



VRIJE
UNIVERSITEIT
BRUSSEL



Bachelor thesis submitted in partial fulfillment of the requirements for the degree
of Bachelor in de Ingenieurswetenschappen: Computerwetenschappen

A STUDY OF MULTI-AGENT PICKUP AND DELIVERY APPROACHES FOR AUTOMATED WAREHOUSE ENVIRONMENTS

Hicham Azmani

June 2021

Promotor: prof. dr. Ann Nowé

Advisors: Andries Rosseau, Roxana Rădulescu

Sciences and bio-engineering sciences



VRIJE
UNIVERSITEIT
BRUSSEL



Bachelorproef ingediend met het oog op het behalen van de graad van
Bachelor of Science in de Ingenieurswetenschappen: Computerwetenschappen

EEN STUDIE VAN HET MULTI-AGENT PICKUP EN DELIVERY PROBLEEM VOOR GEAUTOMATISEERDE PAKHUIZEN

Hicham Azmani

Juni 2021

Promotor: prof. dr. Ann Nowé

Begeleiders: Andries Rosseau, Roxana Rădulescu

Wetenschappen en bio-ingenieurswetenschappen

Abstract

Recent developments in artificial intelligence have brought enormous advancements in the industry. One of these advances was the increasing popularity of automated warehouses. The problem of managing the robots that operate these warehouses is known as the Multi-Agent Pickup and Delivery problem. While there exist many approaches to the problem there is a lack of tools that visualize the problem as well as these approaches.

In this paper, we analyze the Multi-Agent Pickup and Delivery problem (MAPD) as well as suitable approaches for solving it. We implement, visualize and compare a selection of approaches. We implement two decentralized approaches for the online variant of the MAPD problem, the Token Passing algorithm (TP) and the Token Passing with Task Swapping algorithm (TPTS), as well as a centralized algorithm for the offline variant of the MAPD problem: the Task Allocation and Prioritized Pathplanning algorithm (TA-Prioritized). We then visualize these algorithms using the Unity game engine and the ML-agents platform. Finally, we compare the approaches in various simulated warehouse environments and conclude that the TA-Prioritized algorithm outperforms the TP and TPTS algorithms and should be always be used over the TP and TPTS algorithms in situations where it can be used. In situations where the TA-Prioritized algorithm can not be used, the TP algorithm typically outperforms the TPTS algorithm in smaller, more congested, warehouse environments while the TPTS algorithm outperforms the TP algorithm in larger warehouse environments. We also conclude that the inclusion of agent rotations to these algorithms significantly decreases the performances of the algorithms and that in the case of the TA-Prioritized algorithm makes it practically unusable for more than 20 agents which is why we proposed a modified version of the TA-Prioritized algorithm which supports up to 50 agents.

Samenvatting

Recente ontwikkelingen op het gebied van artificiële intelligentie hebben geleid tot enorme vooruitgang in de industriesector. Eén van de domeinen waar er veel vooruitgang is geboekt is het domein van geautomatiseerde pakhuisen. Het probleem van het beheer van de robots die deze magazijnen bedienen, staat bekend als het Multi-Agent Pickup and Delivery-probleem. Hoewel er veel oplossingen voor dit probleem bestaan, is er een gebrek aan hulpmiddelen die zowel het probleem als deze oplossingen visualiseren.

In dit artikel analyseren we het Multi-Agent Pickup and Delivery-Probleem (MAPD) en geschikte manieren om dit probleem op te lossen. We implementeren, visualiseren en vergelijken een selectie van algoritmes. We implementeren twee gedecentraliseerde oplossingen voor de online variant van het MAPD probleem, het Token Passing algoritme (TP) en het Token Passing with Task Swapping algoritme (TPTS), en één gecentraliseerde oplossing voor de offline variant van het MAPD probleem, het Task Allocation and Prioritized Pathplanning algoritme (TA-Prioritized). We visualiseren deze algoritmes gebruikmakende van de Unity game engine en het ML-agents platform.

Ten slotte vergelijken we de algoritmes in verschillende gesimuleerde pakhuisomgevingen en concluderen we dat het TA-Prioritized algoritme over het algemeen beter presteert dan het TP en het TPTS algoritme en dus overall waar mogelijk is gebruikt zou moeten worden. In situaties waar het TA-Prioritized algoritme niet gebruikt kan worden presteert het TP algoritme beter in kleinere drukke pakhuis omgevingen terwijl het TPTS algoritme beter presteert dan het TP algoritme in grotere pakhuis omgevingen. We concluderen ook dat het toevoegen van rotaties aan de robots in de algoritmes leidt tot een significant slechtere prestaties. In het geval van het TA-Prioritized algoritme is deze impact op de performantie van het algoritme zo groot dat het algoritme niet meer bruikbaar is vanaf 20 robots. Hiervoor introduceren we in deze paper een aangepaste versie van dit algoritme dat maximaal 50 agents ondersteund.

Contents

Abstract	iii
Samenvatting	v
1 Introduction	1
1.1 Problem Definition	1
1.1.1 Research Questions	2
1.2 Thesis Structure	2
2 Background	3
2.1 Multi-agent pickup & delivery problem	3
2.2 Approaches to the MAPD problem	6
2.2.1 Token Passing	6
2.2.2 Token Passing with Task Swaps	8
2.2.3 Task Allocation and Prioritized Pathplanning	11
2.2.4 Selection of algorithms	14
3 Implementation	17
3.1 Unity environment	17
3.2 Contributions	19
3.2.1 Token Passing	19
3.2.2 Token Passing with Task Swapping	20
3.2.3 Task Allocation and Prioritized Pathplanning	21
4 Results	23
4.1 Metrics	23
4.2 Evaluation environments	24
4.3 Experimental Results	24
4.3.1 Scalability in terms of number of agents	25
4.3.2 Scalability in terms of warehouse size	30
4.3.3 Performance impact of rotations	33
4.3.4 Performance of parallelized Prioritized Path-planning	34
5 Conclusion	35

Chapter 1

Introduction

In recent years artificial intelligence (AI) has played a crucial role in the development of today's society going from innovation in the automobile industry (Shalev-Shwartz et al., 2016) (Dresner & Stone, 2008), with the recent explosion of self-driving car technologies, to the increasing importance of Internet Of Things (IoT) devices in our everyday life (Manate et al., 2013) (Calvaresi et al., 2017). Artificial intelligence took a huge leap forward in recent years as significant innovations have seen the light in domains such as machine learning, computer vision, etc.

One of the fields of artificial intelligence that has received a lot of attention in recent years is the field of multi-agent systems. Multi-agent systems are composed of a number of agents that interact with each other. Agents are autonomous entities that act on the behalf of others by executing specific actions. These actions can be provoked by a change in the environment that the agent perceives (reactivity) or by an initiative that the agent takes (pro-activeness) (Rabuzin et al., 2006). Agents can also have numerous other characteristics such as the ability to learn, the ability to communicate, and more importantly for multi-agent systems the ability to cooperate (Wooldridge & Jennings, 1994). These agents range from robotic agents, such as robots in automated warehouses (Ma et al., 2017), to software agents such as chatbots (AbuShawar & Atwell, 2015).

1.1 Problem Definition

This paper focuses on one application of multi-agent systems: automated warehouses. These are warehouses where the process of collecting orders in the warehouse and delivering them to a specific location is automated. The agents, in this case, are robots that navigate through the warehouse, each picking up and delivering a specific item.

The problem of picking up and delivering tasks is a more general problem known as the Multi-Agent Pickup and Delivery problem (MAPD) and has been well-researched in recent years. This paper analyzes this problem and three different approaches: the Token Passing (TP) algorithm, the Token Passing with Task Swapping algorithm and the Task Allocation and Prioritized Path-planning (TA-Prioritized) algorithm. that have been developed to tackle this problem. The three approaches are implemented and visualized using the Unity game engine and the ML-Agents toolkit, which allows us to incorporate intelligent agents in 3D environments. Using these implementations the three approaches are evaluated and compared in the context of automated warehouses.

1.1.1 Research Questions

In this paper we try to answer the following research questions:

- What is the multi-agent pickup and delivery problem (MAPD)?
- How do the TP, TPTS and TA-Prioritized algorithms tackle the MAPD problem?
- How can the MAPD problem and the TP, TPTS and TA-Prioritized algorithms be visualized?
- How well do the TP, TPTS and TA-Prioritized algorithms scale in terms of:
 - The number of agents
 - The size of the warehouse
- Which impact does the inclusion of agent rotations have on the performance of the algorithms?

1.2 Thesis Structure

This paper is structured as follows. We start by exploring the Multi-Agent Pickup and Delivery problem and the selected approaches to the problem in Chapter 2. We then discuss the implementation and working of the approaches and the visualization tool in Chapter 3. In Chapter 4 we compare the different approaches in various warehouse environments. Finally, we present a summary of our findings and the next steps to be taken in Chapter 5.

Chapter 2

Background

2.1 Multi-agent pickup & delivery problem

The Multi-Agent Pickup and Delivery problem (MAPD) is the problem of assigning a set of tasks to a group of agents and planning collision-free paths for these agents to execute these tasks. As the problem consists of a set of tasks and a set of agents the MAPD problem consists of a task-allocation component and a path-planning component. This is why most approaches to the MAPD problem (and the three approaches studied in this paper) solve the problem in two phases: a task-allocation phase and a path-planning phase.

In the task-allocation phase, tasks are assigned to the different agents so that each agent has at least one task that it can execute if there are tasks available. In this phase, the goal is to find the “best” task or task sequence for each agent.

In the path-planning phase, we calculate collision-free paths for each agent so that it can execute the task(s) it has been assigned in the task-allocation phase. The path-planning component of the MAPD problem has been well-studied in the context of the Multi-Agent Pathfinding problem (Silver, 2005) which can be seen as a one-shot version of the MAPD problem (Ma et al., 2017). The Multi-Agent Pathfinding problem is the problem of planning collision-free paths for agents from their current location to a goal location. There exist multiple optimal solutions for the MAPF problem that can be used to solve the path-planning component of the MAPD problem: Conflict-Based Search, Space-Time A*, Enhanced Partial Expansion A*, M*, etc.

Now that we have a general understanding of the problem, we take a look at how tasks are defined in the MAPD problem.

2.1.1 Tasks

A task in the Multi-Agent Pickup and Delivery problem always consists of two locations: a pickup location and a delivery location. Any agent that is currently not executing any tasks is a free agent and can be assigned to any of the available tasks. When an agent gets assigned a task it moves from its current location to the pickup location of that task. Once it reaches that location it picks up the task and starts executing it. The agent is now occupied and the task that the agent picked up is removed from the task set so that no other agent can try to execute it as well. The agent then moves from the pickup location of the task to the delivery location of the task where it delivers the task. Once the agent delivers the task, the task is considered done and the agent becomes a free agent again. This means that a new task can once again be assigned to the agent.

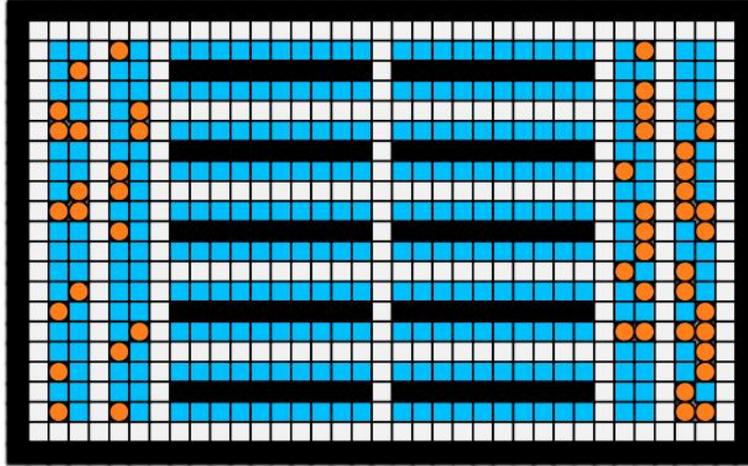


Figure 2.1: Example of a warehouse environment with task endpoints in blue and non-task endpoints in orange (Ma et al., 2019)

We mentioned that tasks are always composed of a pickup and a delivery location. In the MAPD problem these locations are special locations. In the next section we take a look at the different type of locations that can be used in the environment of an instance of the MAPD problem.

