FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Automated Planning for Warehouse Logistics
Student:	Bc. Klára Dvořáková
Supervisor:	doc. RNDr. Pavel Surynek, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2021/2022

Instructions

The task is to develop planning techniques for warehouse logistics. We assume a warehouse with a group of mobile robots that move across the warehouse. A typical task for a robot is to transport an item from its storage location to a packing location. The significant challenge in the context of a warehouse is to plan for multiple robots so they fulfil a stream of tasks arriving online. Specifically, the problem is how to respond to priority tasks. The robots must not conflict with each other. Tasks for the student are as follows:

1. Study relevant literature on warehouse planning and logistics with an emphasis on coordination of multiple robots fulfilling stream of tasks and tasks with priorities.

2. Modify an existing method or develop a new method for coordination of warehouse robots that can handle priorities effectively.

3. Implement the new method as a software prototype and compare it with a relevant alternative algorithm. Analyze the results.

–

[1] Minghua Liu, Hang Ma, Jiaoyang Li, Sven Koenig: Task and Path Planning for Multi-Agent Pickup and Delivery. AAMAS 2019: 1152-1160

[2] John Enright, Peter R. Wurman: Optimization and Coordinated Autonomy in Mobile Fulfillment Systems. Automated Action Planning for Autonomous Mobile Robots 2011

[3] Felix Weidinger, Nils Boysen, Dirk Briskorn: Storage Assignment with Rack-Moving Mobile Robots in KIVA Warehouses. Transp. Sci. 52(6): 1479-1495 (2018)

[4] Pavel Surynek: Unifying Search-based and Compilation-based Approaches to Multi-agent Path Finding through Satisfiability Modulo Theories. IJCAI 2019: 1177-1183

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 29 November 2020 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Automated Planning for Warehouse Logistics

Bc. Klára Dvořáková

Department of Theoretical Computer Science
Supervisor: doc. RNDr. Pavel Surynek, Ph.D.

April 30, 2021

Acknowledgements

I would like to thank my supervisor doc. RNDr. Pavel Surynek, Ph.D. for valuable advice and help with problems that I faced while writing my thesis. I would also like to thank to my family and friends who supported me during my studies.

Declaration

Citation of this thesis

Dvořáková, Klára. *Automated Planning for Warehouse Logistics.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

Tato práce se zabývá automatickou skladovou logistikou. Práce se zaměřuje na pohyb robotů ve skladu tak, aby nedocházelo ke kolizím a roboti efektivně plnili své úkoly. Typickým úkolem je přesun položky z místa uskladnění na místo výdejní. Hlavním cílem práce je zanalyzovat a vyvinout modifikaci existujících metod se zaměřením na úkoly s různými prioritami. Literární rešerše se zabývá problémem hledání cest pro více agentů a popisem existujících algoritmů řešících tento problém. Praktická část práce navazuje analýzou a implementací vylepšení Windowed CBS algoritmu tak, aby efektivně zpracovával úkoly s různými prioritami. Závěrečná část práce je věnována experimentům na různých mapách s různým počtem agentů a analýze jejich výsledků. Výsledkem práce je softwarový prototyp Windowed Priority CBS, který přijímá úkoly s různými prioritami a tyto priority bere v potaz při jejich řešení.

Klíčová slova MAPF, lifelong MAPD, CBS, prioritizace, automatický sklad

Abstract

This thesis describes automated warehouse logistics. The paper focuses on the coordination of the robot in a warehouse so there are no collisions and the robots fulfil their tasks effectively. A typical task is to move an item from its storage location to its delivery location. The main goal of the thesis is to analyze and develop a modification of existing algorithms with a focus on tasks with different priorities. The theoretical part of this paper deals with a multi-agent pathfinding problem and a description of existing algorithms that solve this problem. The practical part of the paper follows up with the analysis and implementation of Windowed CBS improvements so the algorithm would process tasks with different priorities effectively. Finally, experiments are run on several maps with a various number of agents, and the results are evaluated and analyzed. The outcome of the thesis is a software prototype of Windowed Priority CBS that accepts tasks with different priorities and takes said priorities into account when searching for the solution.

Keywords MAPD, lifelong MAPD, CBS, prioritization, automated warehouse

Contents

List of Figures

List of Tables

Introduction

During the past years, the number of people using e-shops to buy goods has increased. As a response to the higher demand, online shops need to enlarge their warehouses and speed up the logistics processes inside their them. One way to make warehouse logistics more efficient is to engage robots in the process and make the warehouse automated as we can see in picture 0.1.

The highest increase in the customer demand arose last year, mainly as a consequence of the closing of many retail stores due to the covid-19 pandemic. The demand grew substantially, and some stores are having troubles shipping all the orders in a reasonable time. Furthermore, some items need to be shipped faster than others, or some members need to have their orders delivered fast. Here is where priorities come into play. With an automated warehouse, we can incorporate different priorities of orders and have them shipped as fast as needed.

This thesis focuses on warehouse logistics. We specifically focus on lifelong multi-agent pathfinding with priorities. We will analyze different approaches to priorities and build a software prototype to coordinate warehouse robots that can handle priorities effectively.

In the theoretical part, we will focus on describing the multi-agent pathfinding problem (MAPF) and the multi-agent pickup and delivery problem (MAPD). We will discuss different algorithms to solve these two problems, emphasizing the constraint-based search (CBS) algorithm. Next, we will define the MAPD problem with task priorities. We will focus on different algorithm modifications to make our new algorithm modification solve tasks with priorities effectively.

In the practical part, we will gather information from the priority analysis, and we will implement various algorithm modifications of the Windowed CBS algorithm. We will test the implemented modifications on various warehouse maps on several instances and with a varied number of robots. Finally, we will evaluate the experimental results and discuss what modifications handle priorities effectively.

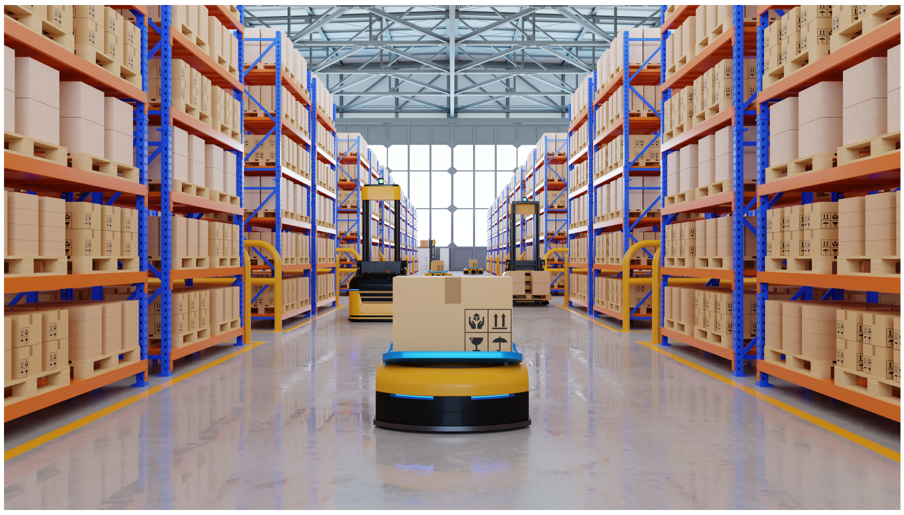

Figure 0.1: Example of an automated warehouse (resource: MAX 3D Design/Shutterstock.com)

Chapter **1**

Background

1.1 Multi-Agent Pathfinding

Multi-agent pathfinding (MAPF) is the problem that focuses on finding paths for multiple agents so that every agent reaches its goal and agents do not collide. We can see multiple real-world applications of this problem in different areas, such as warehouse logistics [1], robotics [2], digital entertainment [3], autonomous vehicles [4] and even some focused on real-time computer games [5], [6] or [7].

In this chapter, we will focus on the offline version of this problem. Offline MAPF means that all tasks that will be processed are known from the very beginning of the solution, and no new tasks are added throughout the computation of the solution. In the following sections, we will discuss MAPF problem definitions, solution evaluation, and we will look at some algorithms that solve offline MAPF problems. [8]

1.1.1 MAPF Formal Definition

MAPF problem can be defined in more ways. In this paper, we will focus on the classical definition of the problem.

In a classical MAPF problem, we define k agents with a tuple $\langle$ G, i, g $\rangle$, where:

- G = (V, E) is an undirected graph with vertices V and edges E. Vertices represent all the possible locations that agents can occupy. Edges represent paths between locations; each edge represents one path from one vertex to another without passing through any other vertex.
- i is a function that maps an agent to its initial position.
- g is a function that maps an agent to its goal (target location).

Time is discretized into timesteps. Each agent performs one action in every timestep. There are two types of actions:

- *move*: The agent is moving from its current location to another vertex that is connected by an edge to the current position.

- *wait*: The agent stays in its current position for one timestep.

Single-agent solution can be defined as a sequence of actions $\pi_i = (\mathbf{a}_1, ..., \mathbf{a}_n)$ for agent i, where the first position of the agent is $s(i)$ and the ending position is $g(i)$. Solution to the MAPF problem is then the joint single-agents solutions of all k agents. This solution is not necessarily a valid solution. To define a valid solution, we first need to discuss and define the two main types of conflicts that can occur between agents.

These conflicts are:

- **Vertex Conflict**: This conflict happens when more than one agent occupies a single vertex at the same timestep. Formally then: there is a vertex conflict between agent i and agent j if there exists such a timestep t, where $\pi_i[t] = \pi_j[t]$. Picture 1.2 demonstrates a vertex conflict.

- **Edge Conflict**: This conflict occurs if two agents are swapping their locations and are using the same edge. Formally then: there is an edge conflict between agent i and j if there exists timestep t, where $\pi_i[t+1] = \pi_j[t]$ and $\pi_j[t+1] = \pi_i[t]$. Picture 1.1 demonstrates an edge conflict.

There are more types of conflicts, such as different types of edge conflicts (following agent) or circle conflicts (all agents follow others in a circle). To properly define the MAPF problem, we need to define what conflicts are going to be considered restrictions. The least of the restrictions is to forbid edge conflict (swapping conflict). Most of the previous works also forbid vertex conflicts.

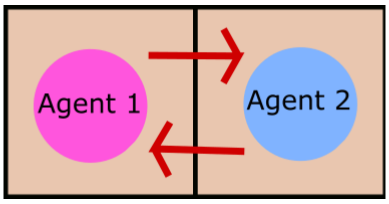

Figure 1.1: Edge conflict

Now we can define a valid solution to the MAPF problem as a solution that does not have any restricted conflicts.

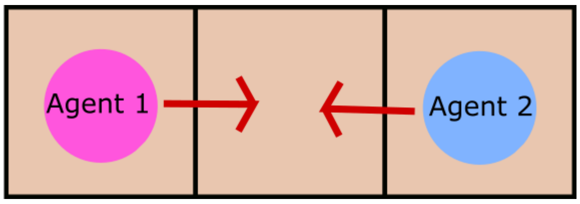

Figure 1.2: Vertex conflict

1.1.2 MAPF Solution Evaluation

In the second part of the chapter, we will discuss different types of solution evaluation. There are two most commonly used objective functions for solution evaluation.

- *Makespan*: This function returns the number of timesteps from the initialization until the last step of the last agent. Formally then, makespan for a solution $\pi = (\pi_1, ..., \pi_k)$ is equal to $max_{1 \leq i \leq k} |\pi_i|$.
- *Sum of costs*: This function returns the sum of all timesteps over all the agents. Formally then the *sum of costs* for solution π is defined as $\sum_{1 \leq i \leq k} |\pi_i|$.

[8]

MAPF Example

Let us take an example of a simple MAPF problem 1.3. The graph $G = (V, E)$ is here a four-connected grid map. Agents can move to their neighbour locations, but they cannot cross to the corner connected locations in one move. Black squares symbolise ”blocked” locations, meaning agents cannot visit these locations. We can see that our problem instance has two agents $A = (a_1, a_2)$, where agents are defined as:

- $a_1 = (G, 5, 7)$
- $a_2 = (G, 8, 5)$.

Intuitive shortest paths for agents are:

- $\pi_1 = \{\{M, 6\}, \{M, 7\}\}$
- $\pi_2 = \{\{M, 7\}, \{M, 6\}, \{M, 5\}\}$

M stands for action move and W stands for action wait.
Evaluation of the joint solution $\pi = (\pi_1, \pi_2)$ is then:

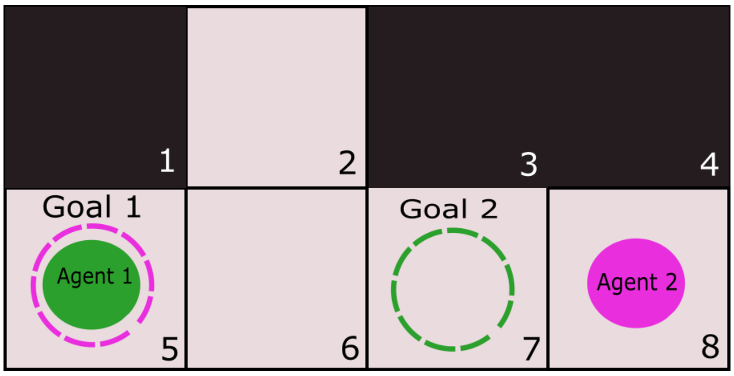

Figure 1.3: MAPF problem example

- $makespan = max(|\pi_1|, |\pi_2|) = max(2, 3) = 3$
- $sumofcosts = |\pi_1| + |\pi_2| = 2 + 3 = 5$

We can see that there is a problem with the solution and that it is not a valid solution. There is an edge conflict in the second timestep, where $\pi_1[2] = \pi_2[1]$ and $\pi_2[2] = \pi_1[1]$. To find a valid solution, we need to resolve this conflict. There is an intuitive resolution of this conflict: if we move the agent a_1 from location 6 to location 2 instead of 7, we can avoid this conflict. The improved path for agent a_1 is now $\pi_1 = \{\{M, 6\}, \{M, 2\}, \{M, 6\}, \{M, 7\}\}$, the path for agent a_2 stays the same. Now we can see there are no more conflicts, and this solution can be considered valid.

Let us evaluate the new solution. Only the path of agent a_1 changed:

- $|\pi_1| = 4 \rightarrow makespan = max(4, 3) = 4$
- $sumofcosts = 4 + 3 = 7$.

We could, of course, change the path of the agent a_2; this agent has no more options to move from location 7 other than location 6, but going back to location 8. Modifying agent a_2 would lead to agent a_2 waiting in its current location. $\pi_2 = \{\{M, 7\}, \{W, 7\}, \{M, 6\}, \{M, 5\}\}$. From this, we can see that this move would result in a vertex conflict in location 6. It is evident, then, that finding a valid solution by modifying agent a_2 is, in this example, more complicated.

1.1.3 MAPF Algorithms

Multi-agent pathfinding is a widely studied area; there are many algorithms capable of solving this problem. We can divide these algorithms into different categories based on different parameters and the way they are operating. There are centralised methods that treat all agents as a single entity [9] and decoupled methods that plan each agent independently and then search for the conflicts [10] or [11]. These methods have different properties, and we can divide algorithms based on their optimality, completeness and time and space complexity.

Example of an *fast* algorithm is prioritized planning [12]. We consider an algorithm to be a *fast* algorithm if the worst-case time complexity is polynomial in the size of the graph G and exponential in the number of the agents [8].

Prioritized planning is a simple algorithm that sorts tasks based on their priority and then plans them one by one. It holds a table where all the locations with the timesteps of all the planned agents are stored. Agents that are being planned are using this table to avoid conflicts with the previously planned agents. There are different ways to set tasks priorities discussed in [13] and [14]. We can see that this algorithm is neither complete nor optimal.

Another example of a *fast* algorithm is MAPP [15]. This algorithm is not optimal but is complete under some circumstances. Other examples are Push-and-Swap [16], or Push-and-Rotate [17]. Even though these two algorithms are complete, they are not optimal.

Optimal Algorithms

The optimal solvers can be classified as follows:

- Extensions of A*
- Increasing Cost Tree Search (ICTS)
- Constraint Programming
- Constraint-Based Search (CBS)

The first group of optimal algorithms is based on A*. This group includes operator independence detection algorithm [18] or M* algorithm [19]. These algorithms are extending an A* algorithm by time domain to search collision-free paths.

The ICTS algorithm [20] divides the search into a high-level and low-level search. The high-level is represented *Increasing Cost Tree* (ICT). In the first step, it finds an optimal solution for all single agents (low-level search). The path lengths (= costs) are then saved to the root of the ICT. The search continues by adding increasing one of the costs in the node by one. The tree is increasing the cost of the joint solution until the low-level search finds a valid solution.

MAPF problem can also be solved by a general-purpose constraint solver. The problem is usually modelled as a Constraint Satisfaction Problem or Constraint Optimization Problem. General-purpose constraint solver then assures a collision-free path [21].

CBS algorithm will be described more deeply since we will build our solution on this algorithm.

