cost	# replans	runtime
k-TP	0	704.05	9.58	4.2	1231.87	17.03	15.43
	1	551.08	5.38	3.96	1288.84	9.5	10.63
	2	524.29	3.84	4.3	965.55	5.57	8.26
	3	541.62	2.7	3.03	1047.69	4.28	31.45
	4	648.21	2.02	5.17	1041.45	2.65	11.49
p-TP, $p_d = .1$	1	704.05	9.58	4.54	1231.87	17.03	17.75
	0.5	624.71	7.04	5.18	1922.1	10.59	39.26
	0.25	569.87	5.25	6.09	2960.43	6.44	53.52
	0.1	611.36	4.38	5.21	1948.69	4.7	53.12
	0.05	543.02	3.16	4.38	1057.41	3.71	18.7
p-TP, $p_d = .02$	0.5	538.18	8.12	4.91	2059.79	9.86	37.26
	0.25	548.05	7.27	5.3	1140.45	6.35	15.22
	0.1	540.39	4.32	4.98	1143.88	7.41	40.38
	0.05	590.78	4.89	8.65	1107.27	7.26	48.75

Appendix B

User Guide

In this appendix, we describe how to run a demonstration of our algorithms and how use the code to replicate the results obtained in Chapter 6.

B.1 Requirements

The code has been tested with Python version 3.6.9.
All the packages needed to run the code can be found in the file *requirements.txt*. To install all the requirements, run the following code:

```
pip install -r requirements.txt
```

B.2 Run One Simulation

Before running the simulation, an environment can be chosen. The *Environments* folder contains different predefined environments, some of which have been used for the experiments in Chapter 6. There exists two main types of environments, differentiated by the presence or absence of the sub-string *_random* in the name. The "random" environments just specify the number of tasks and delay per agents, while the others present fixed tasks and delays (to use these environments, a special simulation parameter must be set). To change the simulation environment, open the file *config.json* and modify the parameter *"input_name"* with the file name of the desired environment.
Then, to start the simulation, the script *demo.py* can be run. The script accepts various command line arguments:

- *-k*: an integer ($k \geq 0$) which represents the robustness parameter for k-TP;
- *-p*: a float ($0 \leq p \leq 1$) which represents the robustness parameter (probability threshold) for p-TP;
- *-pd*: a float ($0 \leq p_d \leq 1$, default 0.02) which represents the expected probability of an agent of being delayed at any time step (used in p-TP);
- *-p_iter*: an integer ($p\_iter \geq 1$, default 1) which represents the number of times a new path can be recalculated if the one calculated before exceeds the probability threshold (used in p-TP);
- *-a_star_max_iter*: an integer ($a\_star\_max\_iter \geq 1$, default 5000) which represents the maximum number of states explored by the low-level algorithm state-space A*;
- *-slow_factor*: an integer ($slow\_factor \geq 1$, default 1) which allows to slow down the visualization;
- *-not_rand*: this parameter needs to be present if the input environment is not randomized.

Note that if the script is run without both k and p, it becomes TP with recovery routines. If the visualization does not start after the end of the simulation, the error could be related to the non-GUI back-end of matplotlib. To resolve this problem, restart the simulation after the following code has been run:

```
sudo apt-get install python3-tk
```

In the following we present some example runs.
Run TP with recovery routines:

```
python3 demo.py
```

Run k-TP with $k = 2$ and slower visualization:

```
python3 demo.py -k 2 -slow_factor 3
```

Run p-TP with $p = 0.6$, $p_d = 0.05$ in a non-randomized environment:

```
python3 demo.py -p 0.6 -pd 0.05 -not_rand
```

B.3 Run Multiple Experiments

To run multiple experiments and collect all the statistics, a specific script, *run_all_experiments_new.py*, can be used. This script contains a list of experiments (easy to modify and extend) that will be run exploiting multithreading; after all the experiments terminate a json file with the results will be saved in the *Experiments* folder. The script can be run with the following code:

```
python3 -m Utils.run_all_experiments_new
```

To see the results plotted as box plots, the script *plot_experiments* can be used. First, modify the file *config.json* changing the parameter *"experiments_name"* with the name of the experiments file that has just been created. Then run the visualization tool with the following code:

```
python3 -m Utils.plot_experiments
```

When a plot is closed, the next one will appear until the end of the experiments.

Bibliography

[1] Noa Agmon, Daniel Urieli, and Peter Stone. Multiagent patrol generalized to complex environmental conditions. In *Proc. AAAI*, 2011.

[2] 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.

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

[4] Maren Bennewitz, Wolfram Burgard, and Sebastian Thrun. Finding and optimizing solvable priority schemes for decoupled path planning techniques for teams of mobile robots. *Robot. Auton. Syst.*, 41(2-3):89–99, 2002.

[5] Boris de Wilde, Adriaan ter Mors, and Cees Witteveen. Push and rotate: cooperative multi-agent path planning. In *Proc AAMAS*, pages 87–94, 2013.