2.1.2 Environment

In the MAPD problem, the environment is represented as a 2D grid. Each location corresponds to an x- and y-coordinate. In the MAPD problem we consider 4 types of locations:

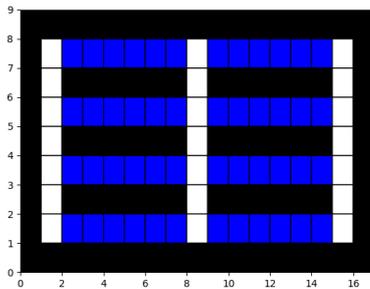
- Task Endpoints: All locations that can be either pickup or delivery locations
- Non-task Endpoints: Initial locations of agents and extra parking locations for agents
- Path locations: Locations that can be used by the agents to move in the environment
- Wall locations: Obstacle locations that the agents can not use to move in the environment

This means that the pickup and delivery locations of tasks will always be task-endpoints. Figure 2.1 is an example of a warehouse environment of an instance of the MAPD problem. In this figure the task-endpoints are colored blue, the non-task endpoints orange, the walls black and the path locations white. Agents can use all the locations except the wall locations to move through the warehouse environment.

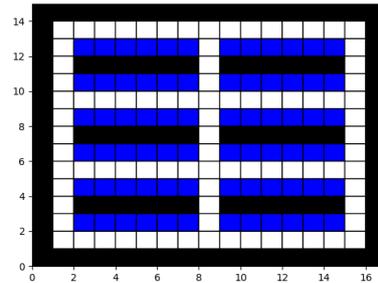
2.1.3 Well-formed MAPD instances

Not every instance of the MAPD problem is solvable by the algorithms that are analyzed in this paper. All of the algorithms in this paper are complete for well-formed instances of the MAPD problem. This means that the algorithm will always be able to find a solution for a well-formed instance of the MAPD problem if it receives enough resources to be able to compute a result. An instance of the MAPD problem is well-formed if it satisfies the following conditions:

- The number of tasks in the task set is finite
- There is at least one non-task endpoint for each agent
- For any two (task or non-task) endpoints there exists a path that traverses no other endpoint



(a) Typical not well-formed warehouse environment



(b) Well-formed warehouse environment

Figure 2.2: Typical not well-formed warehouse environment and well-formed warehouse environment

In the context of automated warehouses, it is fairly easy to create environments that satisfy these conditions. Typically warehouses consist of small and long corridors as can be seen in the left image on figure 2.2. The locations in between the corridors are then generally pickup and delivery locations. Such a warehouse could not be used as a warehouse environment for the MAPD problem as the third condition is not satisfied. For the pickup and delivery locations of tasks to be the locations in the corridors these locations would have to be task-endpoints. If these locations are task endpoints we can clearly see that there is no path that traverses no other endpoints between any of these task endpoints. The other warehouse environment in figure 2.2 represents an environment with larger corridors. This environment could be used in a well-formed instance of the MAPD problem as there exists a path that only consists of path locations between every two endpoints.

Now that we understand all the components of the MAPD problem we look at the difference between the offline and the online variant of the MAPD problem as we analyze approaches for both variants of the MAPD problem in this paper.

2.1.4 Online vs offline MAPD problem

The MAPD problem exists in two variants: the online and offline variant of the problem. The difference between both variants is the possibility to add new tasks during the execution of the algorithm. In the offline variant of the MAPD problem, we assume that all tasks are known at the start of the problem. This means that no other tasks can enter the system once the task-allocation phase of the algorithm has started. As not all tasks are always available to be picked up at the start of the algorithm the offline variant of the problem introduces release times for tasks. The release time of a task is the first timestep at which the task can be picked up. An agent can not pick up a task if the current timestep is smaller than the release time of that task.

In the online variant of the problem, we assume that new tasks can enter the system at any point. Adding a release time for tasks in the online variant of the problem is therefore not necessary as we can simply add a task only when it is available to be picked up. We assume that a task can be picked up from the moment it enters the system.

Both variants of the problem can be encountered in the context of automated warehouses. In an automated warehouse where a product is for example picked up as soon as an order arrives it would be better to use an algorithm for the online problem while in a warehouse where we know beforehand which tasks have to be executed on a specific day it would be better to use an algorithm for the offline variant of the MAPD problem.

Now that we have a better understanding of the MAPD problem we can look at approaches to the MAPD problem in the next chapter.

2.2 Approaches to the MAPD problem

Since the introduction in the literature of the MAPD problem, there have been different approaches to the problem. These approaches differ in their way of allocating tasks and planning paths. Each approach has some advantages and drawbacks. This chapter describes the three selected approaches as well as why they were chosen.

2.2.1 Token Passing

The first implemented algorithm is the Token Passing (TP) algorithm (Ma et al., 2017). The TP-algorithm is a decentralized algorithm for the online variant of the MAPD problem. This means that each agent chooses its own tasks from the task set and plans its own paths. In order to be able to choose tasks and plan collision-free paths, each agent needs some information about the global state of the environment: available tasks, paths of other agents, etc. In the TP-algorithm this information is exchanged between the agents with a token. A token is a shared block of memory that all agents can access. In the TP-algorithm the token contains the set of available tasks, the paths of all the agents, and the current tasks assignments of all the agents. The token is stored by a system that sends the token to any agent that requests it. Whenever an agent reaches the end of its planned path it requests the token and tries to find a task and plan a collision-free path to execute that task.

The TP-algorithm is based on a similar concept as the Cooperative A* algorithm (Silver, 2005). Cooperative pathfinding algorithms solve the Multi-Agent Path-finding problem by breaking down the problem into a series of single-agent pathfinding problems. These problems can then be solved using state-of-the-art algorithms such as the A* algorithm. The same concept is used by the TP-algorithm as the agents plan paths one after the other using a space-time A* algorithm. The space-time A* algorithm is a modified version of the original A* algorithm. In the space-time A*-algorithm a third dimension is added to the algorithm: the time dimension. This allows us to represent the agent's positions in the future in order to avoid collisions between the different agents.

Working of the algorithm

Algorithm 2.1 describes the working of the TP algorithm. The algorithm starts by initializing the token. As mentioned earlier the token contains all the unassigned/available tasks, the paths of all the agents, and the current task assignments of all the agents. To initialize the token the path of each agent a_i is set to the trivial path to its current location: $[\text{loc}(a_i)]$.

Algorithm 2.1: Token Passing algorithm (Ma et al., 2017)

```

1 Initialize token with the (trivial) path  $[loc(a_i)]$  for each agent  $a_i$ 
2 while true do
3   Add all new tasks, if any, to the task set  $\mathcal{T}$ 
4   while agent  $a_i$  exists that requests token do
5     /* system sends token to  $a_i$  -  $a_i$  executes now */
6      $\mathcal{T}' \leftarrow \{\tau_j \in \mathcal{T} \mid \text{no other path in } token \text{ ends in } s_j \text{ or } g_j\}$ 
7     if  $\mathcal{T}' \neq \emptyset$  then
8        $\tau \leftarrow \arg \min_{\tau_j \in \mathcal{T}'} h(loc(a_i), s_j)$ 
9       Assign  $a_i$  to  $\tau$ 
10      Remove  $\tau$  from  $\mathcal{T}$ 
11      Update  $a_i$ 's path in token with  $Path1(a_i, \tau, token)$ 
12    else if no task  $\tau_j \in \mathcal{T}$  exists with  $g_j = loc(a_i)$  then
13      Update  $a_i$ 's path in token with the path  $[loc(a_i)]$ 
14    else
15      Update  $a_i$ 's path in token with  $Path2(a_i, token)$ 
16    /*  $a_i$  returns token to system - system executes now */
17  All agents move along their paths in token for one timestep
18  /* system advances to the next timestep */

```

The TP-algorithm then enters an infinite loop in which it continuously first checks if any new tasks need to be added to the task set. If there are new tasks available the algorithm adds them to the task set of the token. The system will then send the token to any agent that requests it. In the TP-algorithm each free agent, an agent that has currently no tasks assigned to it and is waiting at an endpoint, requests the token in each timestep.

When a free agent receives the token it starts looking for a task [Line 5-16]. Before the agent starts to look for the best task it generates a list of available tasks as not all tasks in the task set are available for each agent [Line 6]. A task is available if the path of no other agent ends in the pickup or the delivery locations of the task. This check is needed as the TP-algorithm assumes that free agents can rest (eventually forever) in the last location of their path which means that the agent that is looking for a task would not necessarily be able to plan a path to the pickup or delivery location of an unavailable task.

If the agent finds at least one available task it selects the best available task using a heuristic [Line 8]. In the implementation of the algorithm of this paper, the agent uses the Manhattan distance heuristic to find the task of which the pickup location is the closest to the agent's location. The agent assigns that task to itself in the token [Line 9] and removes the task from the task set [Line 10]. It then plans a collision-free path from its current location to the pickup location of that task and a collision-free path from the pickup location of the task to the delivery location of the task [Line 11].

If there are no available tasks the agent has to move to a location where it cannot prohibit other agents from executing their tasks. If the agent's current location is not the delivery location of any task in the task set the agent can stay at its current location as it is not blocking any other agent from executing a task. It then updates its path in the token with the trivial path to its current location [Line 13].

If the current location of the agent is the delivery location of any task in the task set the agent has to move to a different endpoint. The agent then looks for the closest (task or non-task) endpoint that is not a pickup or delivery location of any task in the task set and that is not in

the path of any agent. It then plans a collision-free path to that location [Line 15]. The agent can start looking for a new task once it reaches that endpoint.

Once the agent has planned a new path, either to the pickup location of its new task or to an endpoint where it can rest it returns the updated token to the system. The system can then pass the updated token to the next agent that requests the token. Once all the agents that have requested the token have received it and planned their paths all agents move along their path in the token for one timestep [Line 17] and the system increments the current timestep [Line 18].

The TP-algorithm is a basic algorithm as can be seen in the code of algorithm 2.1. This is why we look at an improved version of this algorithm, The Token Passing with Task Swaps (TPTS) algorithm (Ma et al., 2017), in the next section.

2.2.2 Token Passing with Task Swaps

The next algorithm is the Token Passing with Task Swaps (TPTS) algorithm (Ma et al., 2017). The TPTS-algorithm is an improvement of the TP-algorithm. It improves the task-allocation component of the TP-algorithm by allowing free agents to swap tasks with each other if they can reach the pickup location of a task more efficiently Algorithm 2.2 describes the working of the TPTS-algorithm.

Working of the algorithm

Algorithm 2.2: Token Passing with Task Swapping (TPTS) (Ma et al., 2017)

```

1 Initialize token with the (trivial) path  $[loc(a_i)]$  for each agent  $a_i$ 
2 while true do
3   Add all new tasks, if any, to the task set  $\mathcal{T}$ 
4   while agent  $a_i$  exists that requests token do
5     /* system sends token to  $a_i$  -  $a_i$  executes now */
6     GetTask( $a_i$ , token)
7     /*  $a_i$  returns token to system - system executes now */
8   All agents move along their paths in token for one timestep and remove tasks from T
   when they start to execute them
9   /* system advances to the next timestep */
```

Similar to the TP-algorithm the token is first initialized by updating the path of each agent in the token with the trivial path to its current location: $[loc(a_i)]$ [Line 1]. The algorithm then also enters an infinite loop in which it first adds eventual new tasks to the task set [Line 3]. The system then sends the token to each free agent that requests it [Line 5]. When an agent receives the token it looks for a task and plans a new path using the GetTask function [Line 6]. The agent then returns the updated token to the system that sends it to the other agents that are still waiting for the token. Once all the agents that requested the token have planned their tasks and paths all agents move following their path in the token by one timestep [Line 8] and the system increments the timestep [Line 9]. We can see that currently there are no differences between the TP and the TPTS algorithm. The difference between both algorithms can be observed in the code of the GetTask method. Algorithm 2.3 describes the working of this method.