CBS

Conflict-based search (CBS) is a complete and optimal MAPF problem solver for well-formed instances [22]. Its idea is to decompose the problem into a single-agent pathfinding problem. In single-agent pathfinding, constraints are used to guarantee the non-existence of conflicts with other agents. The algorithm can be divided into two levels; the high and the low level. High-level stores the constraint tree (CT), and the low level takes care of the single-agent pathfinding.

A constraint is a tuple (a_i, v, t) (resp. (a_i, v_1, v_2, t)) that prohibits agent a_i to visit vertex v in time t (resp. prohibits agent a_i to move from vertex v_1 to vertex v_2 in time t), its purpose is to avoid vertex conflicts (resp. edge ”swapping” conflicts).

High Level

This level stores and searches a binary constraint tree (CT). Each node of the CT consists of:

- Set of constraints
- Total cost (*Makespan* or *Sum Of Costs* may be used)
- Solution

In each node, first, the constraints are loaded, and then the low-level search takes place. The low-level search returns the shortest path for each agent. When all the single-agent solutions are returned from the low-level search to the node, they are joined and validated. Validation is conducted through timestep iteration and location matching of all the agents. If there is no conflict, the joint solution is valid. A node is a goal node if the joint solution is valid. If there is a conflict, the joint solution is invalid, and the conflict needs to be resolved.

Conflicts are resolved using constraints. When a conflict is found in the joint solution, then it is divided into two constraints. For conflict $Cn = (a_i, a_j, v, t)$,meaning agents a_i and a_j collide in vertex v in time t, there will be two vertex constraints created: (a_i, v, t) and (a_j, v, t), similarly for an edge conflict. One constraint is added to the node's left child and the other one to the node's right child. Each node stores only one constraint (except for the root, which has no constraints stored). The whole set of each node's constraints can be extracted by traversing the tree from the node to the root.

Since we only add one constraint in each node, we only need to update one search for each node. This makes the solution-finding process more efficient.

Let us explain how the CT works on a simple example 1.4. There are two agents for whom we need to find the fastest paths to their goals. Black squares simulate obstacles. In CT 1.5 we can see that there is a collision $Cn = (a_1, a_2, 4, 1)$. As described before, constraints $(a_1, 4, 1)$ and $(a_2, 4, 1)$ are saved to children nodes, respectively. Low-level search is invoked for the left child (resp. right child), low-level searches for a new solution only for agent a_1 (resp. agent a_2). In the second level, both nodes found valid joint solutions. Therefore, they are both considered goal nodes. As we use *make of span* as the evaluation function, we see that the cost is the same in this example so either of them can be used as the final joint solution.

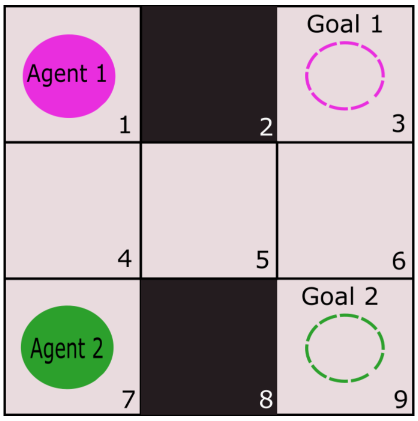

Figure 1.4: MAPF Problem example with two agents

Some conflicts may occur between more than two agents, $Cn = (a_1, .., a_k, v, t)$, where $k > 2$. We have two possible solutions to resolve this conflict:

1. Create k children with $k - 1$ constraints.
2. Solve only the conflict of the first two agents.

If we follow the first approach and create k children, we will no longer have a binary tree. Each of those k children will also have $k - 1$ constraints instead of one, which does not comply with saving only one constraint per node. The second approach will result in having to solve the rest of the conflicts in the deeper levels. If there is no duplication detection implemented, it will lead to searching for the same path with the same constraints twice, as shown in the picture 1.6. The first part of the picture shows method number one, where k children are created with $k - 1$ constraints. The second part of the picture shows the conflict's partial solution, where each node solves only the first two agents in the conflict. We can then see how it leads to a duplicate node.

We can observe the CBS high-level pseudocode in algorithm 1. The algorithm takes a MAPF instance as an input. In lines 1-3, the root of the

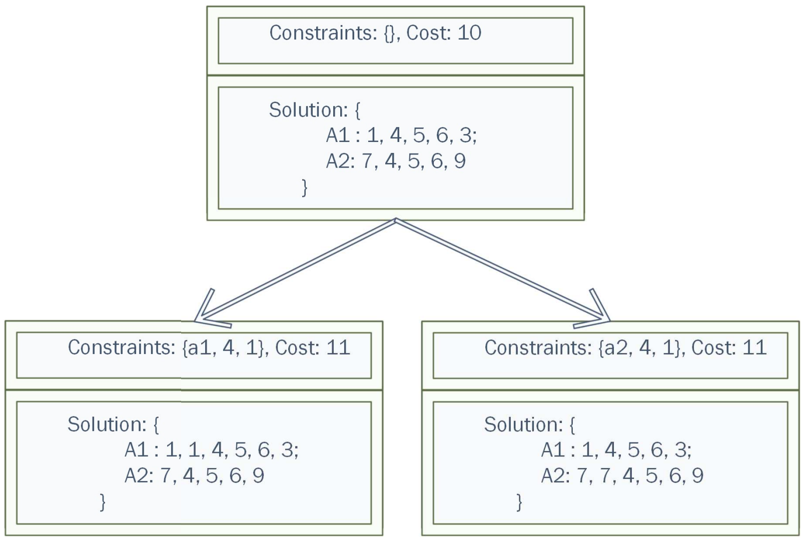

Figure 1.5: Constraint tree for problem 1.4

constraint tree is updated, a solution is found for every task, the cost is estimated, and the root is added to the open nodes' queue. The rest of the algorithm is a while cycle; the node with the lowest cost is processed first (line 6). The processed node is then validated (line 7); if the node's solution is valid, it is returned as a goal node (line 8). If the solution is not valid, the first conflict is found (line 10). A new node is created and added as a child of the current node for each agent in the conflict. To each child, one new constraint of agent a_i is added (line 13), the child's solution is taken from the current node (line 14), only agent a_i's path is recomputed taking the new constraint into account (line 15). Child's cost is computed next (line 16). If the computed cost is less than infinity, it means the solution is valid and therefore, it is added to the queue (line 18).

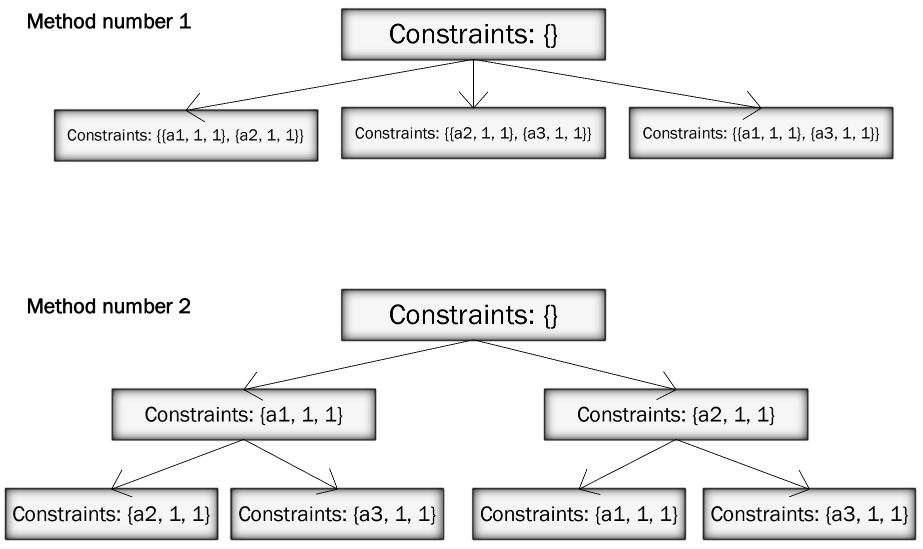

Figure 1.6: Possible conflict solutions for more than 2 agents

Algorithm 1 CBS High Level

```
1: $Root.constraints \leftarrow \{\}$    ▷ Root is the root of the constraint tree
2: $Root.solution \leftarrow low - levelsearchforalltheagents$
3: $Root.cost \leftarrow getCost(Root.solution)$
4: $Q\_Open \leftarrow insertRoot$    ▷ Queue containing open constraint tree nodes
5: while Q_Open is not empty do
6:     $tmp \leftarrow getLowest(Q\_Open)$    ▷ Returns node with the lowest cost
7:     validate $tmp.solution$
8:     if $tmp$ is valid then return $tmp.solution$    ▷ tmp is a goal node, no need to search for conflicts
9:     end if
10:    $Cn \leftarrow getFirstConflict(tmp.solution)$
11:    for each $a_i$ in $Cn$ do
12:        Create new node Child
13:        $Child.constraint \leftarrow tmp.consraint + (a_i, v, t)$
14:        $Child.solution \leftarrow tmp.solution$
15:        $Child.solution \leftarrow low - levelsearchforagenta_i$
16:        $Child.cost \leftarrow getCost(Child.solution)$
17:        if A.cost $< \infty$ then    ▷ Solution found
18:            $Q\_open \leftarrow Child$
19:        end if
20:    end for
21: end while
```

Low Level

The low level does the single-agent path searching part. This algorithm uses an A* search to find the shortest paths.The A* works in a two-dimensional space. In other words, it works with spatial and time dimensions. The low level recieves a set of constraints for each agent from the high level; these constraints are being checked during the search. If the search generates a state that does not comply with the constraints, it is deleted immediately without any expansion. The Manhattan distance from an agent's goal location is used as the basic heuristic for the search.

1.2 Lifelong MAPD

Lifelong Multi-agent pickup and delivery (MAPD) is an online version of the previously described MAPF. Multi-agent pickup and delivery (MAPD) is a specific version of MAPF that adds multiple goals to the agent paths (pickup and delivery locations).

The difference between the lifelong and offline version of this problem is that in the lifelong version the tasks are coming to the system throughout time, meaning we do not know all the tasks initially when we start solving the problem. In lifelong MAPD, agents are constantly being assigned new incoming tasks as opposed to offline MAPF, where, usually, an agent reaches its goal, and stays there untill the end of the program. This approach better simulates real-world applications.

1.2.1 MAPD Definition

In the previous section, we defined the MAPF problem; the formal definition for lifelong MAPD stays similar. The difference is that MAPD is adding a starting position that can differ from the agent's initial position. Instead of defining agents as tuples G = (G, s, g), we need to define a set of agents and tasks separately. Tasks are defined by their start (pickup) and goal (delivery) locations. Since tasks can come to the system at any time, we cannot assign them to the agents at the beginning of the program. A task can be assigned to an agent throughout the program's run; the agent then fulfils the task by going first to the start location and then to the goal location. When a task is assigned to an agent, it is taken away from the set of the tasks, and when the agent achieves the goal location, the task is finished.

In this paper, we will work only with well-formed (solvable) instances of the problems. Therefore we need to define rules to follow when creating an instance to make sure it can be solved. Conditions for a well-formed instance are as follows:

1. Agents can rest (”stay forever”) only in locations that are called endpoint, where they cannot block other agents.
2. There are task (pickup and delivery locations) and non-task endpoints (rest locations); the number of non-task endpoints must be at least the same as the number of agents.
3. For any two endpoints, there exists a way that does not traverse other endpoints.
4. The number of tasks is finite.

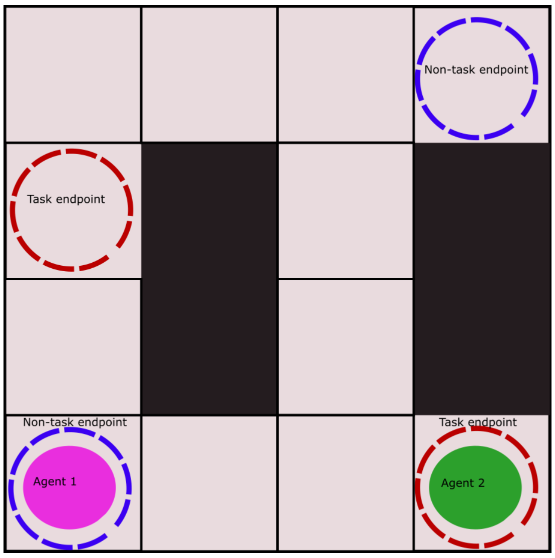

Figure 1.7: Well-formed instance

Examples of MAPD instances can be seen in schemas 1.7, 1.8 and 1.9. Schema 1.7 is an example of a well-formed instance. We can see there an instance with two agents in their initial positions. The number of non-task endpoints is the same as the number of agents, which complies with condition number 2. None of the non-task endpoints is blocking any way between any other of the endpoints.

The second schema 1.8 is not a well-formed instance. We can see that there is a non-task endpoint that is blocking a task endpoint. Since agents can stay in the non-task endpoints ”forever”, there is a possibility that a task could not be executed because of an agent resting in this endpoint and blocking the way to a task endpoint.

Third schema 1.9 is an example of a violation of the condition that there has to be at least the same number of non-task endpoints as the number of

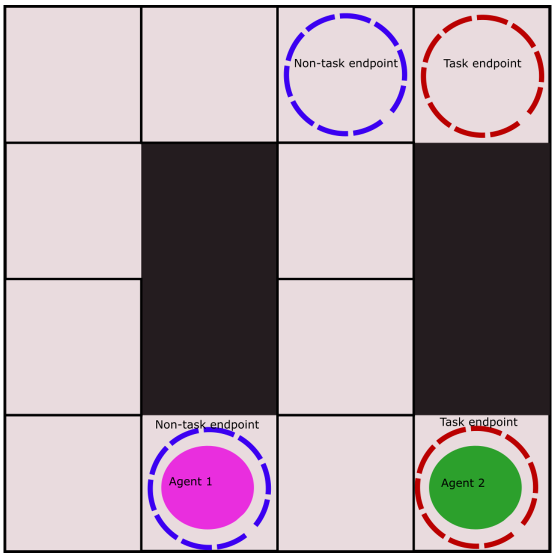

Figure 1.8: Instance violates condition number 3

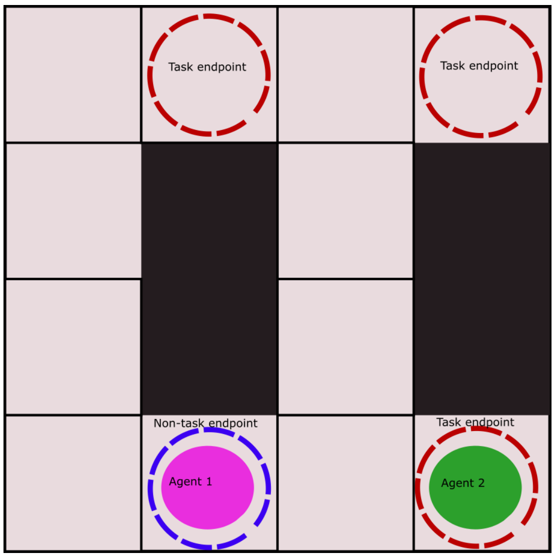

Figure 1.9: Instance violates condition number 2

agents. There are two agents in the schema, but only one non-task endpoint. Non-task endpoints are important to get agents ”out of the way” of the agents who are fulfiling their tasks and make sure the ”resting” agents are not blocking them. [23]

1.2.2 MAPD Solution Evaluation

There is a couple of different ways how to evaluate a MAPD solution. In the previous chapter, we mentioned using *makespan* and *Sum of costs* to evaluate the solution. We cannot use these functions here the same way since this is a lifelong problem and it may not be possible to get all the data from the beginning of the first task until the end of the last task. However, we can modify these functions to evaluate the solution in our MAPD problem.

The objective is to finish each task as fast as possible, which means assigning the task to an agent as fast as possible and find the fastest path to completion. The first evaluation option is to take the average number of timesteps to finish a task after it has been added to the system. This is a suitable evaluation method for the lifelong version since we cannot finish with all the tasks at once. To evaluate the simulations, we can still use the *Sum of costs* function described in the previous chapter.

1.2.3 Lifelong MAPD Algorithms

There are different approaches when it comes to lifelong versions of MAPF algorithms. Most of the MAPF algorithms can be modified to run ”lifelong”. The main types of modifications are:

1. Recomputing the whole solution after each timestep or when a new task comes to the system.
2. Finding path only for the new incoming tasks.
3. Computing solution only for a certain amount of timesteps.

Replanning everything

Any MAPF algorithm can be used in this approach. The idea behind this method is to run a MAPF algorithm every timestep. This leads to a decomposition of the problem to a sequence of MAPF problems. In each timestep, all agents are replanned. Even though it may be the easiest solution to implement, the computational time is very high. We can also see that a great part of the computed solution is useless since the agents will only perform one action before having to recompute their paths again.

Planning only new tasks

The second approach only searches the path for the new tasks upon entering the system. At the very beginning of the program, the algorithm searches solution for the existing tasks and then when a new task comes, it only searches the path for the new task using the existing solution as a base. This approach is faster than the first one, but it may not find a valid solution for the new task.