[6] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[7] Kurt M. Dresner and Peter Stone. A multiagent approach to autonomous intersection management. *J. Artif. Intell. Res.*, 31:591–656, 2008.

[8] Esra Erdem, Doga Gizem Kisa, Umut Öztok, and Peter Schüller. A general formal framework for pathfinding problems with multiple agents. In *Proc. AAAI*, pages 290–296, 2013.

[9] 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.

[10] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Sci. Cybern.*, 4(2):100–107, 1968.

[11] Wolfgang Hönig, Scott Kiesel, Andrew Tinka, Joseph W. Durham, and Nora Ayanian. Persistent and robust execution of mapf schedules in warehouses. *IEEE Robot. Autom. Lett.*, 4:1125–1131, 2019.

[12] Wolfgang Hönig, T. K. Satish Kumar, Liron Cohen, Hang Ma, Hong Xu, Nora Ayanian, and Sven Koenig. Multi-agent path finding with kinematic constraints. In *Proc. ICAPS*, pages 477–485, 2016.

[13] Wolfgang Hönig, T. K. Satish Kumar, Hang Ma, Sven Koenig, and Nora Ayanian. Formation change for robot groups in occluded environments. In *Proc IROS*, pages 4836–4842, 2016.

[14] J.S. Jennings, G. Whelan, and W.F. Evans. Cooperative search and rescue with a team of mobile robots. In *Proc. ICAR*, pages 193–200, 1997.

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

[16] Harold. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly, 2*, pages 83–97, 1955.

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

[18] Ryan Luna and Kostas E. Bekris. Push and swap: Fast cooperative path-finding with completeness guarantees. In *Proc IJCAI*, pages 294–300, 2011.

[19] 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.

[20] 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.

[21] Hang Ma and Sven Koenig. Optimal target assignment and path finding for teams of agents. In *Proc. AAMAS*, pages 1144–1152, 2016.

[22] 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*, pages 837–845, 2017.

[23] 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.

[24] Maja J. Mataric, Martin Nilsson, and Kristian T. Simsarin. Cooperative multi-robot box-pushing. In *Proc. IROS*, pages 556–561, 1995.

[25] Robert Morris, Corina S. Pasareanu, Kasper Søe Luckow, Waqar Malik, Hang Ma, T. K. Satish Kumar, and Sven Koenig. Planning, scheduling and monitoring for airport surface operations. In *Plan. for Hybrid Syst. Workshop*, volume WS-16-12, 2016.

[26] Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant. Conflict-based search for optimal multi-agent pathfinding. *Artif. Intell.*, 219:40–66, 2015.

[27] Guni Sharon, Roni Stern, Meir Goldenberg, and Ariel Felner. The increasing cost tree search for optimal multi-agent pathfinding. In *Proc. IJCAI*, pages 662–667, 2011.

[28] David Silver. Cooperative pathfinding. In *Proc. AIIDE*, pages 117–122, 2005.

[29] Arvind Srinivasan, Timothy Kam, Sharad Malik, and Robert K. Brayton. Algorithms for discrete function manipulation. In *Proc. ICCAD*, pages 92–95, 1990.

[30] 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.

[31] 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.

[32] Pavel Surynek. A novel approach to path planning for multiple robots in bi-connected graphs. In *Proc. ICRA*, pages 3613–3619, 2009.

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

[34] Pavel Surynek. Towards optimal cooperative path planning in hard setups through satisfiability solving. In *Proc. PRICAI*, pages 564–576, 2012.

[35] Pavel Surynek. Reduced time-expansion graphs and goal decomposition for solving cooperative path finding sub-optimally. In *Proc. IJCAI*, pages 1916–1922, 2015.

[36] Pavel Surynek, Ariel Felner, Roni Stern, and Eli Boyarski. Efficient SAT approach to multi-agent path finding under the sum of costs objective. In *Proc. ECAI*, pages 810–818, 2016.

[37] Pavel Surynek, Ariel Felner, Roni Stern, and Eli Boyarski. Modifying optimal SAT-based approach to multi-agent path-finding problem to suboptimal variants. In *Proc. SOCS*, pages 169–170, 2017.

[38] Jirí Svancara. Bringing multi-agent path finding closer to reality. In *Proc. AAMAS*, pages 1784–1785, 2018.

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

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

[41] Ko-Hsin Cindy Wang and Adi Botea. Fast and memory-efficient multi-agent pathfinding. In *Proc. ICAPS*, pages 380–387, 2008.

[42] Ko-Hsin Cindy Wang and Adi Botea. MAPP: a scalable multi-agent path planning algorithm with tractability and completeness guarantees. *CoRR*, abs/1401.3905, 2014.

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

[44] Jingjin Yu and Steven M. LaValle. Multi-agent path planning and network flow. In *Proc. WAFR*, volume 86, pages 157–173, 2012.

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

[46] Jingjin Yu and Daniela Rus. Pebble motion on graphs with rotations: Efficient feasibility tests and planning algorithms. In *Proc. WAFR*, volume 107, pages 729–746, 2014.