The GetTask function looks for tasks in a similar way as the TP-algorithm. It first creates a set of available tasks [Line 2]. A task is available for an agent if no other path in the *token* ends in the pickup or the delivery location of the task. As long as there are still available tasks the

Algorithm 2.3: GetTask function of TPTS-algorithm (Ma et al., 2017)

Input: a_i Agent for which task is planned, $token$: Token provided by system
Output: Boolean: Whether function was able to find a task

```

1 Function GetTask( $a_i, token$ ):
2    $\mathcal{T}' \leftarrow \tau_j \in \mathcal{T} \mid$  no other path in  $token$  ends in  $s_j$  or  $g_j$ 
3   while  $\mathcal{T}' \neq \emptyset$  do
4      $\tau \leftarrow \arg \min_{\tau_j \in \mathcal{T}'} h(\text{loc}(a_i), s_j)$ 
5     Remove  $\tau$  from  $\mathcal{T}'$ 
6     if no agent is assigned to  $\tau$  then
7       Assign  $a_i$  to  $\tau$ 
8       Update  $a_i$ 's path in  $token$  with Path1( $a_i, \tau, token$ )
9       return true
10    else
11      Remember  $token$ , task set and agent assignments
12       $a'_i \leftarrow \text{agentthat is assigned to } \tau$ 
13      Unassign  $a'_i$  from  $\tau$  and assign  $a_i$  to  $\tau$ 
14      Remove  $a'_i$ 's path from  $token$ 
15      Path1( $a_i, \tau, token$ );
16      Compare when  $a_i$  reaches  $s_j$  on its path in  $token$  to when  $a'_i$  reaches  $s_j$  on its
        path in  $token$ '
17      if  $a_i$  reaches  $s_j$  earlier than  $a'_i$  then
18        /*  $a_i$  sends  $token$  to  $a'_i$  -  $a'_i$  executes now */
19        success  $\leftarrow$  GetTask( $a'_i, token$ )
20        /*  $a'_i$  returns  $token$  to  $a_i$  -  $a_i$  executes now */
21        if success then
22          return true
23      Restore  $token$ , task set, and agent assignments
24    if  $\text{loc}(a_i)$  is not an endpoint then
25      Update  $a_i$ 's path in  $token$  with Path2( $a_i, token$ )
26      if path was found then
27        return true
28    else
29      if no task  $\tau_j \in \mathcal{T}$  exists with  $g_j = \text{loc}(a_i)$  then
30        Update  $a_i$ 's path in  $token$  with the path [ $\text{loc}(a_i)$ ]
31        else
32          Update  $a_i$ 's path in  $token$  with Path2( $a_i, token$ )
33        return true;
34    return false

```

“best“ task out of the available task is selected. Selecting the “best“ task is also here done using a distance heuristic (in this case the Manhattan distance) [Line 4]. That task is then removed from the set of available tasks [Line 5]. This is the first difference between the TP and the TPTS algorithms. When the TP-algorithms selects an available task it is sure that it is able to plan a path for that task as no other agent is assigned to that task. When the TPTS algorithm selects a task it first checks whether it can reach the pickup location of that task more efficiently than the agent that is already assigned to that task if there is one. If this is not the case the agent tries the next “best“ available task. This process is repeated as long as there are still available tasks.

When an agent selects a task there are two possible scenarios: no other agent is assigned to the selected task or another agent is already assigned to the selected task but has not yet reached the pickup location of the selected task. If no other agent had already assigned that task to itself the agent can assign that task to itself [Line 7], plans a path to execute that task [Line 8], and returns true to indicate that it was able to find a task.

If another agent had already assigned that task to itself a copy of the token is created [Line 11]. This copy of the token is used to restore the token to its original state in case the agent can not reach the pickup location of the task more efficiently than the agent that is already assigned to the task. The agent removes the assignment of the other agent [Line 13] and removes the other agent’s path from the current token [Line 14]. The agent then plans a path to execute the task [Line 15]. It then compares if it can reach the pickup location of that task more efficiently than the agent that was already assigned the task [Line 17].

If that is the case the agent calls the GetTask function for the other agent [Line 19]. The other agent will then try to find a new task for itself and plan a path to the pickup location of that task or try to plan a path a safe location where it can rest. This is the only situation in which the GetTask function can return false. When the GetTask function is called for the other agent that agent can already be on his way to the pickup location of the task. This means that the other agent is not necessarily on a task endpoint. In a well-formed MAPD instance, there exists a path between every two endpoints that traverses no other endpoint but there is not necessarily a path between two non-endpoint locations that traverses no other endpoint. This means that the other agent could not find a path to a pickup location of a new task or a path to a safe location where it can rest. If that is the case the agent returns false and the token is restored using the copy that was made earlier and the agent for which we were looking for a task tries the next “best“ available task.

If there are no more tasks available the agent tries to plan a path to a safe location where it can rest. As was described earlier the agent can be at a non-endpoint location when the GetTask function is called from line 19. If this is the case the agent tries to plan a path from its current location to the closest free endpoint [Line 25]. If the agent is able to plan such a path it returns true [Line 27]. This allows the agent that called the GetTask function for this agent to swap tasks with this agent. Otherwise, the GetTask function returns false [Line 34] and the agents can not swap tasks. If the agent is currently on an endpoint it checks whether its current location is a free endpoint. An endpoint is free if it is not the delivery location of any task in the task set. If its current location is free it updates its path in the token with the trivial path to its current location [Line 30]. Otherwise, the agent plans a path to the closest free-endpoint [Line 31]. In both cases, the GetTask function returns true as it was able to plan a path to the pickup location of another task or to a safe location. This allows the agents to swap tasks.

Both algorithms that we have described in the previous sections are decentralized algorithms for the online variant of the MAPD problem. The last algorithm in this paper is a centralized algorithm for the offline variant of the MAPD problem: the Task Allocation and Prioritized Pathplanning algorithm. In the next section we take a look at the working of this algorithm.

2.2.3 Task Allocation and Prioritized Pathplanning

The last algorithm is the Task Allocation and Prioritized Pathplanning algorithm (TA-Prioritized) (Ma et al., 2019). In contrast to the TP- and TPTS-algorithms the TA-Prioritized algorithm is a centralized algorithm that tackles the offline variant of the MAPD problem. This means that the algorithm assumes that all tasks are known beforehand. This also means that tasks now also have a release time on top of the pickup and delivery locations that they already had in the online variant of the problem. The algorithms for the online variant of the MAPD problem can also be used to solve the offline variant of the problem but they produce less optimal results as they do not use all the available information. The TA-Prioritized algorithm consists of 2 distinct phases: a task-allocation phase and a path-planning phase. In the task-allocation phase, the algorithm creates task sequences for each agent. Each agent gets assigned an ordered list of tasks that he needs to execute in that order. In the path-planning phase, the algorithm plans one collision-free path for each agent that executes all the tasks of that agent. We start by analyzing the task-allocation component of the algorithm.

Task Allocation

In the task-allocation phase of TA-Prioritized, the algorithm first creates a directed weighted graph for the MAPD instance similarly to (Osaba et al., 2015) and (Yoon & Kim, 2017). The algorithm then solves a special TSP to compute the best task sequences for all the agents. The TA-Prioritized creates a graph $G=(V, E)$. The set of nodes \mathcal{V} of the graph consists of all the agents and tasks (t_i) ($V = A \cup T$). The set of nodes therefore consists of two types of vertices: agent vertices ($a_i \in A$) and task vertices ($t_i \in T$). Next to the vertices there are also 4 types of edges with different weights:

- An edge from agent a_i to task t_j : $\max(\text{dist}(a_i.\text{position}, t_j.\text{pickup}), t_j.\text{release} - \text{time})$
- An edge from task t_i to task t_j : $\text{dist}(t_i.\text{pickup}, t_i.\text{delivery}) + \text{dist}(t_i.\text{delivery}, t_j.\text{pickup})$
- An edge from task t_i to agent a_j : $\text{dist}(\text{pickup}_i, \text{delivery}_i)$
- An edge from agent a_i to agent a_j : 0

The first type of edges is edges from an agent vertex to a task vertex. These vertices represent an agent moving from its current location to the pickup location of its first task. The weight of the edge is the maximum of the time the agent needs to move to the pickup location and the release time of the task. The second type of edges is edges from a task vertex to another task vertex. These vertices represent an agent that is executing his current task and moving to its next task. The weight of the edge is the cost of executing its current task: $\text{dist}(t_i.\text{pickup}, t_i.\text{delivery})$ as well as the distance to move from the delivery location of its current task to the pickup location of its next task: $\text{dist}(t_i.\text{delivery}, t_j.\text{pickup})$. The third type of edges is edges from task vertices to agent vertices. These vertices represent agents executing their last tasks. The weight of this edge is the distance from the pickup location of that task to the delivery location of that task. The last type of edges is edges from an agent vertex to another agent vertex. These edges have no weight as agent a_i has no tasks assigned as it has already delivered its last task.

For every non-trivial instance of the MAPD problem with at least one agent and at least one task, we know that G contains at least one Hamiltonian cycle as G is a complete graph with more than 2 vertices. A Hamiltonian cycle in a graph is a cycle that visits each vertex exactly once. The Hamiltonian cycle, therefore, visits each agent vertex exactly once and can be partitioned into M parts (considering there are M agents) where each part starts with an agent vertex and

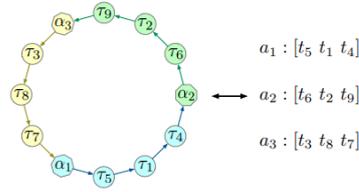


Figure 2.3: Example of a Hamiltonian cycle that can be partitioned into 3 parts that represent the task sequences of the 3 agents (Ma et al., 2019)

is followed by a sequence of task vertices. Figure 2.3 is an example of such a Hamiltonian cycle for 3 agents. These M parts can then be converted into M task sequences for the M agents. Due to the definition of the weight of the edges in the graph the sum of the weight of all the edges in the Hamiltonian path is a lower bound for the makespan of the algorithm. The makespan of the algorithm is the earliest timestep at which all tasks have been executed and all agents have stopped moving. This is an important metric when looking at approaches to the MAPD problem and will later be used to evaluate and compare the different approaches. Looking for a good Hamiltonian cycle in the graph, therefore, corresponds to looking for a Hamiltonian cycle with a low sum of the weights of the edges of the path.

The TA-Prioritized algorithm finds a good Hamiltonian cycle using the LKH-3 TSP solver (Helsgaun, 2017). The TSP solver can solve various TSP problems as well as various types of Vehicle Routing Problems. The graph is provided to the TSP solver as a Traveling Salesman Problem with Time Windows (TPSTW). The LKH-3 solver then looks for Hamiltonian cycles that can be partitioned correctly. In each iteration, the LKH-3 TSP solver provides a Hamiltonian Cycle with a specific cost that is lower than the one computed in the previous iteration. This is done as this is a very slow process. We therefore always provide a time limit to the TSP solver and use the best solution found in that time span. It is important to give the TSP solver enough time as the cost that the LKH-3 solver assigns to a specific Hamiltonian Path is only the lower bound of the makespan of the task sequences in that path. The higher the estimated makespan of the TSP solver the higher the actual makespan will be once the paths of the agents have been planned. Once we have a good Hamiltonian Path from the TSP solver we the path is converted to task sequences by assigning all the tasks of which the task vertices follow an agent vertex to the agent that corresponds to that agent vertex.