An example of an algorithm can be the Token passing (TP) algorithm [23]. This algorithm is similar to cooperative A*, where all the agents search the path one after another. The TP algorithm contains a token, which is a synchronized shared block of memory containing all the current paths of all agents, task set and agents' assignments. When a new task comes, it is planned respecting the current paths in the token. This algorithm is very similar to the COBRA algorithm [24], which is only constituted by one task endpoint instead of two (pickup and delivery).

Windowed planning

The main idea of windowed planning is to plan the paths only for a certain number of timesteps. The algorithm finds a solution to a MAPF problem for the given number of timesteps; after those timesteps, it searches for the paths again, including new tasks that came to the system during the previous solution's execution.

An example of such an algorithm is Windowed MAPF [25]. This algorithm allows assigning a sequence of goal locations to an agent. Collisions are solved only for a given number of timestep w. It defines $h = (w \geq h)$ as the number of steps executed for each solution. Firstly the algorithm updates the start position and counts the distance to the first goal position plus the distance between the following goal locations. The total distance for each agent needs to be at least h. When all agents have their tasks assigned, the planning for the next w steps begins. Windowed MAPF is using Multi-label A* algorithm [26] to find paths for single agents.

The advantage of this approach is that all agents are involved in the solution; throughput is increased. The generated plans are pliable and adapting to incoming tasks. Computational time decreases compared to the first approach.

There are, of course, more methods on how to implement an online version of a MAPF algorithm; these were the most common approaches. Each approach has its pros and cons, and selection depends on its usage.

1.3 Prioritization

This section will introduce a task priority in a MAPD, meaning tasks can have different priorities. Let us take an example of a warehouse; agents take tasks as they come to the system, but not all the tasks have the same value. For example, we can have an e-shop with an automated warehouse. This e-shop has regular members (buyers) and premium members. The advantage of being a premium member is to have their order delivered faster. The automated warehouse needs to react accordingly and finish the task (order) of the premium member first. Another example is a warehouse with food storage; some items may need to be shipped as fast as possible. In this day and age, where the supply chain has seen such a dramatic change, a warehouse cannot afford to stick to the original logistics plan if optimization of resources is in order.

1.3.1 Approaches to prioritization

Some algorithms are using some task prioritization, for example, Prioritized planning. But these algorithms are usually neither complete nor optimal. Therefore we need to focus on modifying other algorithms to achieve better results.

Most MAPF settings do not assume prioritized tasks. During our research, we encountered only one approach to prioritization that can be modified and applied to our problem. We will now briefly describe algorithms that have been proposed in research [27].

CBS-Pri

We already described the CBS algorithm in the previous section section 1.1.3. The main change of this variation is introducing a new cost function. Travel cost is weighted based on priority. This modification leads to CBS prioritizing shorter paths of priority tasks.

CA*-Pri

Cooperative A* for MAPF with priority first sorts tasks by their priorities. After sorting, it plans the tasks one by one using a reservation table (planned tasks write their paths to the reservation table so no other task can use the same location at the same time).

According to the aforementioned research [27], CBS-Pri needs more computational time than CA*-Pri. The problem of CA*-Pri compared to CBS-Pri is that the solution's optimality is not guaranteed.

Chapter 2

Analysis and design

In this chapter, we will focus on the implemented algorithm, its modifications and properties. The goal was to implement an algorithm that solves a lifelong MAPD problem that has tasks with different priorities. As mentioned before, tasks with different priorities are important for warehouse logistics and can increase the productivity and profit of a business.

2.1 Priority MAPD Definition

To implement the algorithm, we need to formulate the priority MAPD problem. We already defined the MAPD problem in subsection 1.2.1. Now we need to add prioritization to this definition.

Now we define priorities in a MAPD problem.

Definition 2.1.1 (Priority)**.** We define n as the number of tasks throughout the run of the program and array $p = (p_1, p_2, ..., p_n)$ as an array of priorities of tasks $t = (t_1, t_2, ..., t_n)$.

- $\forall i$, where $i \in p, i \in \mathbb{N}$

The higher the value of priority is, the higher priority the task has.

We can now establish a new definition for a task in a MAPD problem.

Definition 2.1.2 (Task with priority)**.** A task is a tuple defined as $t = (s, g, init\_t, p)$, where:

- s, where $s \in G$, is start (pick-up) location,
- g, where $g \in G$, is goal (delivery) location
- $init\_t$, where $init\_t \in \mathbb{N}$, is a timestep in which the task enters the system

- p, where $p \in \mathbb{N}$, is task's priority

The definition of lifelong priority MAPD is already mentioned in the definition of lifelong MAPD with tasks with priorities. The lifelong version means that we have all the tasks t that can come to the system in any timestep. In other words, the system does not know the tasks in advance. It can only see the tasks that have been already sent to the system. At the very beginning, there are only tasks with timestep zero; all the other tasks enter the system continuously.

2.2 Data Structure

We follow the MAPD definition from subsection 1.2.1. Only well-formed instances will be considered during the implementation and testing process.

We represent graph G as a four-connected grid. It has specified non-task endpoints (resting locations), pickup and delivery locations, and locations that cannot be visited (places where the items are stored). A G grid example can be seen in the picture 2.1. We can see non-endpoint locations highlighted in blue colour, pickup locations in yellow colour, and delivery locations in green colour. Locations that cannot be visited are black.

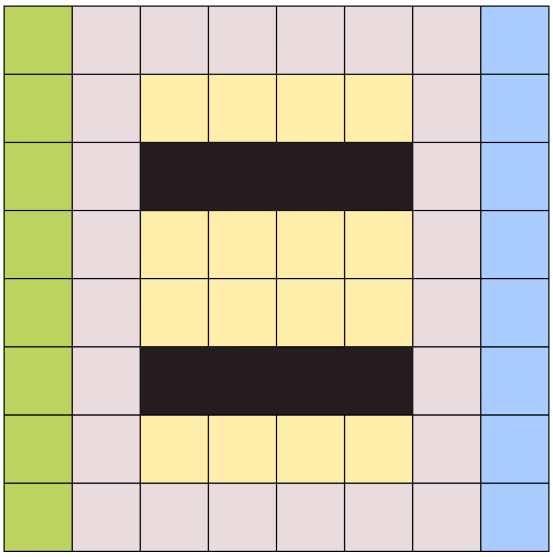
Figure 2.1: Example of a warehouse-style map represented as a 4-connected grid

We are going to use timesteps as a measuring time of one action; move or wait. One timestep measures time to move from one location to its adjacent location. Neighbouring locations are always the same distance away from each other. We do not consider waiting time in the pickup (resp. delivery) location; the agent just needs to pass through the location to fulfil its task.

Mobile robots are represented by agents. All the robots move at the same speed. We assume the environment to be static and deterministic.

The whole data structure is defined as follows:

- 4-connected grid G (map)
- non-task endpoints
- agents with their initial locations
- tasks including initial timesteps and priorities

2.3 Windowed Priority CBS

We implemented a new modification Windowed Priority CBS algorithm for automatic warehouse logistics. As the name suggests, the base solution is the CBS algorithm. The reasons we chose CBS are its optimality and completeness for well-formed instances, as well as its performance.

According to CBS experimental results in [22], it has proved to be faster than other optimal solvers. Although [22] shows that A* outperforms CBS in open spaces, the results in the paper indicate that CBS performs better than A* in bottleneck spaces. Since we are solving a problem for a warehouse, we will need better performance in bottleneck spaces. There is usually not much open space in a warehouse, as corridors between items tend to be small, and we want our algorithm to perform well for a large number of robots that need to be maneuvering around each other. The study also shows results on a 4-connected grid graph that we are using. CBS has performed the best for a higher number of agents in the given graph compared to A* and ICTS. EPEA* (Partial-Expansion A* with Selective Node Generation) [28] has outperformed CBS in the number of generated nodes, but the CBS run time was still lower than the EPEA* run time.

2.3.1 Windowed Approach

As described in section 1.2, there are several ways to implement a lifelong version of a MAPF algorithm.

Replanning everything every single timestep is inefficient. Using this approach could lead to the algorithm being unusable in the real world.

Planning only new tasks when they come is a nice idea, but it is less useful for prioritising tasks. If a new task of a higher priority than the already planned tasks (planned only, not executed yet) comes, the algorithm would have to replan everything again to satisfy the newly incoming task first. This could lead to potentially having to replan everything almost every single timestep.

We decided to use the windowed version because of its effectiveness for our purposes. Let us demonstrate how windowed CBS works: We have a specified number of timesteps w that we need to plan. At the beginning of the planning, we assign tasks to the agents so that the distance between their current location and their tasks goals is at least w, using Manhattan distance to calculate the distances between locations. When tasks are assigned, we run CBS on every single agent.

We had to modify CBS to accept multiple goals. Low-level A* search takes a sorted array of goals that have to be visited in this order and searches the solution accordingly. High-level CBS was modified to satisfy the windowed approach by checking for collisions only for w timesteps. This leads to solutions with timesteps higher than w possibly having collisions within each other. These collisions will be resolved in the next *window*, which speeds up the program.

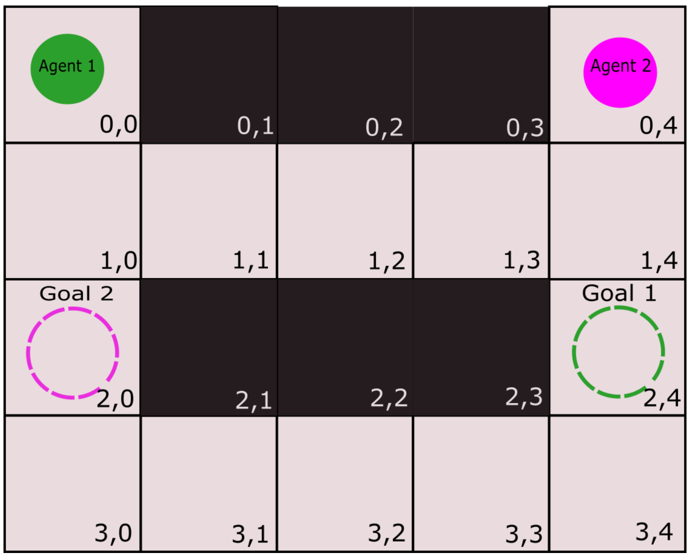

Figure 2.2: Example of a warehouse-style map to demonstrate windowed CBS

Not resolving the possible collisions can cause the agents not to find the optimal solution since there can be a conflict later. The potential conflict may have to be resolved with an agent having to go back and finding a different path. This problem can occur mainly when the number w is small. We can see an example of a non-optimal solution in picture 2.2. We have two agents and their goal positions. No other tasks are coming to the system until these tasks are finished. Let us first solve the problem as a whole. An optimal solution is:

- a_1 : (0,0) $\rightarrow$ (1,0) $\rightarrow$ (1,1) $\rightarrow$ (1,2) $\rightarrow$ (1,3) $\rightarrow$ (1,4) $\rightarrow$ (2,4)
- a_2: (0,4) $\rightarrow$ (1,4) $\rightarrow$ (2,4) $\rightarrow$ (3,4) $\rightarrow$ (3,3) $\rightarrow$ (3,2) $\rightarrow$ (3,1) $\rightarrow$ (3,0) $\rightarrow$ (2,0)

Solution evaluation:

- $makespan = 8$
- $sumOfCosts = 6 + 8 = 14$.

Now we will simulate a windowed approach with $w = 2$, meaning we will replan the solution every two steps and only check for constraints in the first two steps of each partial solution.

- t = 0
  - a_1: (0,0) $\rightarrow$ (1,0) $\rightarrow$ (1,1)
  - a_2: (0,4) $\rightarrow$ (1,4) $\rightarrow$ (1,3)
- t = 2
  - a_1: (1,1) $\rightarrow$ (1,2) $\rightarrow$ (1,3)
  - a_2: (1,3) $\rightarrow$ (1,4) $\rightarrow$ (2,4)
- t = 4
  - a_1: (1,3) $\rightarrow$ (1,4) $\rightarrow$ (2,4)
  - a_2: (2,4) $\rightarrow$ (3,4) $\rightarrow$ (3,3)
- t = 6
  - a_2: (3,3) $\rightarrow$ (3,2) $\rightarrow$ (3,1)
- t = 8
  - a_2: (3,1) $\rightarrow$ (3,0) $\rightarrow$ (2,0)

Solution evaluation:

- $makespan = 10$

- $sumOfCosts = 6 + 10 = 16$

We can see that the second solution is not optimal. The agents are only planned for the two successive timesteps, and agent a_2 cannot see the future conflict in the first run. This leads to the agent having to go back and having to find another path.

Let us set $w = 3$ and simulate the program again:

- t = 0
  - a_1: (0,0) → (1,0) → (1,1) → (1,2)
  - a_2: (0,4) → (1,4) → (1,3) → (1,3)

Even from the first run, it is evident that this solution is even worse than the solution found with $w = 2$.

We will set $w = 4$:

- t = 0
  - a_1: (0,0) → (1,0) → (1,1) → (1,2) → (1,3)
  - a_2: (0,4) → (1,4) → (2,4) → (3,4) → (3,3)
- t = 4
  - a_1: (1,3) → (1,4) → (2,4)
  - a_2: (3,3) → (3,2) → (3,1) → (3,0)

This is an optimal solution. Even though the windowed CBS will not always find optimal solutions, we decided to use this approach based on its effectiveness.

2.3.2 Analysis of the Priorities

In this section, we will focus on different approaches to prioritization. We used these approaches to implement new modifications of the Windowed CBS algorithm for warehouses. As it was mentioned before, each task that arrives to the system has its priority. Tasks with higher priorities need to be processed faster than tasks with lower priorities. We will now discuss possible improvements of Windowed CBS to achieve this goal.

CBS Prioritization

The first improvement is a modification of CBS low-level. We previously defined one timestep as one action (move or wait). Without priorities, we would use Manhattan distance as a heuristic for our A* search in the CBS low-level. Every action of all the agents would cost the same. To achieve better results for high priority tasks, each task's heuristics is multiplied by their priority. This leads to an increased cost of actions for higher priority tasks.

We will demonstrate this prioritization approach in example 2.3. There are two agents who need to get to their goal locations. The priority of the task of agent a_1 is 2, and the priority of the task of agent a_2 is 1. The fastest single-agent paths:

- a_1: (2, 0) → (1,0) → (1,1) → (1,2) → (1,3) → (1,4) → (2,4)
- a_2: (2, 4) → (1,4) → (1,3) → (1,2) → (1,1) → (1,0) → (2,0)

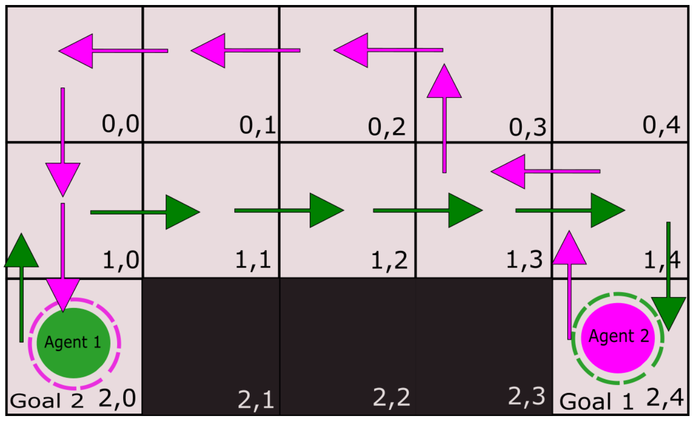

Figure 2.3: Example of CBS prioritization

When we put these two single-agent solutions together, it will result in a collision in vertex $(1, 2)$. To avoid the collision, one of the agents needs to choose a longer path to its goal. If the tasks' priorities are the same, either agent can choose the longer route, and the joint result will be the same. Using the priorities, a_2 is forced to move out of the way of a_1. We can see the updated routes in picture 2.3. The routes are:

- a_1: (2, 0) → (1,0) → (1,1) → (1,2) → (1,3) → (1,4) → (2,4)
- a_2: (2, 4) → (1,4) → (1,3) → (0,3) → (0,2) → (0,1) → (0,0) → (1,0) → (2,0)

Solution evaluation:

- Path cost of $a_1 : 6 * priority = 6 * 2 = 12$
- Path cost of $a_2 : 8 * priority = 8 * 1 = 8$
- $SumOfCosts = 12 + 8 = 20$

If we decided to change the way of a_1 the solution evaluation would look like:

- Path cost of a_1: 8 * 2 = 16
- Path cost of a_2: 6 * 1 = 6
- $SumOfCosts = 16 + 6 = 22$

The *Sum of costs* is higher when changing the way of the agent a_1 due to its higher priority. However, higher priority does not always mean the task will be performed before another one with lower priority.

Let us look at our second example 2.4. We have two agents and their goals. Priorities of the agents' tasks are as follows $p_1 = 2$ and $p_2 = 1$. The resulting path can be seen in picture 2.5. The picture shows that even though a_1 has higher priority than a_2, it will still have to change its path to avoid collisions with a_1.