Once all the tasks have been assigned the algorithm calculates paths for all the agents. This is done using the Prioritized Path planning algorithm as described in the following section.

Prioritized Path Planning

The Prioritized Pathplanning algorithm of TA-prioritized is an improved version of the Prioritized Motion Planning algorithm of (Van Den Berg & Overmars, 2005). This algorithm first assigns priorities to each agent. This priority is used to determine the order in which paths are going to be planned. This is done to avoid increasing the execution time of agents with large task sets. Prioritized Motion Planning uses an estimation of the execution time of each agent on their task set as a priority for that agent. It then plans the paths of these agents in decreasing order of priority. This means that agents with larger estimated execution times are handled first. This is important as once a path has been planned, all the paths of the remaining agents cannot collide with the planned path. This means that some of these agents for which paths are planned later might have to wait in their path which will increase their execution time. This is why it

is important to start with the agents that already have a large estimated execution to avoid increasing the makespan of the algorithm.

The Prioritized Pathplanning algorithm of TA-Prioritized algorithm improves the Prioritized Motion Planning algorithm by using the actual execution times of the agents on their tasks instead of estimates. As the Prioritized Motion Planning algorithm uses the estimated execution times as a priority for each agent at each step the order in which all the paths will be planned is already decided before the algorithm even starts planning paths. The Prioritized Pathplanning algorithm of TA-prioritized only chooses the agent it is going to plan for next after it is done with the agent it is currently planning for. This allows the algorithm to use the actual execution times instead of the estimated execution times as priorities for the agents. The actual execution times can be calculated as we can plan paths for all the remaining agents that do not collide with the already planned paths. We can then select the agent with the largest execution time as the next agent, remove that agent from the remaining agents, and add its planned path to the paths that have already been planned. We then plan new paths for all the remaining agents that do not collide with the updated list of already planned paths and repeat this process until all the agents have a collision-free path.

Planning a path in the Prioritized Pathplanning algorithm of TA-Prioritized works by dividing the total path for each agent into several sub-paths. The path of an agent to complete one task consists of a sub-path from the agent’s current location to the pickup location of the task, a sub-path from the pickup location to the delivery location of the task, and a sub-path from the delivery location of the task to the pickup location of the agent’s next task or a location where the agent can rest. The TA-Prioritized plans these collision-free sub-paths one after the other. The algorithm starts by planning a path for the agent to execute the first task in its task sequence, then the second task of its task sequence, etc. The last sub-path is a path from the agent’s last delivery location to the agent’s initial location which is considered the agent’s parking location. Planning collision-free sub-paths is done using the same space-time A* algorithm as the space-time A* algorithm of the TP and TPTS algorithms.

In order to avoid situations where a sub-path cannot be planned due to multiple agents blocking each other the TA-Prioritized algorithm uses a deadlock-avoidance method called “reservation of dummy paths” (Liu et al., 2019). This deadlock-avoidance method is explained in the following section.

Deadlock-avoidance: dummy path reservations

Avoiding deadlock situations in the TA-Prioritized is achieved by reserving dummy paths. Dummy paths are collision-free paths from the goal location of a sub-path to the agent’s parking spot. This path allows the agent to move back to its parking location and wait there until it can find a collision-free path to the goal location of that sub-path.

To introduce dummy paths the TA-prioritized algorithm modifies the goal function of the A* algorithm that it uses. The A* algorithm normally considers the problem to be solved when the goal node has been expanded. This ensures that the goal node can not be reached in a more efficient way through another path. The TA-prioritized algorithm uses a different goal function for the A* algorithm. The algorithm only considers the problem to be solved when the goal node has been expanded and there exists a collision-free dummy path from the goal location to the agent’s parking spot. Such a dummy path can always be calculated. This is proved by induction in (Liu et al., 2019). The idea behind the proof is that an agent can always wait at its parking spot until all the other agents have finished their tasks and returned to their parking spots before it starts executing its own tasks. At that point, no other agent can block the agent from executing its tasks. By adding these dummy paths at the end of each sub-path the TA-

prioritized algorithm ensures that an agent can always move back to its parking location to wait until it can plan a collision-free sub-path to the goal location of the sub-path.

2.2.4 Selection of algorithms

Now that we understand how the selected algorithms work we take a look at the characteristics and differences of the selected algorithms and why these algorithms were selected to represent the MAPD problem in the context of automated warehouses.

Offline/online setting

The first difference between the selected algorithms is the difference between the algorithms for the online and the algorithms for the offline versions of the MAPD problem. As we want the tool to represent the MAPD problem in its most general form we have selected algorithms for both variants of the problem. The Token Passing (TP) algorithm and the Token Passing with Task Swaps (TPTS) algorithm are algorithms for the online variant of the MAPD problem while the Task Allocation and Prioritized Pathplanning (TA-Prioritized) algorithm is an algorithm for the offline variant of the MAPD problem. As described in section 2.1.4 the main difference between both versions of the problem is that all the tasks have to be known at the start of the algorithm in the offline variant of the problem while in the online variant of the problem, it is assumed that tasks can enter the system at any point.

Representing both variants of the MAPD problem in the tool is important as both variants can be found in real-world warehouses. An example of an automated warehouse environment for which all tasks are known beforehand is for example a warehouse where customers can buy products and pick these products up at a specific day and time. This is similar to the system that was used by various stores during the COVID-19 crisis. In these warehouses, all tasks of a specific day and the time at which they need to be picked up are known beforehand. In this warehouse environment, offline algorithms such as the TA-Prioritized algorithm can be used as they will be more efficient than the algorithms for the online MAPD problem. In the previous example, all the packages that need to be picked up on a specific day were known beforehand. This information is however not always available. A warehouse where products need to be picked up as soon as someone orders a product for example is a warehouse environment where only algorithms for the online MAPD problem can be used.

Centralized/decentralized approaches

Another important characteristic of the different approaches is the distinction between the centralized and decentralized approaches to the problem. The difference between both approaches is where the tasks and paths are planned. Centralized approaches use a central system to allocate all the tasks and plan paths for all the agents while decentralized approaches let each agent plan its own tasks and paths. Both approaches have advantages and disadvantages that will be studied in this section.

One of the major advantages of decentralized approaches is the fact that these systems are easier to scale than centralized systems as they can be easily extended to a fully distributed algorithm (Claes et al., 2017). The analyzed decentralized approaches that tackle the MAPD problem (TP, TPTS, and TP-SIPPwRT) for example let all agents assign tasks to themselves and plan their own paths based on some global information that is provided to them via a token. This means that new agents can easily be added to the system as these will calculate their own tasks and paths. In approaches where a central system is used for the planning of tasks and

paths, this is more difficult as the new agents will increase the number of computations that have to be done by the single central system.

Another advantage of decentralized approaches is the robustness of these approaches compared to centralized approaches. As all the agents plan their own tasks and paths in decentralized approaches there is no central system that can fail. This means that only individual agents can fail. If this is the case the agent can be removed while the other agents can continue with their tasks. This is not the case in a centralized approach. If the central system fails in a centralized approach no task or path can be planned for any agent.

Decentralized approaches also have disadvantages. One of these disadvantages is the cost that is associated with these approaches. Decentralized approaches require agents that are able to compute their own tasks and paths. This requires robots with more advanced hardware for example in the setting of automated warehouses which is not always available. Robots in centralized approaches need less advanced hardware as they just need to be able to follow a given plan.

Another disadvantage of decoupled algorithms is the amount of communication needed between the agents. In a centralized algorithm such as TA-Prioritized and TA-Hybrid, only the central entity communicates with the agents. It assigns each agent one or more tasks and gives it one or more paths to execute these tasks. Once the agents have received this information no further communication is needed until the system gets new tasks that it needs to assign. The agents do not have to communicate between themselves as the central entity guarantees that there will be no collisions between the agents. In decoupled algorithms such as the TP, TPTS, and the TS-SIPPwRT algorithms this is not the case. As the agents plan their own tasks and paths they need to receive global information about the available tasks and the paths of the other agents to plan their own tasks and plans. In the TP, TPTS, and TS-SIPPwRT algorithms this information is exchanged in the form of tokens. Consistently sending this token back and forth between the agents can be slow in large environments.

Now that we understand the selected approaches and why they were selected we look at the implementation of the visualization tool and the selected approaches.

Chapter 3

Implementation

In this chapter, we give an overview of the implementation and working of the visualization tool and the selected approaches.

3.1 Unity environment

The visualization tool is created using the Unity game engine and its ML-agents platform. The Unity engine is a popular cross-platform game engine that allows the development of 2D and 3D-applications as well as interactive simulations. We use the Unity game engine to create a visual representation of the MAPD problem and the selected approaches. The selected approaches are implemented in Python and are connected to the Unity game engine via Unity’s ML-agents platform (Juliani et al., 2018). The ML-agents toolkit is an open source project which allows developers to create and manage environments that are simulated using the Unity game engine. The ML-agents platform allows us to interact with the simulation using its Python API. Using the ML-agents platform to implement these planning approach allows the tool to later support learning approaches as well without having to change the current implementation of the tool.

The tool supports both the offline and the online variant of the MAPD problem. The following section describes the working of the visualization tool. We first take a look at how we can configure the visualization tool to support different warehouse environments.

3.1.1 Configuration

The visualization tool supports a variety of environments. An environment in the tool is represented by a configuration file. Creating a new environment, therefore, consists of creating a new configuration file for that environment and providing that configuration file to the visualization tool. Various elements of the warehouse environment can be configured. The following list represents all the elements that can be configured using a configuration file:

- The warehouse setup: The current warehouse layout is determined by a 2D matrix in which open locations are represented by 0, obstacles are represented by 1, non-task endpoints are represented by 2 and task endpoints are represented by 3.
- The number of degrees an agent can rotate in 1 timestep: needs to be a divider of 90° .
- The amount of time needed for an agent to execute one action (movement from one location to another, rotating, picking up / delivering a task): default 1 second.

- The number of agents and their starting positions in the warehouse.
- The initial tasks in the task set.

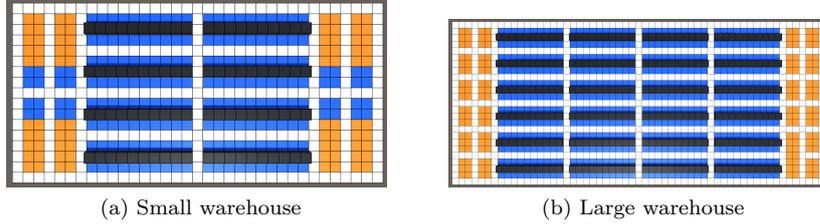


Figure 3.1: The 2 currently implemented warehouse environments

The visualization tool currently contains 2 warehouse environments that are represented in Figure 3.1.

3.1.2 Initialization

Once the Unity visualization loads a configuration file the environment is created in Unity. Based on the matrix provided in the configuration file the floor and the walls are generated. As can be observed in Figure 3.1 task endpoints are colored in blue, non-task endpoints in orange, paths in white and walls in black. Once the environment is generated all the information provided in the configuration file is sent to the implemented MAPD algorithm via the ML-agents platform. The MAPD algorithm is then initialized with all the provided data: agents and tasks are created, pathfinding planners are created with the provided environment, etc. The communication between the Unity engine and the MAPD algorithm is handled by 2 communication channels created with the ML-agents framework. A first channel is used to send all the required data for the initialization of the MAPD algorithm as described above while a second channel is used to communicate task-related information such as the addition of new tasks in the online variant of the MAPD problem.

3.1.3 Agent Actions

Once the algorithm is initialized the Unity engine starts requesting actions for the agents. An action consists of a position, a rotation, and two booleans that indicate whether the robot needs to pick a task up or deliver a task in this action. When an agent executes an action it can either move to a neighboring location or pick a task up or deliver it. This is where the tool makes a distinction between the offline and the online variants of the MAPD problem. In the offline variant of the tool, the Unity engine requests all the actions for all the agents at once. Currently, the only algorithm for the offline MAPD problem that is implemented in the tool is the TA-Prioritized algorithm. As the TA-Prioritized algorithm calculates the paths of all the agents at the start of the algorithm it only looks up the next action of each agent in the pre-computed paths and returns that action. In the online variant of the tool each Unity robot requests an action individually to the MAPD algorithm. For each Unity robot an Agent is created and managed in the MAPD algorithm. This agent computes the next action of the Unity robot and sends that action back. Both the TP and TPTS algorithms return the next position of the agent if the agent has already planned a path or the agent or start looking for a task and plan a new path for the agent if the agent has reached the end of its planned path.

Another point in which the environment for the offline version of the MAPD problem differs from the environment of the online MAPD problem is the task creation.

3.1.4 Task Creation

As explained in section 2.1.4 the difference between both variants of the problem is the availability of the tasks. The online variant of the problem allows new tasks to enter the system during the execution of the program while this is not allowed in the offline variant of the problem. To make this concept clear the environment for the online approaches of the MAPD problem always generates a new task as soon as a task is delivered. The environment of the offline approaches of the MAPD problem does not do this. Once all tasks are delivered in the offline environment the problem is considered solved and the simulation can be stopped, while in the online variant the problem is never actually solved as tasks are continuously added.

Now that we have an idea of how the Unity game engine and the ML-agents platform are used to create a visualization tool we can have a look at the contributions to this tool that are presented in this paper.

3.2 Contributions

This section describes the contributions to the tool that are presented in this paper. In this paper, we focused on the (online and offline) variants of the MAPD problem. We implemented 3 algorithms to tackle the MAPD problem and visualized them using the Unity tool.

3.2.1 Token Passing

The first implemented algorithm was the Token Passing (TP) (Ma et al., 2017) algorithm. The TP-algorithm was implemented from scratch based on the pseudo-code of section 2.2.1 that was provided by (Ma et al., 2017). The algorithm consists of a System that is responsible for the creation of the agents, the token and the tasks. When the information about the environment is provided by the Unity engine to the ML-agents platform the System is created. The system then creates and saves the provided tasks and creates the agents. For each Unity robot, an agent is created. This agent will plan the actions that are executed by the Unity robot. Each agent in the Token Passing algorithm has a space-time A* planner that it uses to plan paths to the pickup and delivery locations of tasks. The space-time A* algorithm that is used in the Token Passing algorithm (as well as in the Token Passing with Task Swaps algorithm) is a modified version of the space-time A* algorithm that is used by the Task Allocation and Prioritized Pathplanning algorithm. In order to be able to use this implementation of the space-time A* algorithm in the Token Passing algorithm some modifications were needed.

Modifications to space-time A*-algorithm

The first modification that was applied to the algorithm is the addition of rotations. Robots in the Unity environment have to rotate before they can start moving in different directions. In the original implementation of the algorithm, it was assumed that agents could move in any direction without having to rotate. Adding rotations was done by adding a fourth dimension to the states of the A* algorithm so that each state now consists of an x-coordinate, a y-coordinate, a timestep, and now a rotation as well. In order to make sure that agents could only move if they have the correct rotation the node expansion of the algorithm was also modified. Agents can now stay at their current location (and current rotation), rotate clockwise (by adding a constant

to their rotation), or rotate counterclockwise (by removing a constant degree from their current rotation). On top of that when agents have specific rotations, they can move in specific directions. Agents can move up when their current rotation is 0° , right when their current rotation is 90° , down when their current rotation is 180° and finally left when their current rotation is 270° . When the current rotation of an agent is for example 90° the agent has 4 possible moves: staying at its current location and not rotate, rotate counterclockwise, rotate clockwise, or move to the location to the right of the agent.

When adding rotations the heuristic that is used by the A* planner also needs to be adapted. The original implementation of the algorithm used the Manhattan distance as a heuristic. The Manhattan distance between two point P_1 and P_2 is calculated with the following formula: $dist(P_1, P_2) = abs(P_1.x - P_2.x) + abs(P_1.y - P_2.y)$. This formula does not take the rotation of the agents into account which causes agents to sometimes keep rotating instead of just waiting at a location. In order to prevent agents to rotate when they do not need to rotate we can add the absolute value of the difference between the rotations of the two locations. This adds a cost of 1 for each rotation the agent needs to execute.

Another modification that was applied to the original implementation of the algorithm is the reservation of parking locations of agents. The original implementation of the algorithm assumed that each agent had 1 parking spot that no other agent could ever use in a path. This assumption was made as the algorithm implementation of the space-time A*-algorithm moved each agent back to its parking location after the agent reached its final delivery location. This means that an agent always moves to its parking location to rest. This is not the case in the Token Passing algorithm. In the Token Passing algorithm, agents can rest at their last delivery location as long as it is not the delivery location of a task in the task set. Instead of always reserving the parking location of the agents we, therefore, need to reserve the last location on the path of an agent. If an agent tries to use a location that is the end of the path of another agent it has to make sure to use that location before the agent that will rest there arrives. Otherwise, the agent needs to find a different path. By reserving the last location of the paths of the agents we partly solve the problem. Agents that still need to plan a path will now only use that location if they can reach it before the agent that will rest on that location but there is still a problem with agents that have already planned their paths. When an agent plans a path it assumes that this path will always be collision-free and therefore never recomputes its path. If agent A has already planned a path that uses a location L_1 but another agent B plans a new path that will end in location L_1 before agent A reaches that destination this might lead to a collision. In order to avoid these collisions, an agent has to check whether any other agent has already planned a path that contains its goal location. If that is the case the agent has to wait until that agent has passed its goal location before it can reach that location and rest there. This is the final modification to the original implementation of the space-time A*-algorithm.

3.2.2 Token Passing with Task Swapping

The Token Passing with Task Swaps (TPTS) algorithm was also implemented from scratch using the pseudo-code provided by (Ma et al., 2017) that is explained in section 2.2. The TPTS algorithm uses the same modified space-time A*-algorithm as the TP algorithm.

3.2.3 Task Allocation and Prioritized Pathplanning

The Task Allocation and Prioritized Pathplanning algorithm in the tool is a modified version of the implementation of (Cawood, 2020)¹. This implementation of the TA-Prioritized algorithm visualizes the planning algorithm using the MESA python library. The first modification that was implemented was to add support for multiple warehouses. The original implementation of the algorithm only supports 2 warehouse environments. In order to support more environments, the LKH-3 TSP solver has to be used to generate new task sequences as this is not supported by this implementation. In order to generate these sequences .TSP files that contain the MAPD instance have to be generated. These files can then be provided to the TSP solver so that it can generate task sequences. In order to be able to generate these .TSP files automatically a script was written to take in a set of tasks and an environment and automatically generate these files that can be provided to the TSP solver. The LKH-3 TSP solver generates .tour files that represent the Hamiltonian Path that the solver found. These files can then be provided to the original implementation of the algorithm so that the path can be converted into task sequences as explained in section 2.2.3.

Parallelizing Prioritized Pathplanning

The last modification that was made to the TA-Prioritized implementation of (Cawood, 2020) was to parallelize part of the algorithm. As described in section 2.2.3 the Prioritized Pathplanning algorithm plans paths for the agents in decreasing priority. It always selects the agent with the highest priority out of the remaining agents and plans a path for it. In the Prioritized Pathplanning implementation of TA-Prioritized, the algorithm calculates the paths of all the remaining agents as if they were the next agent for which a path needs to be planned. This provides the actual execution time of each agent if they were to be selected as the next agent for which a path needs to be planned. The algorithm then selects the agent which has the path with the highest cost (in this case the longest path), removes that agent from the remaining agents, and plans a path for that agent. This process is repeated until the algorithm has planned a path for each agent. Calculating the paths of all the remaining agents in order to determine which agent is going to be selected next is done sequentially in the original implementation of Prioritized Pathplanning. This is not a major problem for this implementation as the algorithm is relatively slow, especially with a large number of agents. There is however a problem in our use-case of the algorithm as our environments support rotations. Adding rotations increases the search space of the algorithm which slightly increases the runtime of planning a single path. This slight increase of the runtime when calculating a path has however a significant impact on the runtime of TA-Prioritized as the algorithm calculates a path for each remaining agent before it selects an agent. This means that in the case of 50 agents the algorithm first plans a path for 50 agents, selects one agent, then plans a path for 49 agents, selects an agent, etc. In order to reduce the runtime of TA-Prioritized, we parallelized the planning of the paths for the remaining agents. The impact of this parallelization is analyzed in the next chapter.

Now that we have an understanding of the contributions of this paper to the visualization tool we can evaluate and compare the different approaches in the context of automated warehouses.

¹<https://github.com/Pieter-Cawood/M-TA-Prioritized-MAPD>

Chapter 4

Results

In this chapter, we evaluate and compare the different approaches that were described in this paper. Different comparisons are made to get a clear understanding of the strengths and weaknesses of each approach. We start off by comparing the scalability of the algorithms in terms of the number of agents. We then analyze the scalability of the algorithms in a larger warehouse environment. Next, we analyze the performance loss of the space-time A*-algorithm that is caused by the addition of rotations. Finally, we compare the original (unparallelized) implementation of the TA-Prioritized algorithm to the parallelized version of the algorithm presented in this paper.

4.1 Metrics

Before we start comparing the algorithms we have to define the metrics that we will use to compare the algorithms. For each of the comparisons mentioned above, different metrics are used. To analyze the scalability of the algorithms in terms of agents and in warehouses of different sizes the following metrics were used:

- **Makespan:** The makespan of an algorithm is the timestep at which all the tasks have been executed and all the agents have stopped moving. (Ma et al., 2017)
- **Average service time:** The service time of a task is the amount of the timesteps between when a task enters the system and when a task is delivered. (Ma et al., 2017)
- **Execution time per timestep:** The execution time of the algorithm is the number of milliseconds between when the algorithm starts the task-allocation phase and when all the tasks have been executed and all the agents have stopped moving. We look at the execution time per timestep instead of the total execution time as this gives us an idea of how long the algorithm needs on average to compute all the actions of all the agents for 1 timestep.

To evaluate the difference in performance between the original space-time A* algorithm and the modified version of the algorithm that supports rotations the following metrics are used:

- **Runtime:** We use the runtime that the algorithm needs to be plan a specific amount of collision-free paths (e.g. 10 paths).
- **Average length of the paths**

- Average size of state space: Lastly we use the average size of the state space as a metric. The state-space contains all the possible states that the algorithm could still explore in order to find the goal state during the search. Each state in the state space represents a possible state for the agent.

Finally to compare the (unparallelized) original implementation of the TA-Prioritized algorithm and the modified (parallelized) TA-Prioritized algorithm we used the following metrics:

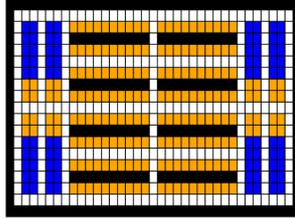
- Runtime per timestep
- Speed-up: the ratio of the runtime of the original implementation and the runtime of the modified implementation

Now that we know which metrics will be used to evaluate and compare the algorithms we present the environments in which the algorithms will be evaluated and compared.