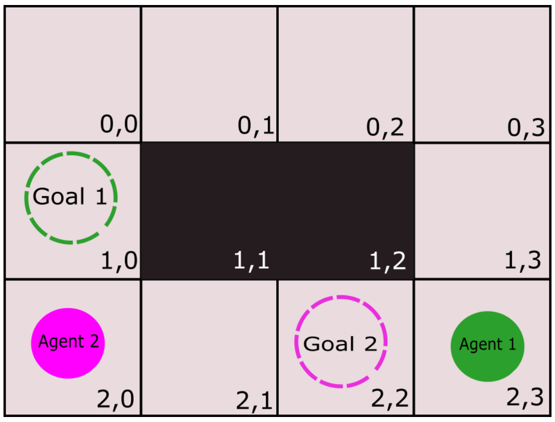

Figure 2.4: Example 2 of CBS prioritization

Solution evaluation:

- Path cost of $a_1 : 6 * priority = 6 * 2 = 12$
- Path cost of $a_2 : 2 * priority = 2 * 1 = 2$

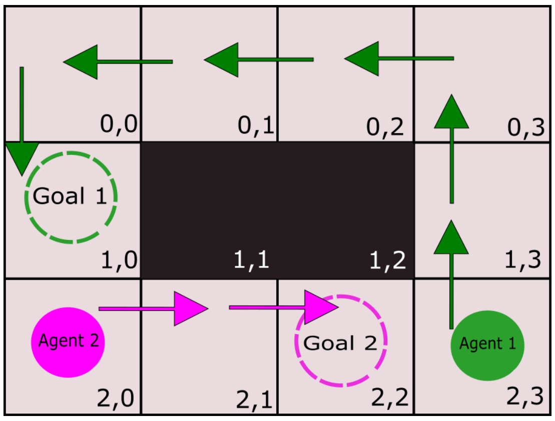

Figure 2.5: Example path of CBS prioritization, $p_1 = 2$, $p_2 = 1$

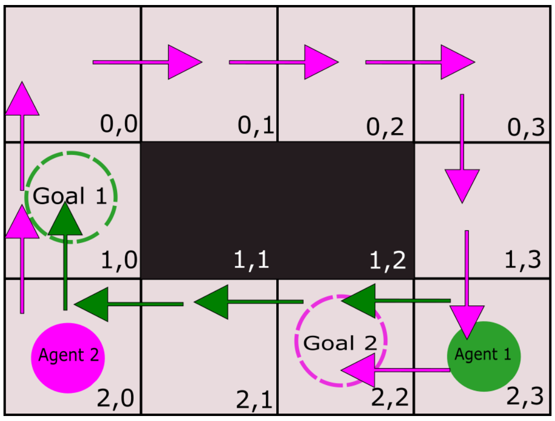

Figure 2.6: Example path of CBS prioritization, $p_1 = 4$, $p_2 = 1$

- $SumOfCosts = 12 + 2 = 14$

Solution evaluation if a_2 changed its way to avoid a_1 (path can be seen in picture 2.6) :

- Path cost of a_1 : $4 * priority = 4 * 2 = 8$
- Path cost of a_2 : $8 * priority = 8 * 1 = 8$
- $SumOfCosts = 8 + 8 = 16$

We can see that even though the task of the agent a_1 would finish faster, the *sum of costs*would be higher than in the first solution. The joint solution will change if we change the tasks' priorities. We set $p_1 = 4$ and $p_2 = 1$. The result can be seen in picture 2.6. Here the solution evaluation is:

- Path cost of $a_1 : 4 * priority = 4 * 4 = 16$
- Path cost of $a_2 : 8 * priority = 8 * 1 = 8$
- $SumOfCosts = 16 + 8 = 24$

If we use the solution from picture 2.5 :

- Path cost of $a_1 : 6 * priority = 6 * 4 = 24$
- Path cost of $a_2 : 8 * priority = 2 * 1 = 2$
- $SumOfCosts = 24 + 2 = 26$

When using these priorities, we need to change the way of the agent a_2. These examples show how CBS works using weighted priorities for the path cost.

Priority Queue for Assigning Tasks

We need to implement some strategy for assigning tasks to agents. We will use a priority queue for these purposes. This queue will sort tasks based on their priorities, meaning high priority tasks will be assigned before low priority tasks. While this is a simple solution, it can lead to some tasks never being executed. Since we are implementing a prototype of a lifelong MAPD; there can be high priority tasks entering the system so the low priority ones are skipped and never assigned. To avoid this problem, we need to set a parameter s equal to a certain number of timesteps. When comparing tasks, we will add the number of timesteps that the task has been already waiting in the system divided by parameter s to the task's priority.

As an example, we will compare a couple of tasks, where t_init is the time they entered the system:

- $t_1 : t\_init = 0, p = 1$
- $t_2 : t\_init = 25, p = 2$
- $t_3 : t\_init = 35, p = 4$
- $t_4 : t\_init = 40, p = 2$

Now we will to define parameter $s = 20$ and current timestep $t_c = 42$. We can compute the waiting by substracting the time a task entered the system from the current time: $(t_c - t\_init)$. We will compute values for the priority queue accordingly: $\lfloor (t_c - t\_init)/s \rfloor + p$.

Computed values of tasks for the priority queue are as follows:

- $t_1 : \lfloor(42 - 0)/20\rfloor + 1 = 3$
- $t_2 : \lfloor(42 - 25)/20\rfloor + 2 = 3$
- $t_3 : \lfloor(42 - 35)/20\rfloor + 4 = 4$
- $t_4 : \lfloor(42 - 40)/20\rfloor + 2 = 2$

We can see that t_1 and t_2 have the same value. Here we can decide if we prioritize the task that has entered the system earlier or the task with higher priority. In this example, we will prioritize the task with higher priority. The order of the tasks will then be: t_3, t_2, t_1 and t_4. We can, of course, change the parameter s depending on how important it is to fulfill low priority tasks in a reasonable time.

Reassigning Tasks

At the beginning of each *window*, tasks are assigned to the agents. In case of not managing to fulfill all the tasks in one run, agents keep the rest of the assigned tasks for the next *window*. An improvement is implemented for reassigning previously assigned tasks. After each run, assigned tasks where agents haven't reached the pickup location are put back to our priority queue. New tasks are then assigned based on their position in the priority queue.

A problem example is ilustrated in picture 2.7. We have two agents and three tasks. The timesteps in which each task enters the system are as follows: $t\_init_1 = 0$, $t\_init_2 = 0$ and $t\_init_3 = 2$. The task priorities are: $p_1 = 1$, $p_2 = 1$ and $p_3 = 5$. We set $w = 3$. First, let's look at the solution without reassigning the tasks after each run.

An example solution can be seen in picture 2.8.

Joint solution is:

- t = 0
  - a_1: (1,4) → (1,3) → (1,2) → (1,1)
  - a_2: (2,4) → (3,4) → (3,3) → (3,2)
- t = 3
  - a_1: (1,1) → (1,0) → (2,0) → (3,0)
  - a_2: (3,2) → (3,1) → (2,1) → (1,1)
- t = 6
  - a_1: (3,0) → (3,1) → (3,2) → (3,3)
  - a_2: (1,1) → (0,1) → (0,0)

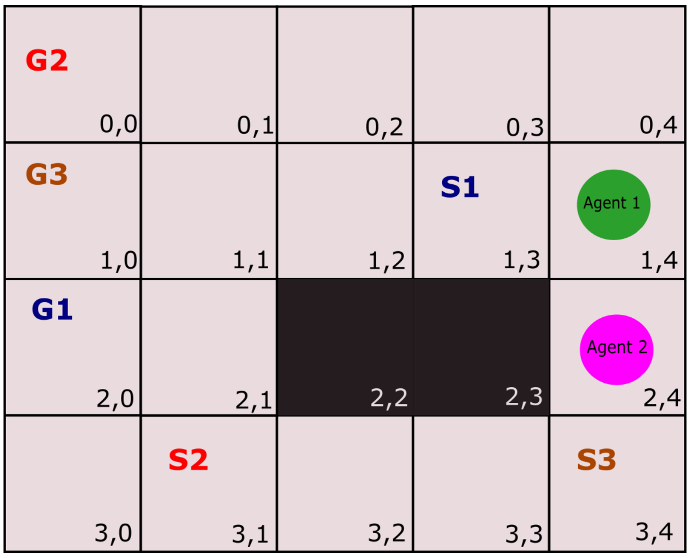

Figure 2.7: Example map for reassigning tasks

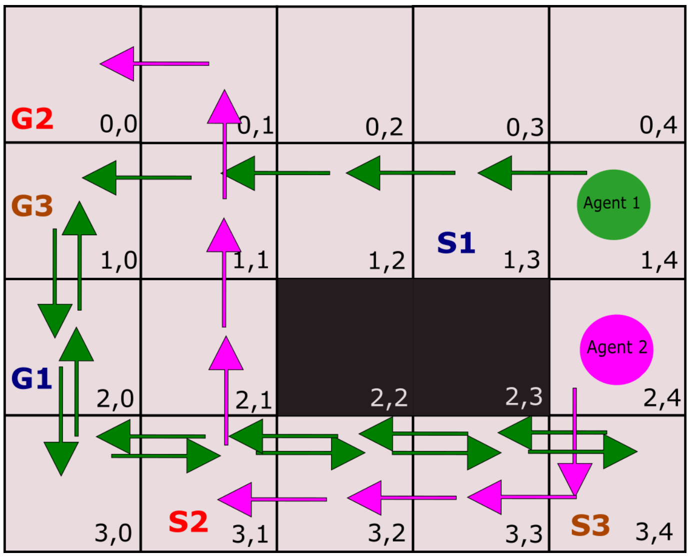

Figure 2.8: Reassging tasks example - path without reassigning tasks

- t = 9
  - a_1: (3,3) → (3,4) → (3,3) → (3,2)
- t = 12
  - a_1: (3,2) → (3,1) → (3,0) → (2,0)

- t = 15
  - a_2: (2,0) $\rightarrow$ (1,0)

We can see that when first run finishes and $t = 3$, a_1 is in location (1,1) and a_2 is in location (3,1). Since the heuristic of a_1 to finish its first task is 2, while the heuristic of a_2 is bigger than 3, then the new task will be assigned to agent a_1.

Solution evaluation:

- Path Cost $t_1 = (waitingTime + runTime) * priority = (1 + 4) * 1 = 5$
- Path Cost $t_2 = (waitingTime + runTime) * priority = (4 + 4) * 1 = 8$
- Path Cost $t_3 = (waitingTime + runTime) * priority = (8 + 6) * 5 = 70$
- $SumOfCosts = 5 + 8 + 70 = 83$

Now we look at the solution with reassigning tasks. We can see in picture 2.9 that in $t = 3$ task 2 is reassigned to a_1 and task 3 is assigned to a_2.

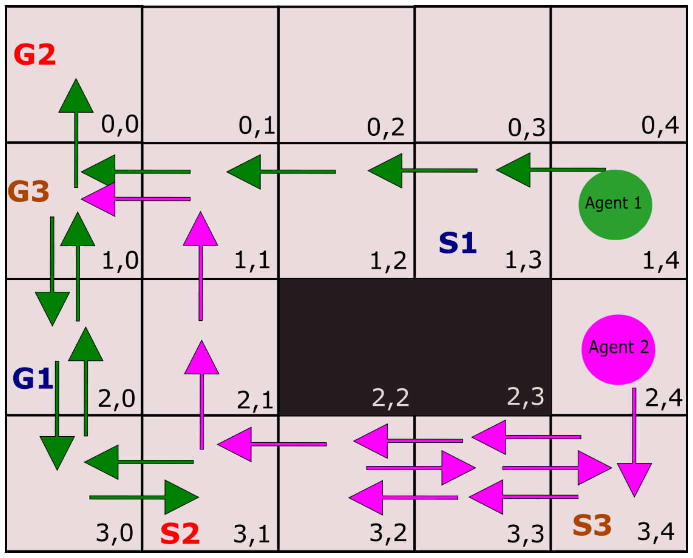

Figure 2.9: Reassging tasks example - path withreassigning tasks

Evaluation of this solution is:

- Path Cost $t_1 = (waitingTime + runTime) * priority = (1 + 4) * 1 = 5$
- Path Cost $t_2 = (waitingTime + runTime) * priority = (7 + 4) * 1 = 11$
- Path Cost $t_3 = (waitingTime + runTime) * priority = (2 + 6) * 5 = 40$

- $SumOfCosts = 5 + 11 + 40 = 56$

If we compare the $SumOfCosts$ of these two solutions, the second one performs better. Even if the priority of t_3 was 1, the solution with task reassigning bears better results.

Solution replanning

As we mentioned before, the solution is being replanned every w steps. We can get into a situation where a high priority task enters the system and needs to wait a long time before it can be planned. To react to this situation, we can replan the solution earlier if a task with a high priority enters a system.

We will demonstrate this approach in example 2.10. We have two agents and four tasks, we set $w = 10$. Properties of the tasks are as follows:

- Task 1 : $t\_init = 0, p = 1$
- Task 2: $t\_init = 0, p = 1$
- Task 3: $t\_init = 0, p = 1$
- Task 4: $t\_init = 5, p = 10$

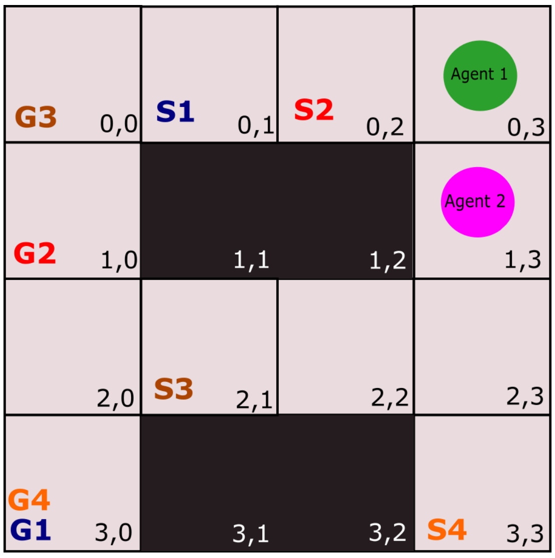

Figure 2.10: Replan tasks example map

We will set $p_r = 10$ to be the lowest priority to trigger immediate replanning.

Solution with replanning is then:

- t = 0
  - a_1: (0,3) $\rightarrow$ (0,2) $\rightarrow$ (0,1) $\rightarrow$ (0,0) $\rightarrow$ (1,0) $\rightarrow$ (2,0)
  - a_2: (1,3) $\rightarrow$ (0,3) $\rightarrow$ (0,2) $\rightarrow$ (0,1) $\rightarrow$ (0,0) $\rightarrow$ (1,0)
- t = 5
  - a_1: (2,0) $\rightarrow$ (3,0) $\rightarrow$ (2,0) $\rightarrow$ (2,1) $\rightarrow$ (2,2) $\rightarrow$ (2,3) $\rightarrow$ (3,3) $\rightarrow$ (2,3) $\rightarrow$ (2,2) $\rightarrow$ (2,1) $\rightarrow$ (2,0)
  - a_2: (1,0) $\rightarrow$ (1,0) $\rightarrow$ (1,0) $\rightarrow$ (2,0) $\rightarrow$ (2,1) $\rightarrow$ (2,0) $\rightarrow$ (1,0) $\rightarrow$ (0,0)
- t = 15
  - a_1: (2,0) $\rightarrow$ (3,0)

First, the tasks are assigned as follows (based on the distances from agents locations):

- a_1 : task 1
- a_2: task 2 and 3

We can see that it is replanned in timestep 5 when the task enters the system. When replanning, task 4 is assigned to a_1 since this agent is closer to the start location than agent a_2. The agent a_1 first needs to go to the delivery location of its first task, then it continues to fulfill task 4.

Solution evaluation:

- Path cost $t_1 = (2 + 4) * 1 = 6$
- Path cost $t_2 = (2 + 3) * 1 = 5$
- Path cost $t_3 = (9 + 3) * 1 = 12$
- Path cost $t_4 = (6 + 5) * 10 = 110$
- $SumOfCosts = 133$

If we did not replan in timestep 5 and waited until timestep 10, the solution evaluation would be:

- Path cost $t_1 = (2 + 4) * 1 = 6$
- Path cost $t_2 = (2 + 3) * 1 = 5$
- Path cost $t_3 = (7 + 3) * 1 = 10$
- Path cost $t_4 = (10 + 5) * 10 = 150$
- $SumOfCosts = 171$

The path of the first two tasks stays the same; the waiting time of the third task will decrease since task 4 is not assigned to agent a_1 and agent a_2 does not have to wait for a_1 to pass.

This was an example of how replanning can improve the results. Of course, it is not always so useful, and it can lead to longer computational times. The problem here is to set an adequate priority number when replanning takes place. Another improvement could be waiting until a certain amount of high priority tasks are present in the queue before replanning.

2.3.3 Idle Approaches

There are multiple approaches to what the agent should do when there are no more tasks for them to process. In offline MAPF agents either ”disappear” or stay in their positions until the end of the program. In online MAPF, they cannot ”disappear” since we need to use them in the future. They can stay in their current location until a new task enters the system, or they can move to their non-task endpoint location. We decided to move agents to their non-task endpoints so they would not occupy a delivery location in case some other agent needed to go there.

2.3.4 Algorithm Properties

In this section, we will discuss the completeness and optimality of the implemented solution. We implemented more versions of the algorithm, but these properties are the same for all the implemented versions.