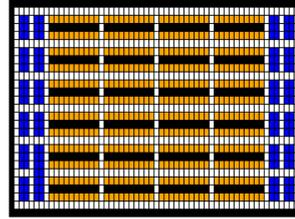
4.2 Evaluation environments

In order to evaluate the algorithms 2 different warehouse environments are used:

- Small warehouse with 192 task endpoints and 80 non-task endpoints
- Large warehouse with 480 task endpoints and 146 non-task endpoints



(a) Small warehouse



(b) Large warehouse

Figure 4.1: Both warehouses used for comparisons: small warehouse contains 192 task endpoints and 80 non-task endpoints, large warehouse contains 480 task endpoints and 146 non-task endpoints

Figure 4.1 is a visual representation of both warehouses. Now that we know which metrics and environments are used to evaluate and compare the algorithms we can look at the first comparison.

4.3 Experimental Results

In this section, we evaluate and compare the algorithms based on the collected metrics in the two warehouse environments.

4.3.1 Scalability in terms of number of agents

We first take a look at the scalability of the algorithms in terms of the number of agents in the warehouse.

In order to test the scalability, we run each algorithm with 5, 10, 20, 30, 40, and 50 agents. All the algorithms are first executed in the small warehouse environment with the same task set of 500 randomly generated tasks. All of the tasks are added to the task set in the first timestep. Figure 4.2 and Table 4.1 report the results of the different algorithms.

Makespan: The first metric we analyze is the makespan of the different algorithms. In general, we can see for all the algorithms that the makespan decreases as the number of agents increases. This is due to the fact that more agents can execute tasks simultaneously so each agent needs to execute fewer tasks on its own. The length of the path of the agents to execute their tasks and the makespan of the algorithm are therefore shorter.

When we look at the makespans of the different algorithms we can see that the TP algorithm consistently has the largest makespan. This is expected as the TP-algorithm is the most basic algorithm of the three. In the TP-algorithm agents select a task based on the distance from the agent’s current location to the pickup locations of the tasks and simply plans a collision-free path to execute that task.

The TPTS-algorithm always produces makespans that are slightly lower than the makespans of the TP-algorithm as the TPTS-algorithm saves time by allowing agents to swap tasks if this can reduce the makespan. We can see that the difference between the makespan of the TP and the TPTS algorithms is larger when there are fewer agents. This is due to the fact that the agents are generally further away from the tasks as there are less agents to execute all the tasks. This allows the agents to swap tasks more often. For larger amounts of agents, the warehouse gets congested and which makes it more difficult for the agents to reach the pickup location quicker than the agent that is already assigned to that task.

From the results, we can also notice that both the TP and the TPTS algorithms have significantly higher makespans than the TA-Prioritized algorithms even though all algorithms know all the tasks beforehand as all the tasks are added to the task set in the first timestep. This is due to the working of the algorithms. The TP and TPTS algorithms only look for the next “best“ task for each agent and ignore possible tasks that can be executed after that task. The TA-Prioritized algorithm plans task sequences for each agent so it does not only look at the next “best“ task but also looks for tasks to be executed after that task. The TA-Prioritized algorithm might therefore decide to go for a task that is slightly further away from its current location if that can reduce the makespan of the algorithm. The TA-Prioritized algorithm also looks for tasks for all the agents at the same time and only gives an agent a task or task sequence when no other agent can execute that task or task sequence more efficiently. This is not the case of the TP algorithm as every agent looks for his “best“ task. This means that in a scenario where we have two agents, agent A and agent B, and we know that agent B is only right next to the pickup location of a task the task might still be executed by agent A if agent A requests the token before agent B. The TPTS algorithm would re-assign the task in this case to give it to agent B as task swapping is allowed in the TPTS algorithm but the same can happen with the TPTS algorithm if the agent is not able to plan a path to an endpoint where it can rest.

Next to comparing the makespans of the algorithms, we can also take a look at the difference in makespan of the same algorithms for 2 different amounts of agents. We can see that the speed-up ($prev_{makespan}/current_{makespan}$) starts at 1.84 for the TP-algorithm. This means that when we double the number of agents (from 5 agents to 10 agents) in the tasks are executed about 85% faster. In a perfect scenario (for example a scenario where collisions were ignored)

1cm

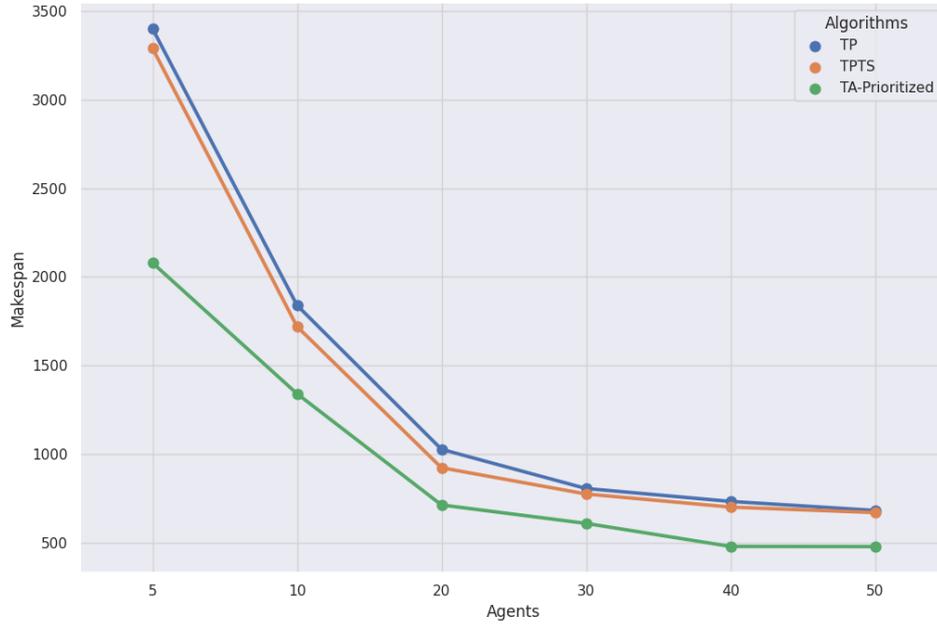


Figure 4.2: Graph of makespans of TP, TPTS and TA-Prioritized for 5, 10, 20, 30, 40 and 50 agents in the small warehouse

agents	TP		TPTS		TA-Prioritized	
	mkspn	speed-up	mkspn	speed-up	mkspn	speed-up
5	3401	/	3290	/	2079	/
10	1840	1.84	1720	1.91	1341	1.55
20	1027	1.80	923	1.86	713	1.88
30	806	1.27	775	1.19	609	1.17
40	733	1.10	701	1.11	479	1.27
50	682	1.07	670	1.04	478	1

Table 4.1: Table of makespans of TP, TPTS and TA-Prioritized for 5, 10, 20, 30, 40 and 50 agents in the small warehouse

the speed-up would be equal to 2 as we have twice as many agents available, but this is not the case here as agents sometimes stand in each other's way. We can clearly see that the more agents we add the more congested the warehouse is as the speed-up keeps on decreasing. This is expected especially if we consider that these results were collected in the small warehouse which is relatively small for 50 agents.

The same can be said about the TPTS-algorithm as the makespan of the TPTS algorithm is generally only slightly lower than the makespan of the TP-algorithm.

For the TP and the TPTS algorithm, we can see that the speed-up keeps on decreasing as the number of agents increases due to the increased congestion of the warehouse. For the TA-Prioritized algorithm, this is not really the case. We can for example see that going from 10 to 20 agents has a significantly higher speed-up than going from 5 to 10 agents. This can be explained in 2 ways. The first reason for this varying speed-up is the working of the Prioritized Pathplanning algorithm that TA-Prioritized uses. As section 2.2.3 explains the TA-Prioritized algorithm orders the agents in decreasing order of priority. The first agents for which paths are planned are the agents with the largest estimated execution times. This is done to avoid increasing the actual execution times of these agents as the makespan will be equal to the largest actual execution time of all the agents. Agents for which paths are planned later on will have to wait in order to avoid collisions which increases their actual execution times. This way of planning paths is less impacted by the congestion of warehouses as the agents with the lowest actual execution times have to wait in order to avoid collisions. This has a less important impact on the makespan of the algorithm than in the case of the TP and the TPTS algorithms.

The second reason why the speed-ups of the TA-Prioritized algorithm do not always decrease when the amount of agents is increased is due to the fact that the TSP solver is not always able to find the best task sequences in the time it was given (in this case 1000s). This leads to some agents having less optimal task sequences than others. For 50 agents for example the average length of the path of each agent was +/- 250 but due to some agents having received more tasks than others from the TSP solver some agents had very small paths (for example 60) because they had only 1 or 2 tasks to execute while others had larger paths (for example 478) as they were executing a lot more tasks. This is however something that we can not observe by only looking at the makespan of the algorithms. One of the ways to partially solve this problem is to give the TSP solver more time to compute better task sequences for the agents.

Service Time: From the results of table 4.2 and Figure 4.3 we can see that the average service time of the algorithms is closely related to the makespan of the algorithm. This is expected as all of the tasks are made available to all the algorithms at the beginning of the algorithms. The service time of a task is therefore equal to the timestep at which the task was delivered. As the makespan is the first timestep at which all the tasks are delivered and all the agents stopped moving these two will be correlated.

Runtime per timestep: Another interesting metric when looking at the scalability of the algorithms is the runtime per timestep. The runtime per timestep gives us an idea of how long the algorithm need on average to compute actions for all the agents in a timestep. This metric is especially important for the TP and TPTS algorithms as the TA-Prioritized does all of its computations at the start of the algorithm. The TP and TPTS algorithms look for tasks and compute new paths for agents when an agent has reached the delivery location of its task so the computations happen during the execution of the program. A high runtime per second means that the algorithm will need to pause the movement of the agents in order to compute the actions of all the agents. Before we take a look at the results of table 4.3 and figure 4.4 it is important to remember that these metrics have been collected in Python, a dynamically typed programming

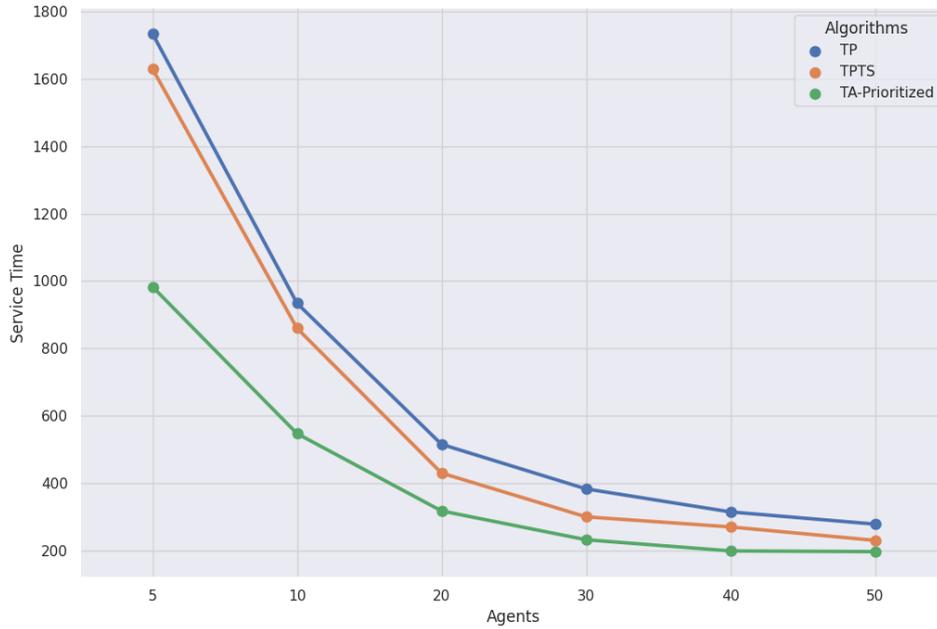


Figure 4.3: Graph of average service times of TP, TPTS and TA-Prioritized for 5, 10, 20, 30, 40 and 50 agents in the small warehouse

agents	TP		TPTS		TA-Prioritized	
	service time	speed-up	service time	speed-up	service time	speed-up
5	1735	/	1630.37	/	982.99	/
10	934	1.86	860.09	1.90	547.23	1.80
20	515	1.81	430.76	2.00	317.54	1.72
30	383	1.27	301.03	1.43	231.99	1.37
40	314	1.01	273.98	1.10	196.66	1.80
50	278	1.07	231.65	1.18	196.52	1.00

Table 4.2: Table of average service time of TP, TPTS and TA-Prioritized for 5, 10, 20, 30, 40 and 50 agents in the small warehouse

language. In a real-world environment, one would implement these approaches using a more appropriate programming language such as the original implementation of the algorithms in C++ (Ma et al., 2017).