Optimality

First, we need to define the optimality of a lifelong MAPD with prioritization to discuss the optimality of the implemented solution. First, we need to define the solution cost used to determine an optimal solution.

We will use weighted cost as we used in the prioritization of CBS. The cost is then $SumOfCosts = sum(pathcost * priority)$. Now we need to adapt this parameter to a lifelong MAPD. We will count path cost as the difference between the timestep a task entered the system and the timestep the task was finished.

We know that CBS returns an optimal solution [22]. Using the cost function $SumOfCosts = sum(pathcost * priority)$ in prioritization CBS will result in CBS returning optimal solutions even after adding priorities.

Now we will look at the lifelong approach.

Proposition 2.3.1. Windowed Priority CBS is optimal.

Proof. Even though our priority CBS returns optimal solution, the task assignment cannot determine how to assign the tasks to produce an optimal solution consistently. We will prove this using an example in picture 2.11. We have one agent, and two tasks, the properties of the tasks are:

- Task 1: $t = 0, p = 1$
- Task 2: $t = 2, p = 2$

We set the parameter $w = 2$.

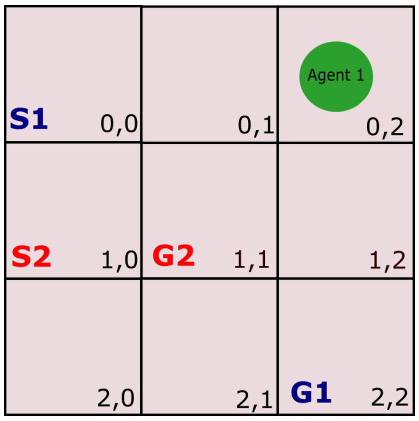

Figure 2.11: Optimality example map

The optimal solution can be seen in the picture 2.12. The shortest path for task 2 is to go directly from the agent's initial location to the task's start location. It will arrive there in time step 3, just one timestep after task 2 enters the system. After finishing task 2, the agent continues with task 1.

Evaluation of the optimal solution:

- Path cost $t_1 = 10$
- Path cost $t_2 = 2$
- $SumOfCosts = 10 * 1 + 2 * 2 = 14$

Now we will demonstrate the solution of the implemented algorithm 2.13.

It assigns task 1 to the agent in timestep 0. The agent performs two actions:

- (0,2) $\rightarrow$ (0,1) $\rightarrow$ (0,0)

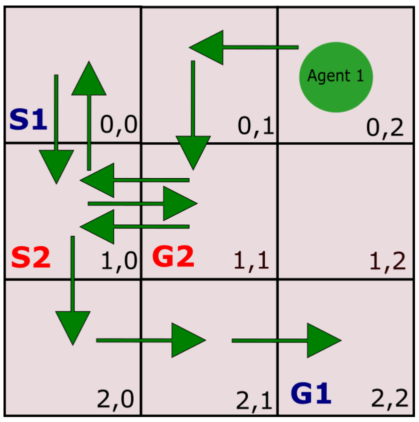

Figure 2.12: Optimal path

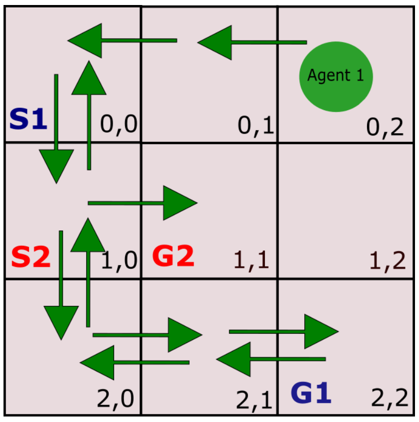

Figure 2.13: Non-optimal path

Now the rest of the solution is replanned. Task 2 has entered the system, but the agent has already reached the pickup location of task 1. It has to finish task 1 first and then continue to task 2. The following path is:

- (0,0) → (1,0) → (2,0)
- (2,0) → (2,1) → (2,2)
- (2,2) → (2,1) → (2,0)
- (2,0) → (1,0) → (1,1)

Solution evaluation:

- Path cost $t_1 = 6$
- Path cost $t_2 = 8$

- $SumOfCosts = 6 * 1 + 8 * 2 = 24$

We can see that the implemented algorithm will not return an optimal solution. In a lifelong version of the MAPD problem, we cannot ensure the optimality of the algorithm because we do not know what tasks enter and when they enter the system to plan the path accordingly.

□

Completeness

We will now discuss the completeness of the Windowed Priority CBS. The completeness of the CBS algorithm for well-formed problems is proved in [22]. Now we need to focus on the lifelong version of the algorithm. Tasks are entering the system at any moment of the program's run; for the algorithm to be complete, we need to process them all eventually. This can be a problem when the system is overwhelmed by tasks all the time and there are not enough agents to fulfill these tasks. It can come to the point when some of the tasks stay in the system "forever".

We will now prove that the algorithm is complete if all the tasks are eventually assigned.

Proposition 2.3.2. Windowed Priority CBS is complete if all the tasks are eventually assigned and the number of tasks is finite.

Proof. CBS is complete for well-formed instances [22]. We assume all the tasks will be assigned to agents. Windowed Priority CBS will run CBS for all of the tasks. We also suppose the number of tasks is finite, so the number of runs of CBS will be finite.

From the definition, CBS is complete; if we run it a finite amount of times, the solution will always be found, and therefore, Windowed Priority CBS is complete. □

Chapter 3

Experimental Evaluation

This chapter discusses the testing results of our algorithm. We tested four versions of the algorithm on four different maps. On each map, we tested five instances, with 30, 40, 60, 75 and 100 tasks, where each of these instances was tested for 2, 4 and 8 agents. The priorities of the tasks were set on a scale from 1 to 10.

We tested Windowed CBS with no priority; then we ran the same experiments with the following modifications:

- **No priority** - does not consider task priorities
- **CBS** - uses weighted cost based on the task priority and assigns tasks using a priority queue, a modification of the no priority version
- **Reassign tasks** - can reassign already assigned task if the start location has not been reached in the last run, a modification of the CBS version
- **Replan** - a modification of the version with reassigning tasks, replans the path immediately if a task with priority 9 or 10 enters the system

All of the experiments were run on Dell Latitude 5490 with processor Intel(R) Core(TM) i5-8350U and RAM 16GB.

Before performing the actual testing, we ran experiments to set the parameter w and to set parameters in the priority queue for assigning tasks. These tests were performed on the first map, and the results were used for the rest of the tests.

The first map is a warehouse-style corridor map 3.1. On the right side (pink locations), there are non-task endpoints; on the left side (blue areas), there are delivery locations. The black locations indicate obstacles, and the green locations are the pick-up locations.

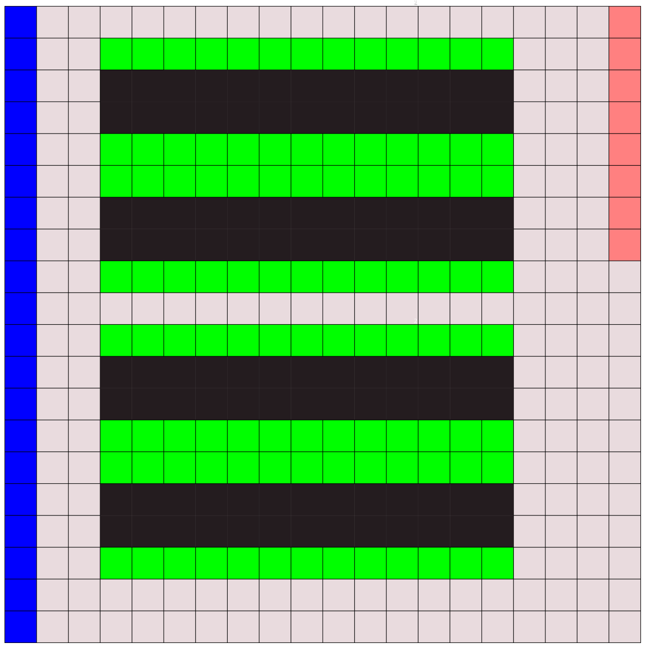

Figure 3.1: Experimental Map 1 - Warehouse-style corridors

3.1 Setting Parameter w

Before we test the modifications, we need to specify the w parameter that we will use for the experiment. We used our warehouse-style corridors map to test how to set parameter w so the program would run relatively fast and the program would return a solution with good evaluation. In this test, we did not use priorities; we ran a simple Windowed CBS. The tasks were assigned to the agents based on the timestep in which they entered the system. We measured the total number of the timesteps for which the algorithm ran. The results can be seen in the graph 3.2. Results for each instance highly depend on the order in which the tasks enter the system. Because of this, we ran the test on five different instances for three different number of agents, and summed up the total number of timesteps for all the instances with a specific amount of agents. As we can see, the results of each instance highly depend on the order of the tasks that are entering the system. For that reason, we ran the test on five different instances for each number of agents. We can see that, on average, $w = 10$ performed the best for all tested numbers of agents.

The second parameter we measured was time. We can see the results in the graph 3.3. For four agents, the summed up results for the five instances was the lowest for $w = 10$. For eight agents, the program ran the fastest for $w = 6$. Finally,we created a special graph for two agents 3.4, finding that the fastest setting for two agents was $w = 16$.

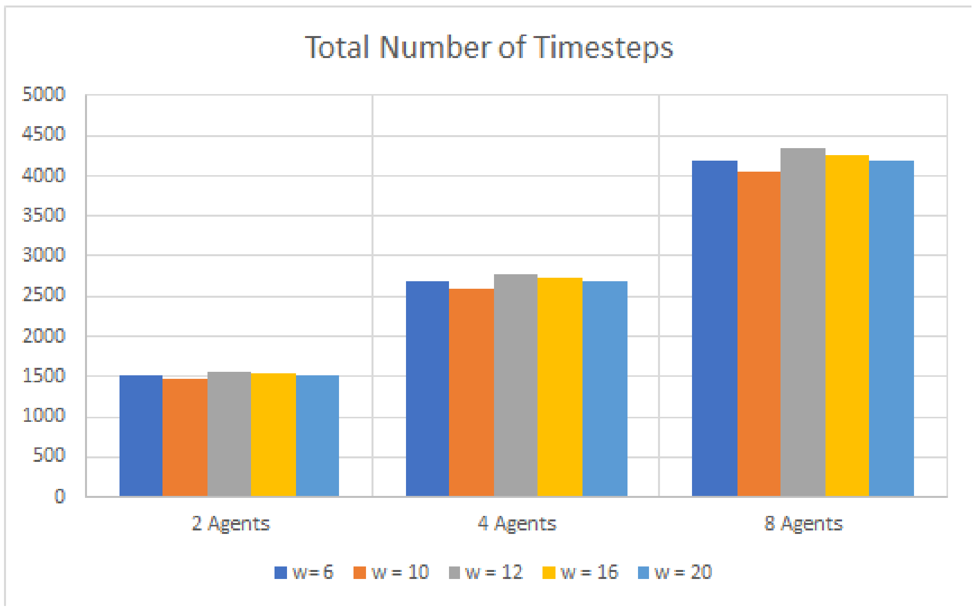

Figure 3.2: Total number of timesteps for different w settings

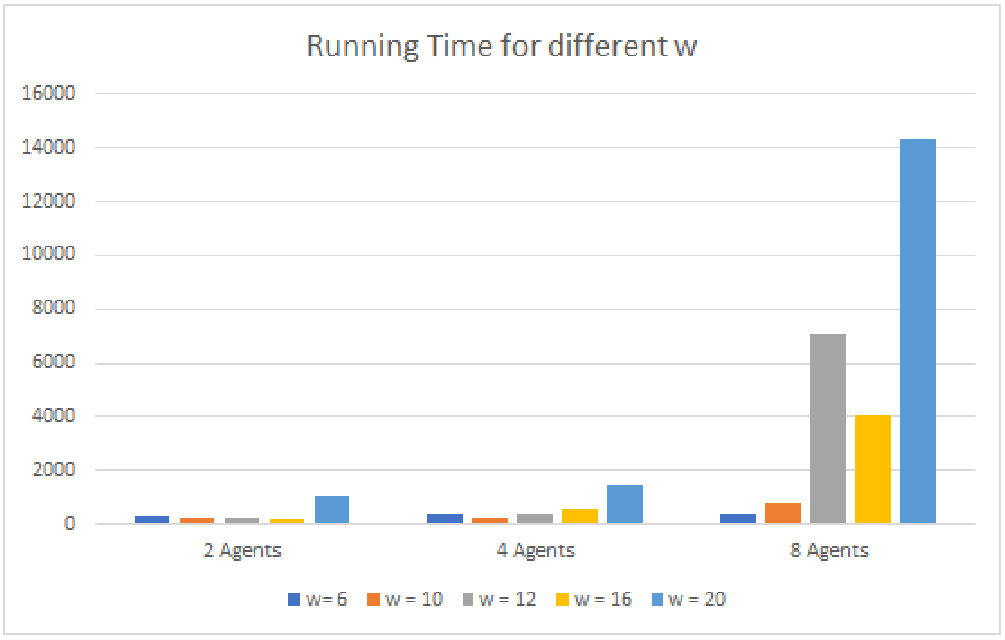

Figure 3.3: Running time of the program for different w settings

Based on these results, we decided to set the parameter $w = 16$ for instances with two agents. As previously stated, program ran the fastest, and

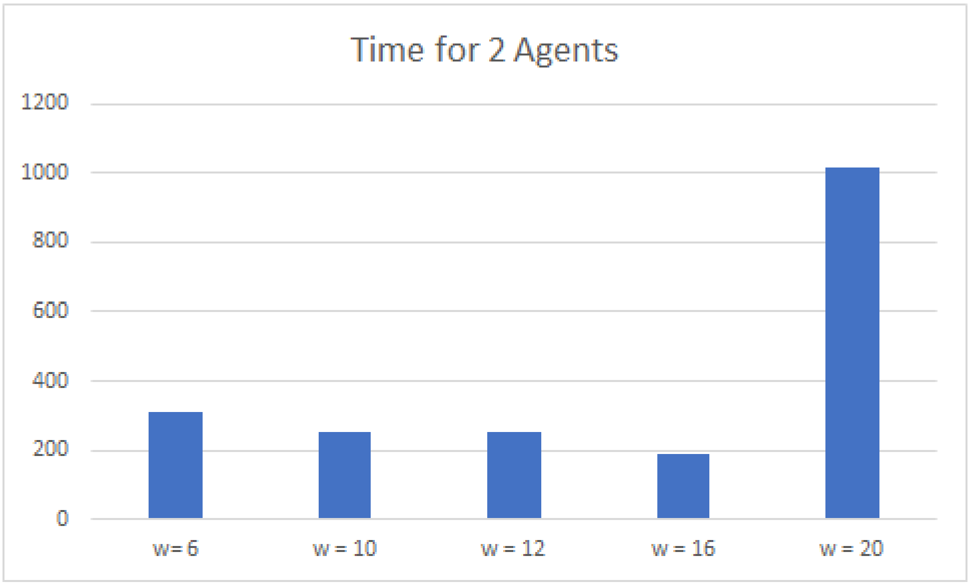

Figure 3.4: Running time of the program for different w settings

the total number of timesteps was just slightly higher than for other w settings. For instances with four and eight agents, we decided to to set the parameter $w = 10$. This setting was the best setting for instances with four agents. It was also the best in the total number of timesteps and second in the running time for eight agents. As said in the previous paragraph, we will use this setting for the rest of the experiments.

3.2 Priority Queue s Setting

As we described in the Analysis and Design chapter, the priority queue sorts tasks by their priorities. We need to define parameter s (parameter previously defined in section section 2.3.2), so the tasks with low priorities are not in the system until the end of the experiment.

We ran our experiments on the warehouse-style corridors map 3.1 on five instances with four agents. We focused on measuring waiting time for each priority to see how long it takes for tasks to be assigned. We define waiting time as a number of timesteps counted since the task entered the system until the pickup point of the task was reached. We can see the results in graph 3.5. The number of timesteps spent by waiting are summed up for the five instances.

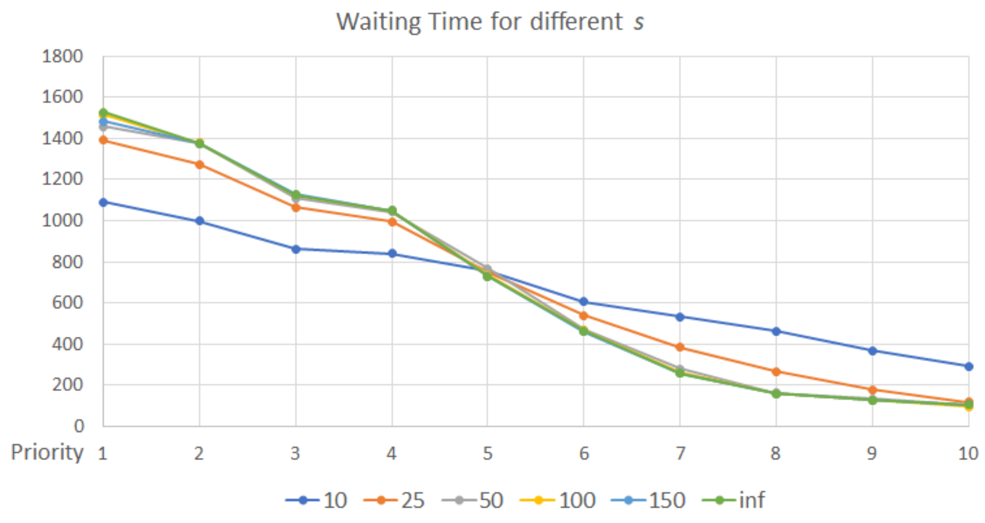

Figure 3.5: Waiting time

We can see that the lower we set parameter s, the better it performs for lower priorities. If we set parameter $s = 10$, the low priorities perform way better than for the rest of s settings, but the high priorities perform worse than the rest of the settings. We decided to set parameter s to be equal to 25 because it performs better for low priorities and performs well for high priorities, such as 9 or 10.

3.3 Warehouse-style Corridors Map

Now that we have set the parameters, we need to run the experiment, and we can start with testing the different modifications of the algorithm. We will begin with warehouse-style corridors map 3.1. This map is a typical warehouse-style map, given that it has a couple of areas where the items are stored with rows in between. We ran the experiments for 30, 40, 60, 75 and 100 tasks in an instance. Each instance ran for 2, 4 and 8 agents. We now present the results per the number of agents.

In graph 3.6 we can see the waiting time in timesteps for each priority. The waiting time is summed up from all the instances. When we look at the version without priorities, the waiting time is similar for all the priorities. The rest of the lines show our modifications, and we can see a decrease in the waiting time for higher priorities. For this experiment, CBS version with weighted cost proved better than the reassigning tasks and replan modifications. This is caused by the times the tasks enter the system combined with the fact that we only had 2 agents.

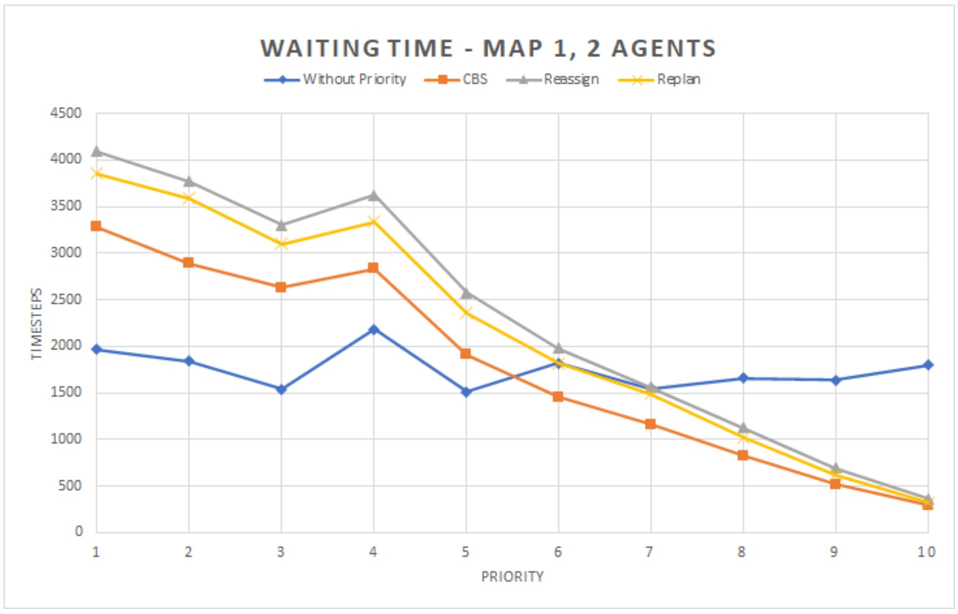

Figure 3.6: Waiting time for instances with 2 agents in warehouse-style corridors map

For example, if an agent has a task with priority 9 assigned but does not reach the pickup location by the end of the window and a task with priority 10 has entered the system in the meantime, the agent may get the task with the priority 10 assigned. The previous task stays in the priority queue or is assigned to a different agent, which can be further from the pickup location. This can cause prolonged waiting time, even for higher priorities.

Next, we will look at the graph 3.7 which shows us summed up running times for each priority. We can see that the results are very similar for all the modifications of the algorithm. However, the only significant difference is the algorithm with no priorities compared to all the other algorithms. The running time is influenced by the timestep a task starts being executed, obstacles in its way and its priority. Since there are only two agents, they do not have to modify their paths much to avoid collisions with the other agents. The main factor here is the timestep in which a task starts being executed.

Now we will look at the graphs where we had four agents executing the tasks. The summed up waiting time is showed in graph 3.8. We can see that CBS was performing the best for almost all the higher priorities. There is barely any difference between the reassign and the replan modification. These two modifications performed slightly better than CBS only for priority 10.

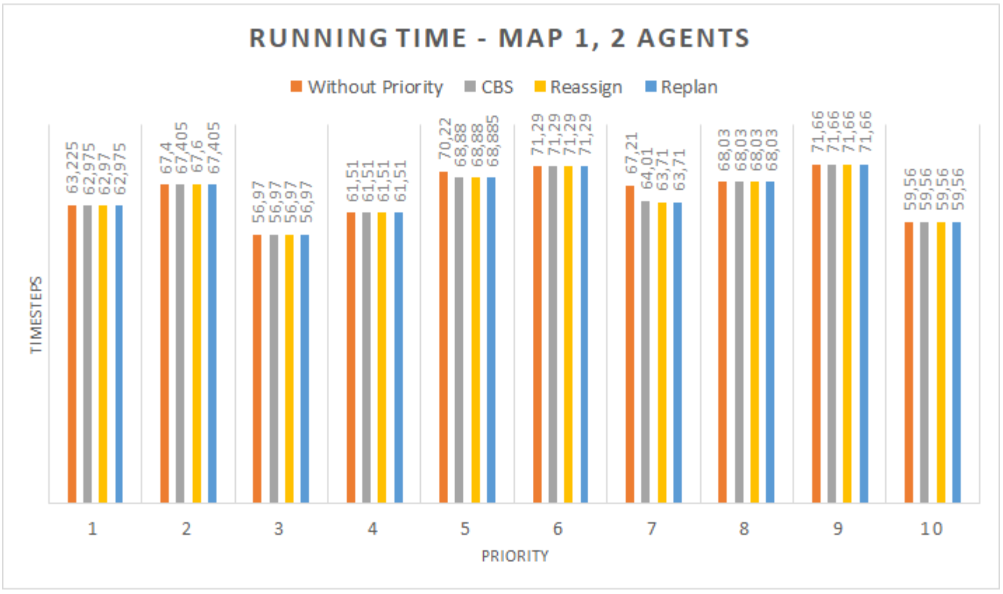

Figure 3.7: Running time for instances with 2 agents in warehouse-style corridors map

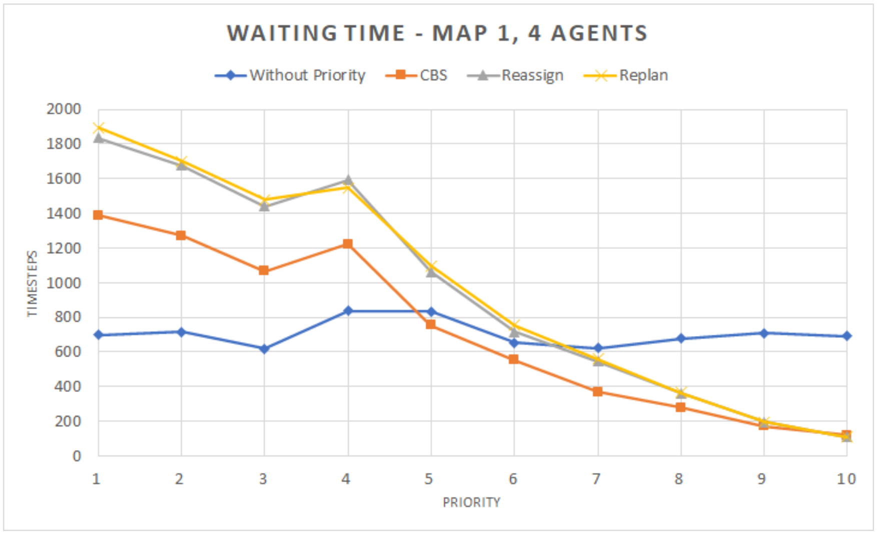

Figure 3.8: Waiting time for instances with 4 agents in warehouse-style corridors map

We then present the *sum of costs* in graph 3.9. Its reasults are very similar to the results in the graph with waiting times; there were barely any differences between the running times of the tasks. In the sum of costs graph, only the reassign version performed better than CBS for priority 10. In both of these graphs, we can see that the reassign and the replan version performance improves for higher number of agents compared to the CBS version.

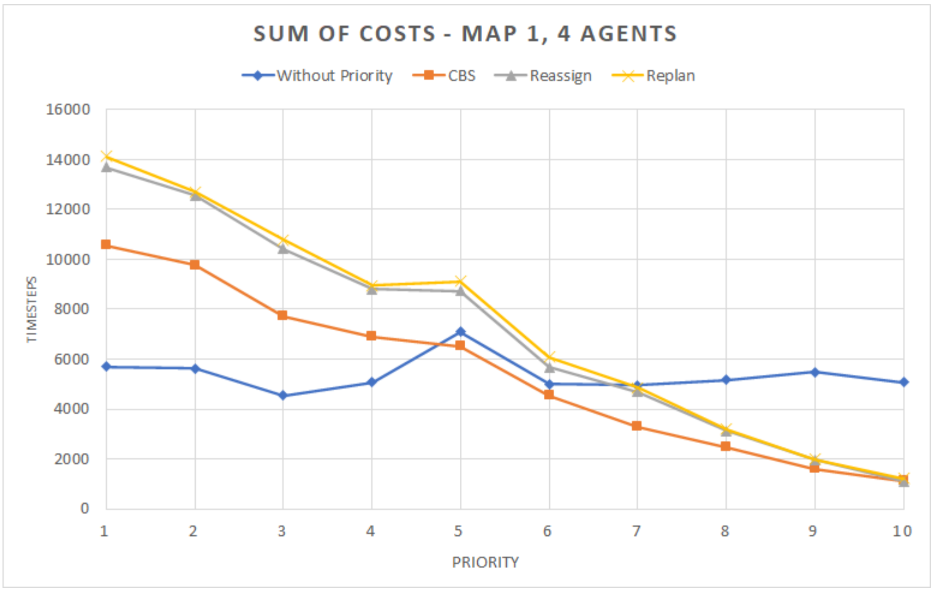

Figure 3.9: Sum of costs for instances with 4 agents in warehouse-style corridors map

Now, let us have a look at the solution evaluation for instances with eight agents. First, we will discuss a graph with summed up average waiting times 3.10. This time, we can see that the reassign and replan version outperformed the CBS version for priorities higher than seven. Replan performed the best for priorities nine and ten. The sum of costs graph 3.11 is similar to the graph with waiting times. This means the running times did not change much between versions.

We saw that the higher the number of the agents is, the better some modifications perform. It is caused by having more agents to assign a task to, and therefore when a task is reassigned or a whole solution is replanned, the number of timesteps to process high priority tasks decreases. Suppose there are just two or four agents. In that case, the reassignment can be counterproductive in the sense of assigning a higher priority task when the agent is already on the way to the pickup location of a lower priority task. This increases the time of the lower priority tasks since there may not be enough agents to assign them in the same timestep.

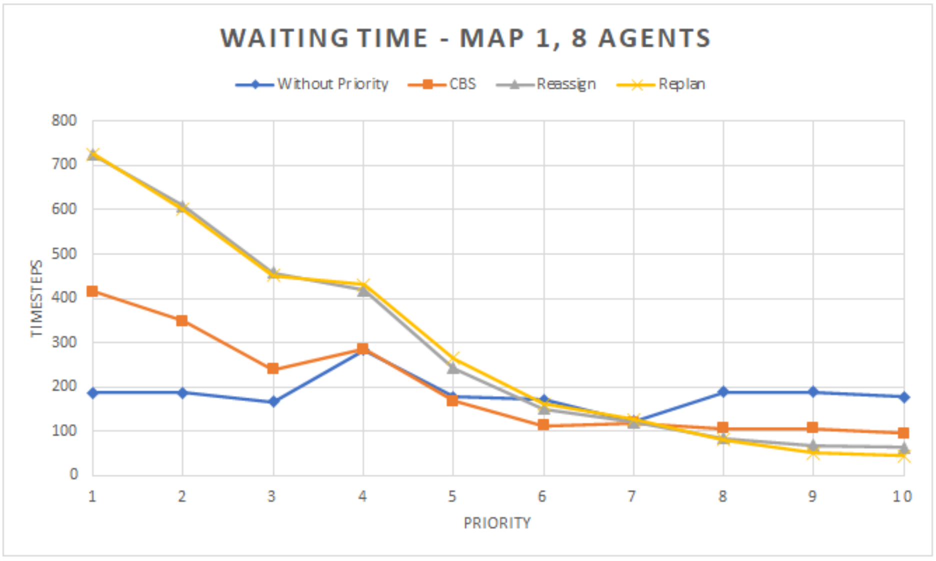

Figure 3.10: Waiting time for instances with 8 agents in warehouse-style corridors map

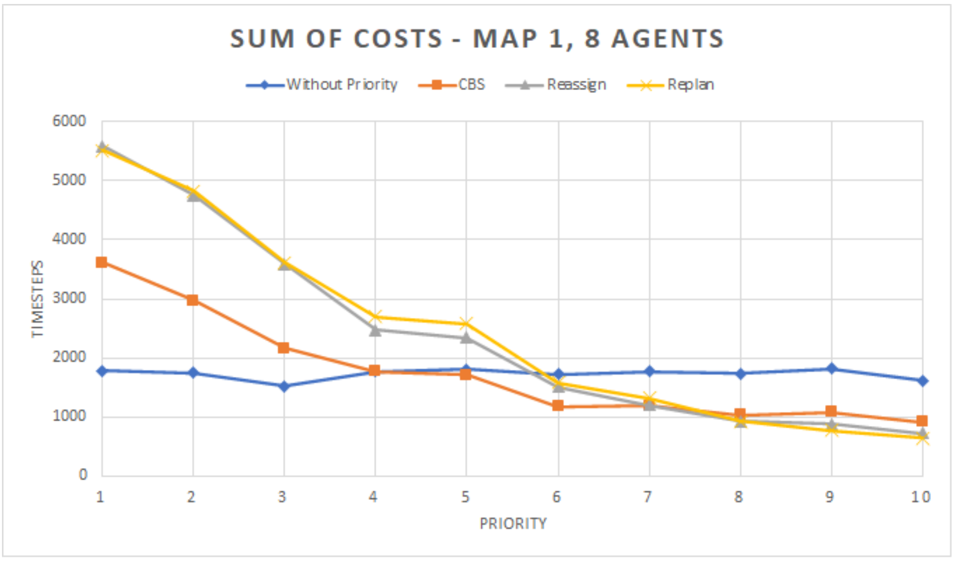

Figure 3.11: Sum of costs for instances with 8 agents in warehouse-style corridors map

At the end of this section, we will discuss the running time of the program. The graph 3.12 shows average running time per instance. Each instance ran three times, with two, four and eight agents. We can see that the version without any priorities was the fastest for most of the instances, followed by the CBS version; reassign and replan running time varied. Replan version running time is higher due to the higher number of low-level searches. Reassign version can run for a longer time since it can assign high priority tasks even when another task was assigned to an agent and the agent already started moving to the pickup point. For this reason, the low-level search is called more times, and that prolongs the running time. The running time also highly depends on how the tasks are sorted and on the w size.

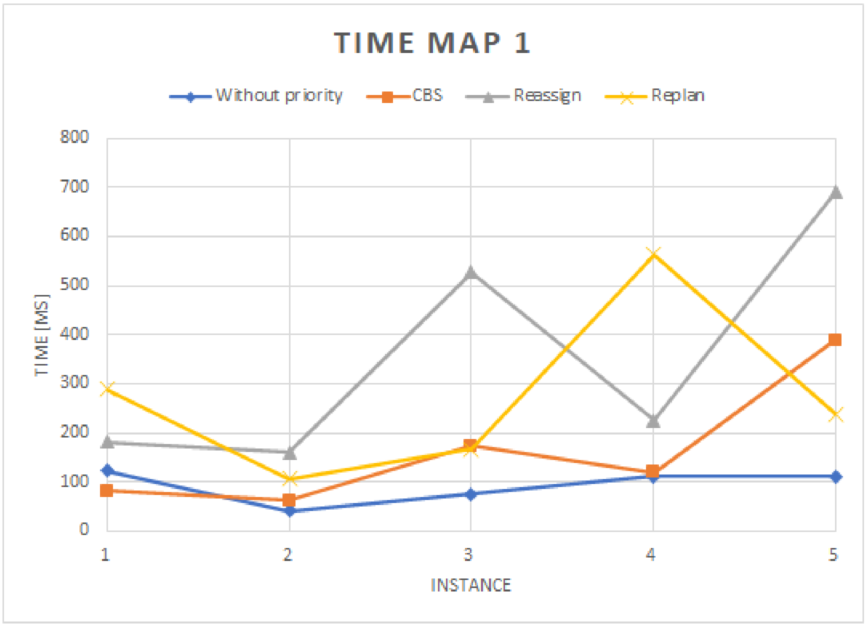

Figure 3.12: Running time of the program for warehouse-style corridors map

3.4 Centralized Pickups Map

In this section, we will test our program on a map with the items centralized in the middle of the map. We can see the floor plan in the picture 3.13. There is a big obstacle in the centre of the map, and the pickup locations are around this obstacle. Delivery locations are around three edges of the map, and non-task endpoints are in the right corner of the map. The colouring scheme in the picture is the same as in the previous map.