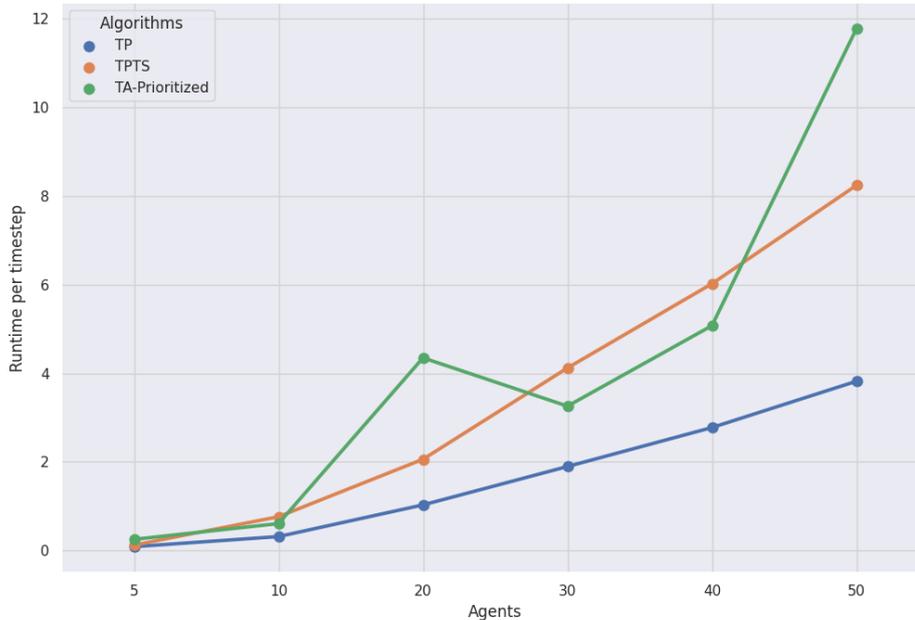


Figure 4.4: Graph of runtimes per timestep of TP, TPTS and TA-Prioritized for 5, 10, 20, 30, 40 and 50 agents in the small warehouse

agents	TP		TPTS		TA-Prioritized	
	runtime/t	slowdown	runtime/t	slowdown	runtime/t	slowdown
5	0.09s	/	0.12s	/	0.25s	/
10	0.32s	3.55	0.76s	6.33	0.61s	2.44
20	1.03s	3.22	2.06s	2.71	4.35s	7.13
30	1.90s	1.84	4.14s	2.01	3.26s	0.75
40	2.78s	1.46	6.02s	1.45	5.08s	1.56
50	3.82s	1.37	8.24s	1.39	11.79s	2.32

Table 4.3: Table of run-times per timestep of TP, TPTS and TA-Prioritized for 5, 10, 20, 30, 40 and 50 agents in the small warehouse

The first thing we can notice when looking at the results of table 4.3 is that increasing the number of agents in the warehouse in any algorithm significantly increases the runtime per timestep. We can for example see that the service time per timestep for the Token Passing algorithm with 5 agents is about 80ms while the service time per timestep for 50 agents is about

4 seconds. This means that the algorithm on average needs 4 seconds in each timestep to decide actions for all the agents in a warehouse containing 50 agents. This increase is expected as the algorithm needs to look for more tasks and calculate more paths in each timestep as there are more agents.

This is an important metric to take into account if we consider using these algorithms in a real-world environment as this indicates that for 50 agents the algorithm needs on average 4 seconds before it can compute the actions that need to be executed. If the agents in our real-world environment can move from one position to another in less than 4 seconds this means that our agents will need to wait on our algorithm to compute the next actions. This is not optimal especially in our visualization where agents only need 1 second to move in between two locations. Running the visualization with 50 agents causes the algorithm to “lag“ in each timestep while the actions are computed.

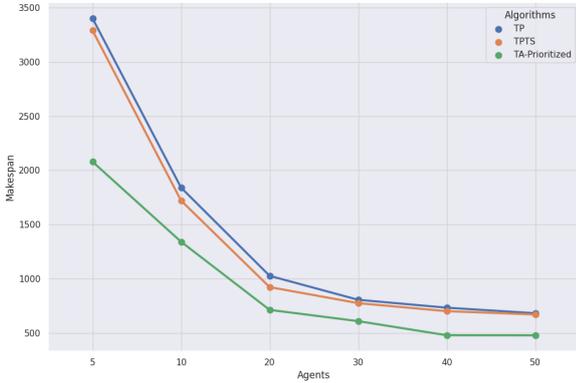
This can however be adapted in the visualization by changing the number of time agents need to move in the configuration file. It is also important to consider that the results presented in this paper are computed using Python implementations of the algorithms. Implementations in other programming languages such as (Ma et al., 2017) reports that their C# implementation of the TP algorithm never produced running times per timestep larger than 10ms and that the TPTS algorithm never produced running times per timestep larger than 200ms which are significantly more acceptable.

Conclusion Based on the results of this section we can conclude that in smaller warehouse environments the algorithm that produces the best results is the TA-Prioritized algorithm as it consistently produces significantly smaller makespans and average service times than the TP and TPTS algorithms. When we look at the runtime per timestep of the 3 algorithms we can however see that this lower makespan and service time comes at a cost as the running time per timestep of the TA-Prioritized algorithm is 3 times larger than the running time per timestep of the TP algorithm. As the TA-Prioritized algorithm can be ran beforehand this is however still acceptable. The TA-Prioritized algorithm should therefore always be the first algorithm to consider in a smaller warehouse environment. The TPTS algorithm is the second best algorithm in terms of makespan and service time. It produces makespans and service times slightly smaller than the makespans and service times of the TPTS algorithm but takes on average twice as long time. In our simulated environments where agents need 1000ms to execute the actions that they are given we could use the TPTS algorithm until 10 agents without experiencing any lag. For the TP-algorithm we could use up until 20 agents with minor lag. We can see that if we only look at the runtime per timestep of both algorithms it is better to use the TP-algorithm with 20 agents than the TPTS with 10 agents in the smaller warehouse environment. For our small warehouse environment it is therefore better to use the TP algorithm for a large number of agents due to its running time even though it produces slightly larger makespans and service times than the TPTS algorithm.

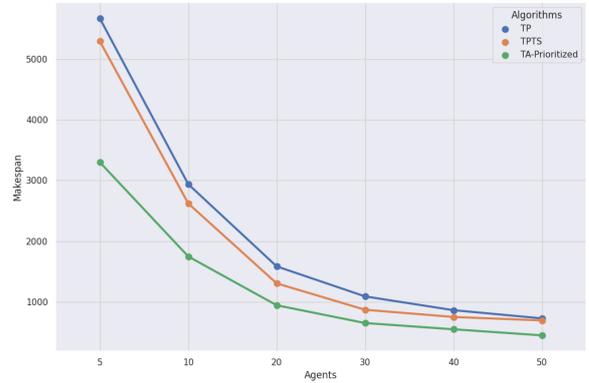
4.3.2 Scalability in terms of warehouse size

In this section we compare the results of the previous section which were collected in the small warehouse environment with results of the algorithms in the large warehouse environment. This warehouse environment is twice as large as the small warehouse.

Makespan: When we compare the makespan of the algorithms in the small warehouse environment with the makespan of the algorithms in the large warehouse environment we can see from



(a) Small warehouse environment



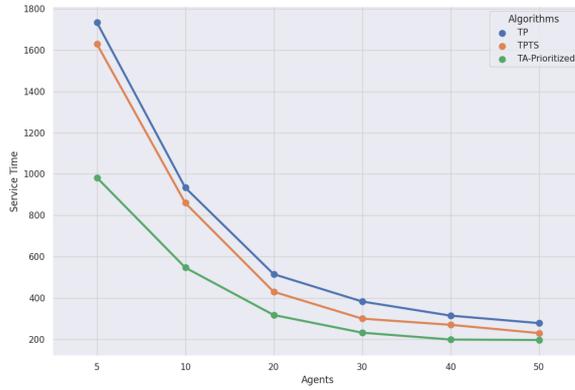
(b) Large warehouse environment

Figure 4.5: Makespan of 5, 10, 20, 30, 40 and 50 agents of TP, TPTS and TA-Prioritized in the small and large warehouse environments

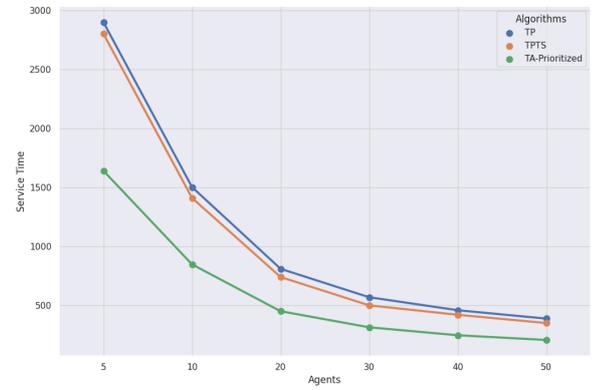
Figure 4.5 that there is a similar trend. The makespans in the larger warehouse environment are significantly larger than the makespans in the small warehouse environment but we can see that the difference in makespan between the different algorithms is similar in both warehouse environments. We can see that the difference in makespan between the TP and the TPTS algorithm in the large warehouse environment is slightly larger than the difference in the smaller warehouse environment. This is expected as the length of the average paths to the pickup locations of tasks is larger. This allows agents to reduce the makespan more when they swap tasks. We can also see that the difference between the three algorithms for 50 agents in the larger warehouse environment is smaller than the difference between the three algorithms for 50 agents in the small warehouse environment. This is due to the fact that the TA-Prioritized algorithm is able to deal with congestion better than the other 2 algorithms due to the Prioritized Pathplanning algorithm. The TP and TPTS algorithms however were more limited due to congestion in the small warehouse environment. As the large warehouse environment is twice as large and the number of agents and tasks is the same in both environments this gives the algorithms more space and limits the congestion which allows them to produce smaller makespans.

Service Time Similarly to the service time of the different agents in the small warehouse environment we can see from Figure 4.6 that the service time of the algorithms in the large warehouse environment follows the same trend as the service time of the algorithms in the small warehouse environment. This is expected as the service time of the tasks is equal to the timestep at which the task was delivered as all the tasks are released at the beginning. The makespan is the first timestep at which all tasks have been delivered and the agents have stopped moving. It therefore makes sense that the average service time of the tasks follows the same trend as the makespan.

Runtime per timestep Finally, we look at the runtime per timestep of the different algorithms in Figure 4.7. This is the first metric where we can see a clear difference between the results in the small warehouse environment and the results in the large warehouse environment.

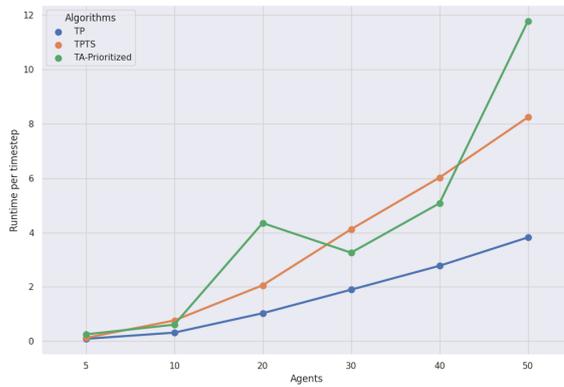


(a) Small warehouse environment

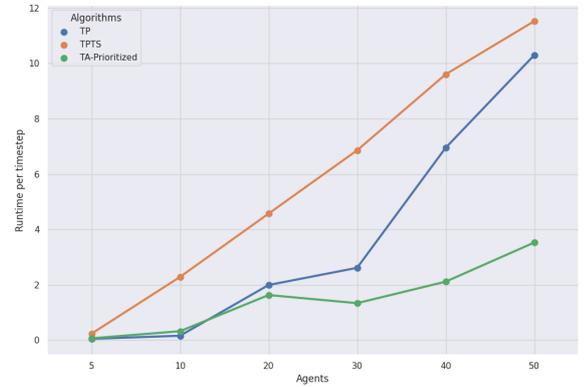


(b) Large warehouse environment

Figure 4.6: Service time of 5, 10, 20, 30, 40 and 50 agents of TP, TPTS and TA-Prioritized in the small and large warehouse environments



(a) Small warehouse environment



(b) Large warehouse environment

Figure 4.7: Runtime per timestep of 5, 10, 20, 30, 40 and 50 agents of TP, TPTS and TA-Prioritized in the small and large warehouse environments

We can see that in the small warehouse environment the TA-Prioritized algorithm has a significantly larger runtime per timestep than the TP and the TPTS algorithm while in the large warehouse environment we can see that the TA-Prioritized algorithm has the smallest runtime per timestep. This is due to the larger size of the warehouse which makes it easier for the path-planning algorithm to plan paths. The calculation of actual execution times for all the agents in the Prioritized Pathplanning algorithm of TA-Prioritized is parallelized, but before the next agent can be chosen all the actual execution times have to be calculated. This means that if planning the path of any of the agents take significantly more time than planning the paths of the other agents the benefits of the parallelization are lost. As the large warehouse environment is twice as large as the small warehouse environment is is significantly easier to plan collision-free paths which reduces the time it takes to plan paths for the agents. This explains the significantly lower runtime per timestep of the TA-Prioritized algorithm.

Conclusion: When we compared the algorithms in a small warehouse environment in the previous section we noted that the best algorithm in terms of makespan and service time was the TA-Prioritized algorithm but that it had a significantly larger runtime per timestep than the TP and TPTS algorithms. This is not the case anymore in the large warehouse environment which makes it clear that the TA-Prioritized algorithm is by far the best algorithm to use in a larger warehouse environment. If the TA-Prioritized algorithm can not be used because the tasks can not be known beforehand the TPTS algorithm produces again better makespans and average service times than the TP algorithm which makes it the second best algorithm in terms of makespan and average service time. The difference between both algorithms in terms of runtime per timestep is also significantly lower when we look at larger number of agents such as 40 and 50 agents. From this we can conclude that the TPTS algorithm performs better in larger warehouse environments than the TP algorithm.

4.3.3 Performance impact of rotations

One of the modifications that were made to the space-time A* algorithm that is used by the three algorithms is the addition of rotations. This modification significantly slowed down the performance of the algorithm. In order to measure the performance loss caused by the addition of rotations, we collected the following metrics for both implementations: the average length of paths planned, the average size of the search space during the planning of a path, and the average time it takes for the algorithm to come up with a path.

paths	A* without rotations			A* with rotations		
	runtime	length	state space	runtime	length	state space
10	2.95ms	18	59	34.25ms	21	216
20	15.48ms	22	103	55.18ms	25	257
40	37.99ms	22	94	145.91ms	25	352
80	56.90ms	25	123	169.73ms	26	317

Table 4.4: Table of average runtime per timestep of TP, TPTS and TA-Prioritized for 5, 10, 20, 30, 40 and 50 agents in the small warehouse

From the results of Table 4.4 we can see that even though the average length of the planned paths is not significantly larger when adding rotations, the average runtime needed to plan the paths and the average size of the state space are significantly larger. Planning 10 collision-free

paths using the A* implementation without the rotations only takes about 3ms while this takes 10 times longer with the A* implementation with rotations. The state space, all the possible states that the algorithm can select to expand during the search, is also 4 times larger for the A* implementation with rotations. We can therefore conclude that rotations have a big impact on the performance of the algorithms and should only be used when necessary to avoid a major performance loss.

4.3.4 Performance of parallelized Prioritized Path-planning

As we have seen in the previous section the addition of rotations to the space-time A* algorithm significantly increases the runtime of planning a single path. This decreases the performance of the three algorithms: TP, TPTS, and TA-Prioritized as they all use the same implementation. In section 3.2.3 we mention that the performance loss due to the addition of rotations is especially important for the TA-Prioritized algorithm due to the working of the algorithm and we propose in that section to parallelize part of the algorithm. In this section we measure the difference in performance between the modified implementation of TA-Prioritized that uses parallelization and the original implementation of TA-Prioritized that does all the computations sequentially. Table 4.5 presents these results.

	Non-Parallelized	Parallelized	
agents	runtime/t	runtime/t	speed-up
5	0.45s	0.25s	1.80
10	2.29s	0.61s	3.75
20	36.48s	4.35s	8.38

Table 4.5: Table of average runtime per timestep of TA-Prioritized with and without parallelization for 5, 10, 20 agents in the small warehouse

From the results of table 4.5 we can clearly see why this last modification was needed if we plan on using the TA-Prioritized algorithm with a large number of agents. We can see that the speed-up is relatively small for a small number of agents such as 5 as the algorithm only needs to plan 14 paths and there is a relatively low number of constraints as the algorithm has at most 4 other paths that it needs to avoid collisions with. When we look at a larger amount of agents such as 20 we can see that the non-parallelized algorithm takes almost twice as long to compute the paths of the 20 agents as the parallelized algorithm takes to compute the paths for 50 agents. A runtime per timestep of 19 for 20 agents corresponds to about 4 hours as the makespan of the algorithm for 20 agents was 713 as we can see in table 4.5. Even though we can calculate all these paths beforehand as the TA-Prioritized algorithm is an algorithm for the offline variant of the MAPD problem this runtime is already extremely large for the number of agents. Especially when we look at the fact that the runtime per timestep for 20 agents is 8 times larger than the runtime per timestep for 10 agents. Based on these results we can expect the algorithm to take more than 12 hours to compute the paths of 50 agents in the relatively small warehouse environment that we used. Based on these results we can therefore conclude that running the original implementation of the TA-Prioritized algorithm is not an option if we want to support a large number of agents. In that case, the modified TA-Prioritized algorithm is a better choice.

Chapter 5

Conclusion

In this paper, we analyzed, implemented and visualized the MAPD-problem and three approaches to the problem: the Token Passing (TP) algorithm, the Token Passing with Task Swapping (TPTS) algorithm and the Task-Allocation and Prioritized Pathplanning (TA-Prioritized) algorithm. We implemented and visualized these approaches by using the Unity game engine and the ML-agents platform to create a tool that is able to support various warehouse environments. Finally, we compared these approaches in the setting of automated warehouses and concluded that the TA-Prioritized algorithm offers the best performance in smaller and larger warehouse environments. When the TA-Prioritized algorithm can not be used we concluded that the TP algorithm is more suitable for a larger number of agents in smaller warehouse environments while the TPTS algorithm is more suited for larger warehouse environments. We also concluded that supporting the rotations of agents significantly reduces the performance of the algorithms and rotations should only be used if necessary. Finally we concluded that the parallelized implementation of the TA-Prioritized algorithm outperforms the original implementation of (Cawood, 2020) when rotations are added to the algorithms.

Future Work: The tool currently supports three different approaches to the MAPD problem but can be extended with others approaches. In this paper we for example only focused on one algorithm for the offline variant of the MAPD problem, the TA-Prioritized algorithm, while there exists other approaches such as the Task-Allocation and Hybrid Path Planning algorithm (Ma et al., 2017). Other planning approaches such as the idea of converting the MAPD problem to a Spatial Task Allocation Problem proposed by (Claes et al., 2017) can also be considered. Next to the traditional planning algorithms that this paper focused on other approaches such as learning approaches (e.g reinforcement learning (Sutton & Barto, 1998)) could also be considered.

Bibliography

- AbuShawar, B., & Atwell, E. (2015). Alice chatbot: Trials and outputs. *Computación y Sistemas*, 19(4), 625–632.
- Calvaresi, D., Marinoni, M., Sturm, A., Schumacher, M., & Buttazzo, G. (2017). The challenge of real-time multi-agent systems for enabling iot and cps. *Proceedings of the international conference on web intelligence*, 356–364.
- Cawood, P. (2020). M-TA-Prioritized-MAPD. <https://github.com/Pieter-Cawood/M-TA-Prioritized-MAPD>
- Claes, D., Oliehoek, F., Baier, H., Tuyls, K., et al. (2017). Decentralised online planning for multi-robot warehouse commissioning. *AAMAS'17: PROCEEDINGS OF THE 16TH INTERNATIONAL CONFERENCE ON AUTONOMOUS AGENTS AND MULTIA-AGENT SYSTEMS*, 492–500.
- Dresner, K., & Stone, P. (2008). A multiagent approach to autonomous intersection management. *Journal of artificial intelligence research*, 31, 591–656.
- Helsgaun, K. (2017). An extension of the lin-kernighan-helsgaun tsp solver for constrained traveling salesman and vehicle routing problems. *Roskilde: Roskilde University*.
- Juliani, A., Berges, V.-P., Vckay, E., Gao, Y., Henry, H., Mattar, M., & Lange, D. (2018). Unity: A general platform for intelligent agents. *arXiv preprint arXiv:1809.02627*.
- Liu, M., Ma, H., Li, J., & Koenig, S. (2019). Task and path planning for multi-agent pickup and delivery. *AAMAS*, 1152–1160.
- Ma, H., Hönig, W., Kumar, T. S., Ayanian, N., & Koenig, S. (2019). Lifelong path planning with kinematic constraints for multi-agent pickup and delivery. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33, 7651–7658.
- Ma, H., Li, J., Kumar, T., & Koenig, S. (2017). Lifelong multi-agent path finding for online pickup and delivery tasks. *arXiv preprint arXiv:1705.10868*.
- Manate, B., Munteanu, V. I., & Fortis, T.-F. (2013). Towards a scalable multi-agent architecture for managing iot data. *2013 Eighth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, 270–275.
- Osaba, E., Diaz, F., Onieva, E., López-García, P., Carballedo, R., & Perallos, A. (2015). A parallel meta-heuristic for solving a multiple asymmetric traveling salesman problem with simultaneous pickup and delivery modeling demand responsive transport problems. *International Conference on Hybrid Artificial Intelligence Systems*, 557–567.
- Rabuzin, K., Maleković, M., & Bača, M. (2006). A survey of the properties of agents. *Journal of Information and Organizational Sciences*, 30(1), 155–170.
- Shalev-Shwartz, S., Shammah, S., & Shashua, A. (2016). Safe, multi-agent, reinforcement learning for autonomous driving. *arXiv preprint arXiv:1610.03295*.
- Silver, D. (2005). Cooperative pathfinding. *AIIDE*, 1, 117–122.
- Sutton, R., & Barto, A. (1998). *Reinforcement learning: An introduction*. MIT Press.

- Van Den Berg, J. P., & Overmars, M. H. (2005). Prioritized motion planning for multiple robots. *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 430–435.
- Wooldridge, M., & Jennings, N. R. (1994). Agent theories, architectures, and languages: A survey. *International Workshop on Agent Theories, Architectures, and Languages*, 1–39.
- Yoon, S., & Kim, J. (2017). Efficient multi-agent task allocation for collaborative route planning with multiple unmanned vehicles. *IFAC-PapersOnLine*, 50(1), 3580–3585.