We already explained the impact of a different number of agents on the solution evaluation in the previous section, so in this section, we will show only summarized experimental results. The results are summed up for all instances and all agents and divided by three (= instance is run for two, four and eight

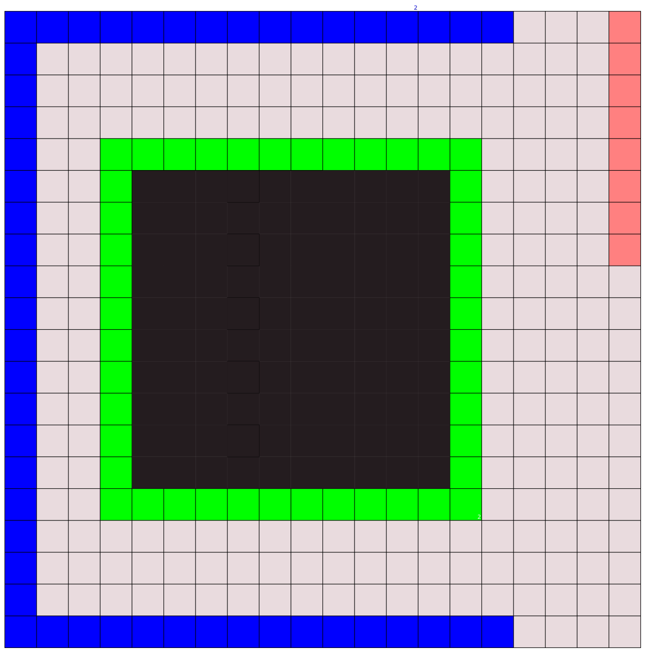

Figure 3.13: Experimental map 2 - Centralized pickups map

agents). The presented results then show the average summed up result for all instances for a specific configuration of number of agents.

The waiting time is shown in this graph 3.14. We can see that, on average, replan version has the best results when it comes to high priorities. This means the improvement for a higher number of agents overweighted a slight deterioration in the quality of the solution for lower number of agents. All the implemented modifications performed better for higher priorities than the no priority version.

Let us look at the *sum of costs* in graph 3.15. We can see here the sum of the waiting and running time. The graph is similar to the graph with waiting times. Once again, all the newly implemented versions overperformed no priority version for higher priorities.

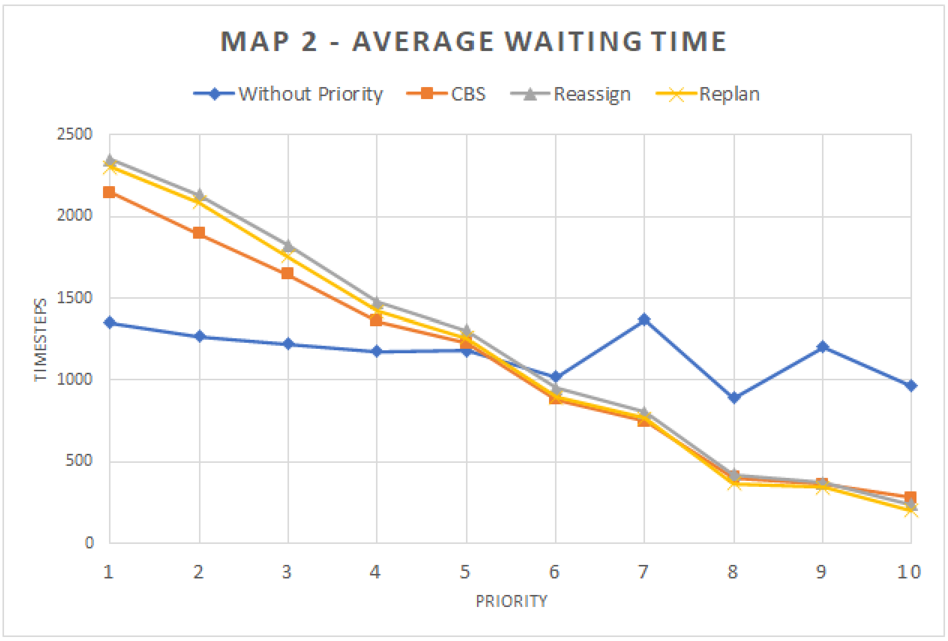

Figure 3.14: Waiting time for instances in centralized pickups map

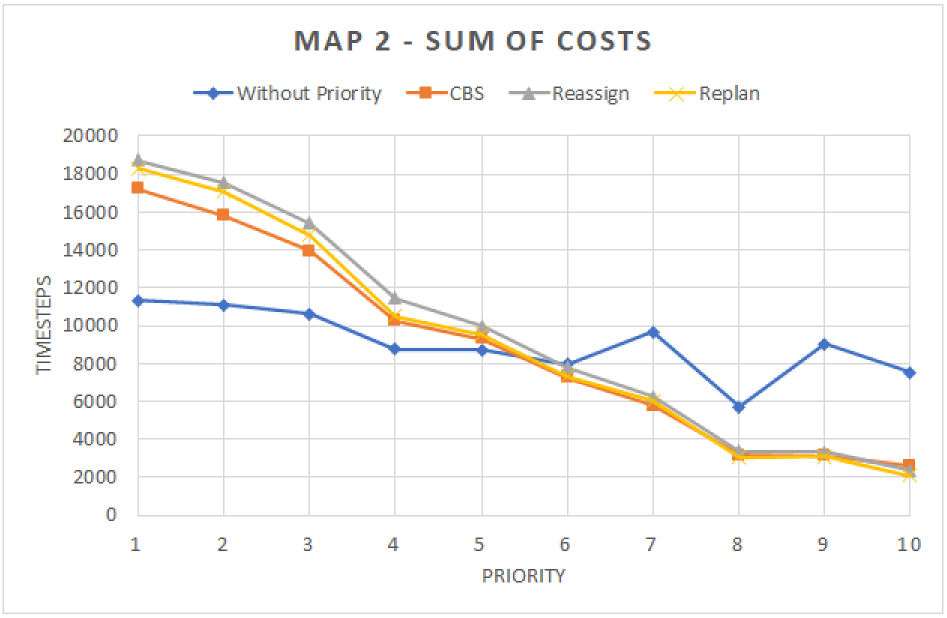

Figure 3.15: Sum of the costs for instances in centralized pickups map

We will discuss program running time now; the graph 3.16 represents the running time of the program the same way as we did for map one. When we compare the running times to the previous map, the program ran longer for instances in this map. This is caused by using Manhattan distance as a heuristic in the CBS low-level search instead of the actual distance. Since the this map had a significant obstacle in the middle of the map, it took more time for the algorithm to find the solution.

The running times differ a lot between each version. The reassign version bears the best results; the rest varies greatly. As in the warehous-style corridors map, the time depends a lot on the order in which the tasks are being assigned.

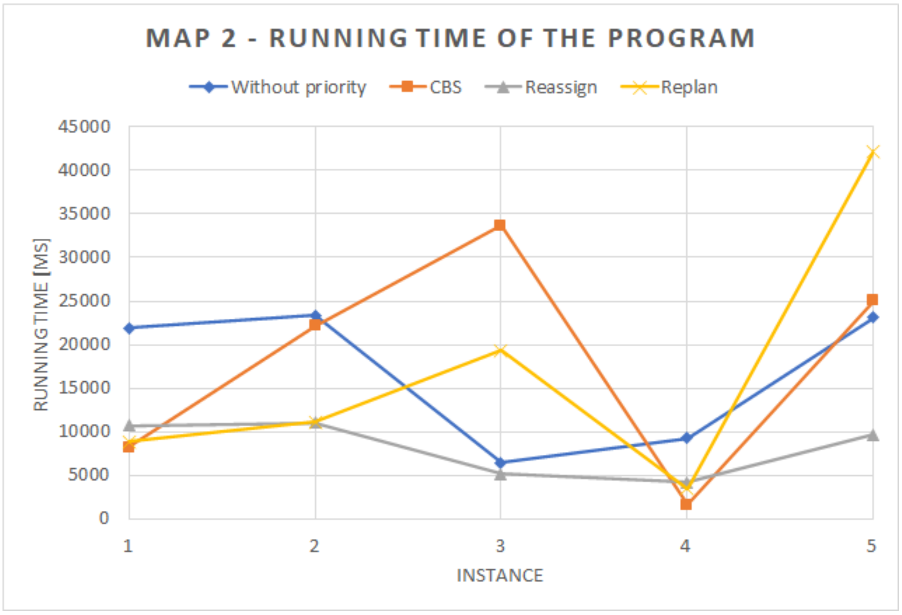

Figure 3.16: Running time of the program for centralized pickups map

3.5 Maze Map

The maze map 3.17 represents randomly distributed obstacles around which are the pickup locations. The non-task endpoints are in the top-left and bottom-left corner. The delivery locations are along the right edge. We will run the test the same way we did for the previous maps.

We can see the comparison of waiting times for the maze map in graph 3.18. The results of the version with no priorities does not change much based on the priority. The results of the rest of the versions are pretty similar. The CBS version performs the best until priority 7, versions reassign and replan outperform the CBS version for priorities 9 and 10. This is caused by the

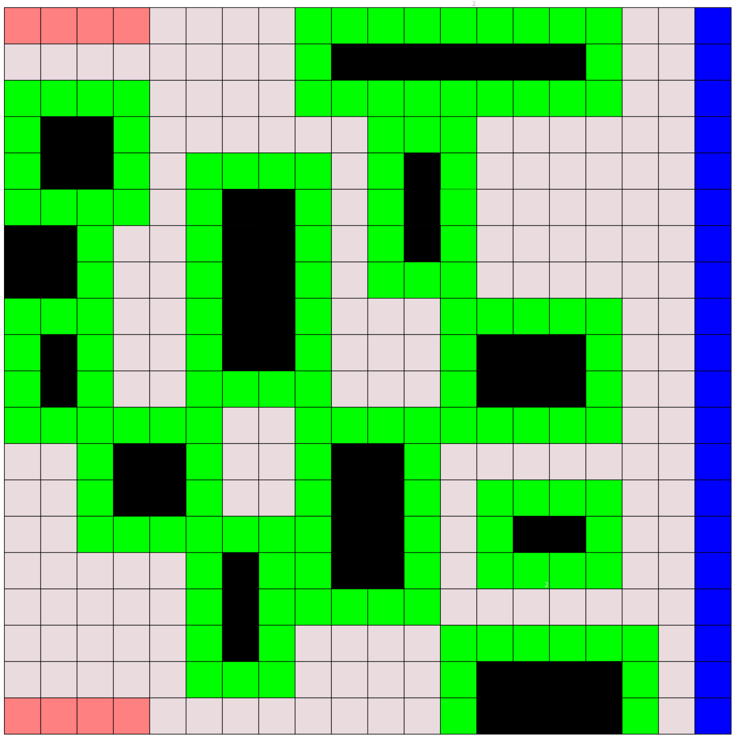

Figure 3.17: Experimental map 3 - maze map

immediate replanning in the replan version, and the task reassignment in the reassign version.

If we look at graph 3.19, we can see that the differences between the versions in the *sum of costs* are very similar to the differences between the waiting times. This means the running time for all the versions was almost the same. This can be caused by using the priority queue in all the modifications (but for the no priority version). Since the tasks are assigned based on their priorities, it does not happen often; there would be tasks with very different priorities running simultaneously. For this reason, the CBS weighted cost does not change the result significantly.

The actual running time of the program differs a lot for each instance and each version; we will discuss it in the experiments' summary.

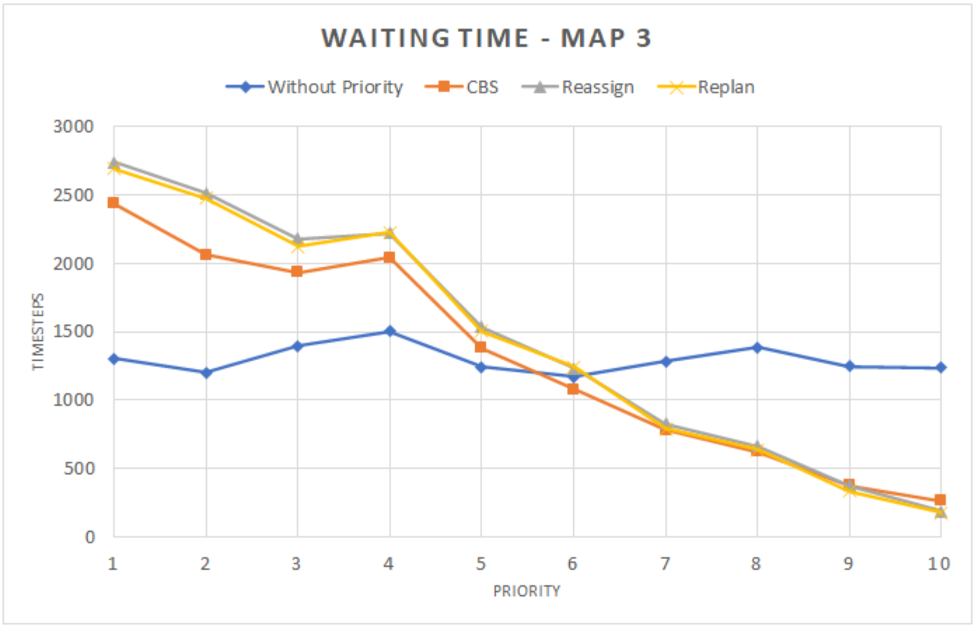

Figure 3.18: Waiting time for maze map in timesteps

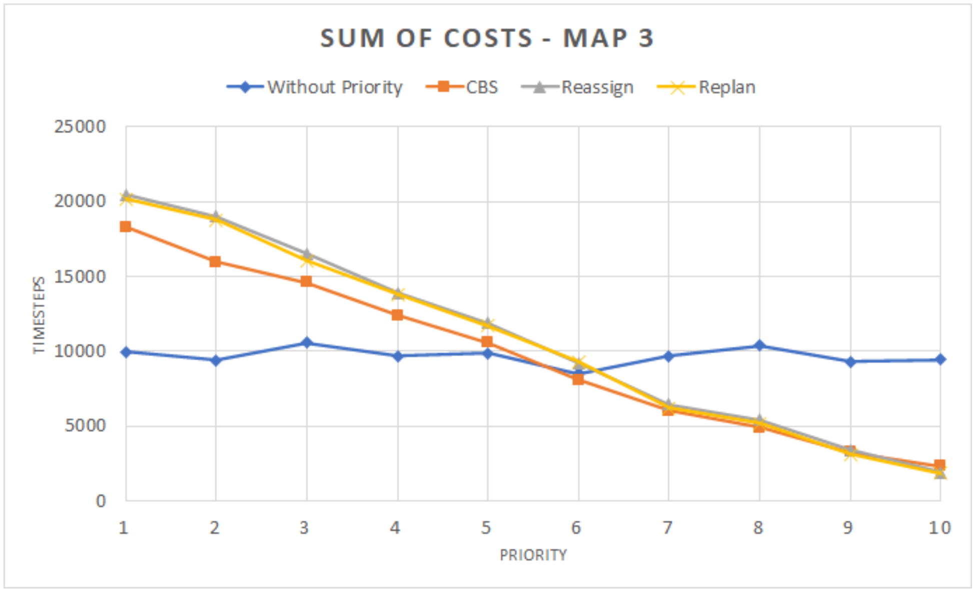

Figure 3.19: Sum of costs for maze map in timesteps

3.6 Stripe Map

The last map 3.20 represents a warehouse as a long stripe with an obstacle on the left side of the map, items pickup locations around it and delivery locations along the right edge. We will run the test the same way we did for the previous maps.

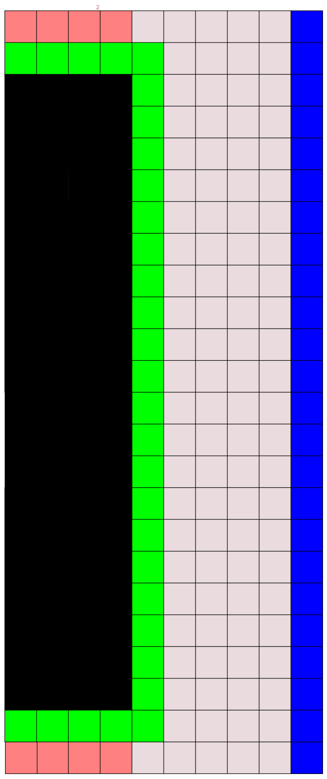

Figure 3.20: Experimental map 4 - stripe map

We will first look at the graph with waiting times 3.21. The replan version performed the best for priorities 6 and higher, followed by the reassign version for priorities 8 to 10. The CBS version performed the best for all the priority modifications for low priorities up to 5. No priority version performed similarly for all the priorities.

Now, let us present the graph with the *sum of costs* 3.22. It is very similar to the graph with waiting times, meaning the running times in timesteps were similar for all the versions.

The results from this map correspond with the results from the previous map. The replan version works well in this type of map in the sense of replanning immediately the path when a high priority (9 or 10) task enters the system. Task reassigning also helps to slightly improve the solution.

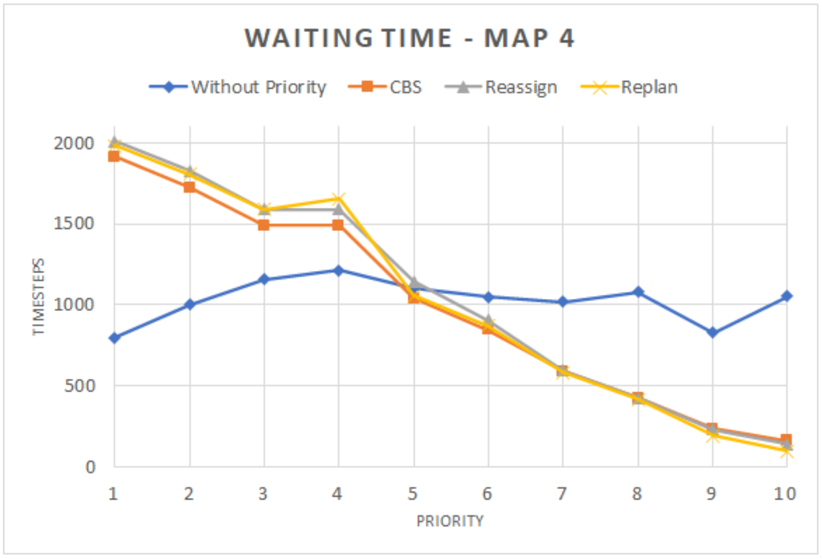

Figure 3.21: Waiting time for stripe map in timesteps

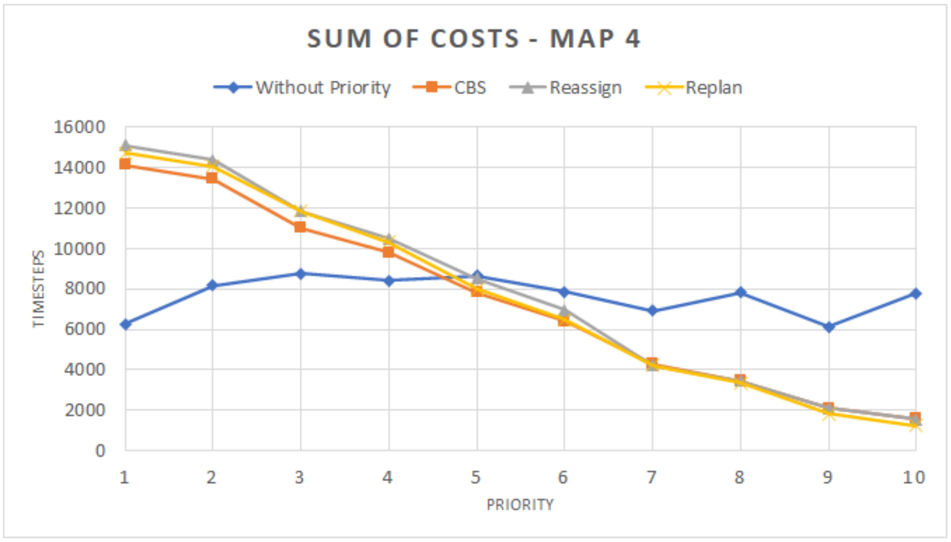

Figure 3.22: Sum of costs for stripe map in timesteps

3.7 Experiments Summary

In this section, we will present graphs that summarize the results for all of the experiments. First, we will look at the graph of the waiting time measured in timesteps 3.23. We can see that the no priority version performs the best for the first half of the priorities. From priority 6 onwards, all the other modifications have better results than the no priority version. The CBS modification performed the best for priorities 6 to 8. The reassign version performed slightly worse than the replan version for all the priorities. This is caused by the replan version being an improvement of the reassign version.

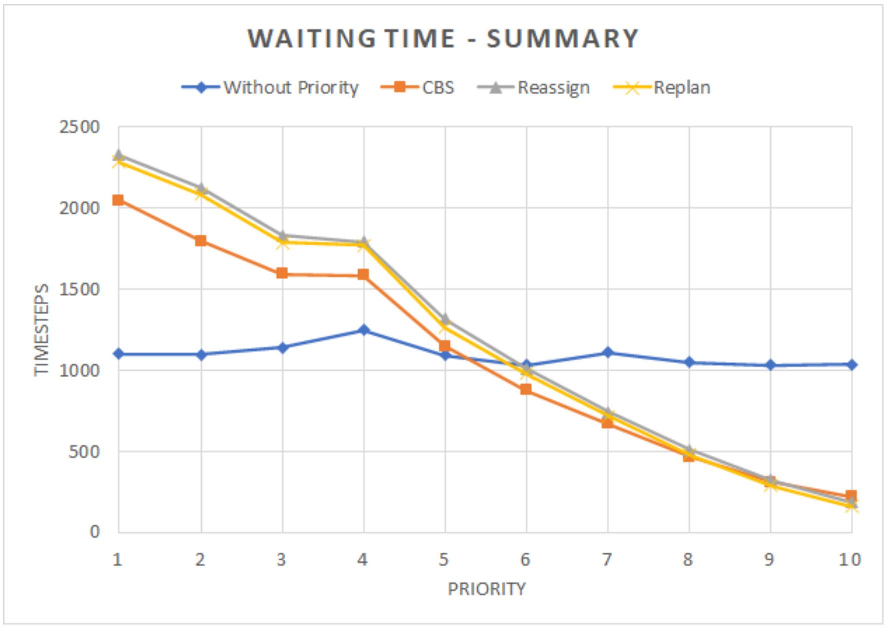

Figure 3.23: Summary of waiting times in timesteps

Now we will focus on the running time of the tasks in timesteps. In the graph 3.24, there are only slight differences between the number of timesteps within each priority. Nevertheless, we can see that the version with no priorities performed slightly worse for tasks with higher priorities.

Let us discuss the *sum of costs* which includes both running and waiting times. In graph 3.25, we can see the *sum of costs* in timesteps for all the versions and priorities. If we look at the higher half of the priorities, we can see that CBS modification performs the best for priorities 6 to 8. The replan and reassign versions both outperform the CBS version in priority 9, and the replan modification performs the best for priorities 9 and 10. This result complies with the setting of immediate recomputing of the path when a task with priority 9 or 10 enters the system.

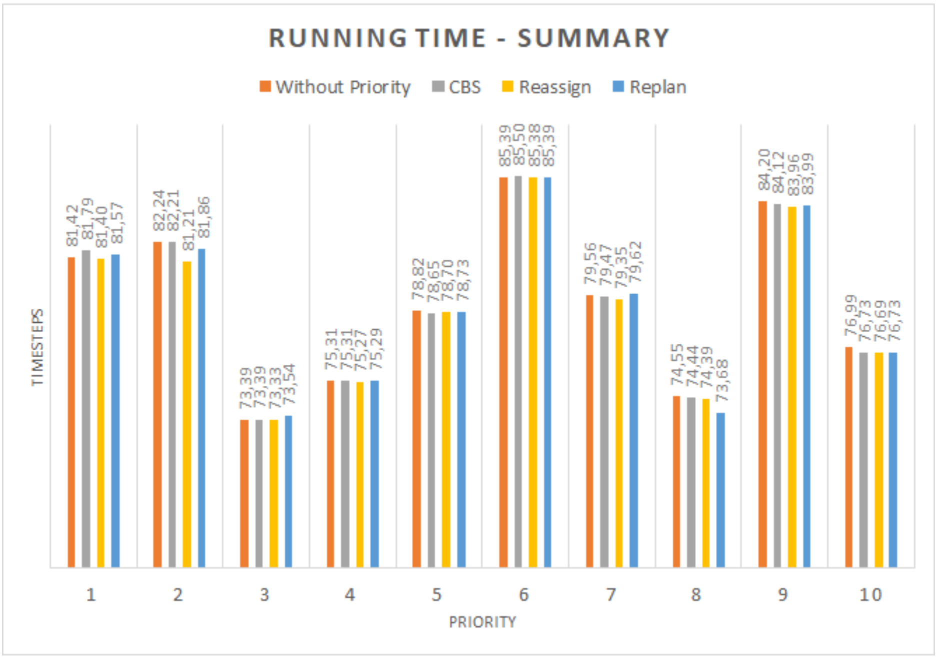

Figure 3.24: Summary of running times in timesteps

Finally, we will discuss the running time of the program. We can see the average time in milliseconds for each instance 3.26. The graph shows an average time to run a certain number of tasks (30, 40, 60, 75 and 100) on different maps with a different number of agents (two, four and eight). We can see that the priority modifications performed better than the no priority version for instances with 30 and 40 tasks. For the following instances, the time fluctuates greatly. The time change is caused mainly by the different order of the tasks that are being assigned to the agents. A different order can cause more conflicts between the agents or longer paths, both of which contribute to prolonging the running time of the program. We can see that the replan version was underperforming for the instances with 100 tasks. This can be caused by a higher amount of solution plannings compared to the other algorithms since this version replans the solution anytime a high priority (9 or 10) enters the system. Overall, we did not find high dependency on the version and the program's running time other than in the replan version.

This chapter presented the main results of the conducted experiments, all the results and measured data can be found in the enclosed CD.

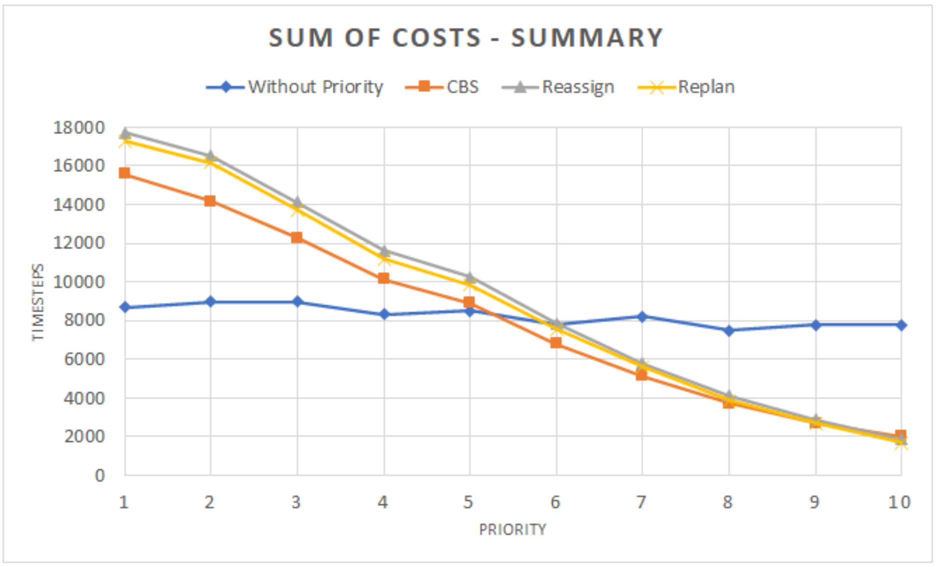

Figure 3.25: Summary of sum of costs in timesteps

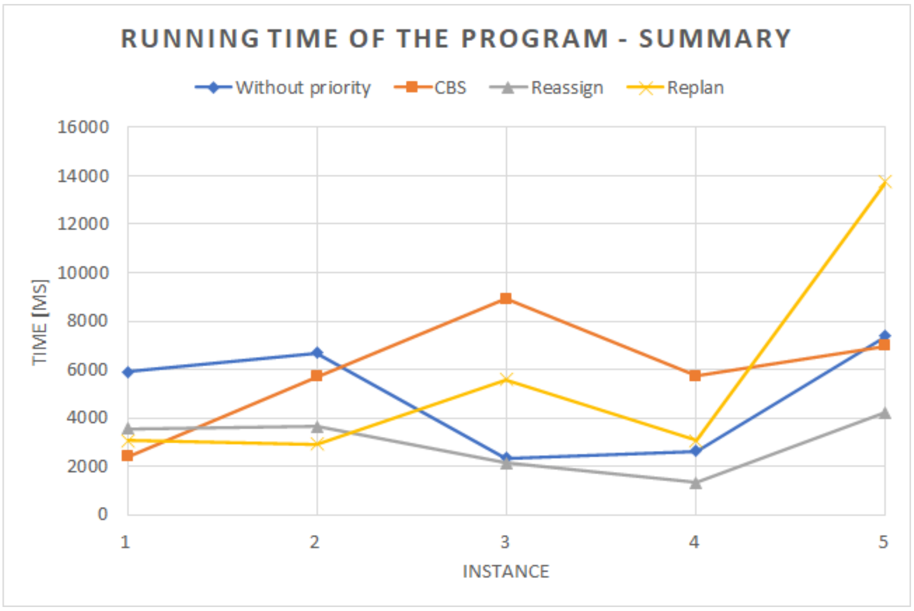

Figure 3.26: Summary of running time of the program in milliseconds

Conclusion

The primary purpose of this thesis was to develop techniques for warehouse logistics. Specifically, to analyze modifications of an existing MAPF algorithm that would handle priorities effectively. Another goal was to design and implement these improvements of a MAPF algorithm. The last goal was to analyze the results. The first part of testing focused on setting parameters, such as parameter w to determine running periods in the lifelong algorithm, while the second part of the experiments focused on running different algorithm modifications in several warehouse-style maps on various instances with multiple robots.

We managed to accomplish all the goals of the thesis. To achieve them, we researched logistics in automated warehouses with a focus on MAPF and MAPD problems. We focused on MAPF algorithms and lifelong approaches to modify these algorithms, as well as possible improvements for a Windowed CBS algorithm to consider task priorities.

We designed and implemented three different Windowed Priority CBS algorithm modifications. The first modification included CBS low-level A* search using a weighted cost based on the priority of the task and using a priority queue for assigning tasks based on their priority. The second modification included CBS weighted cost and priority queue modifications; in addition to that, we implemented task reassignment for the robots. After each run, if a robot has not reached the start location of an already assigned task and there was a higher priority task in the priority queue, the robot would return the assigned task to the priority queue and be assigned the task with a higher priority. The third modification added immediate replanning if a high priority task enters the system.

We compared Windowed Priority CBS modifications experimental results to a Windowed CBS algorithm and within each other. We discovered that the modifications' quality of the solutions depends on the number of robots processing the tasks. The higher the number of robots was, the more effective task reassigning and immediate replanning was. For a lower amount of robots, modification with CBS weighted cost and priority queue performed the best. If we compare our algorithm modifications to Windowed CBS, all of the modifications performed better for the higher half of the priorities on average. When talking about the program's running time, the version with immediate replanning had the highest running time on average. For the rest of the modifications and the Windowed CBS, the running time depends on the tasks' order the most.

In the future, we can build upon this algorithm by adding other important features for warehouse logistics, such as humans walking in the warehouse or changing the task priority after it already entered the system. We built a solid software prototype for handling task priorities in automated warehouses, and our experimental results can be later used for further research on this topic.

Bibliography

[1] Wurman, P.; D'Andrea, R.; et al. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI Mag. 29(1)*, 9 2008.

[2] Veloso, M.; Biswas, J.; et al. CoBots: robust symbiotic autonomous mobile service robots. *IJCAI*, 2015: p. 4423.

[3] Ma, H.; Yang, J.; et al. Feasibility study: moving non-homogeneous teams in congested video game environments. In *Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2017, pp. 270–272.

[4] Bounini, F.; Gingras, D.; et al. Online Trajectory Planning With a Modified Potential Field Method on Distributed Architectures for Autonomous Vehicles. *ITS World Congress 2017 Montreal*, 2017, [cit. 2021-03-08]. Available from: `https://blob.opal-rt.com/medias/L00161_1010.pdf`

[5] Silver, D. Cooperative Pathfinding. *Aiide*, volume 1, 2005: pp. 117–122.

[6] Buro, M.; Furtak, T. M. RTS games and real-time AI research. In *Proceedings of the Behavior Representation in Modeling and Simulation Conference (BRIMS)*, volume 6370, 2004.

[7] Botea, A.; Müller, M.; et al. Using Abstraction for Planning in Sokoban. In *Computers and Games*, edited by J. Schaeffer; M. Müller; Y. Björnsson, Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 360–375.

[8] Stern, R. Multi-agent path finding–an overview. *Artificial Intelligence*, 2019: pp. 96–115.

[9] Barraquand, J.; Latombe, J.-C. Robot motion planning: A distributed representation approach. *The International Journal of Robotics Research*, volume 10, no. 6, 1991: pp. 628–649.

[10] LaValle, S. M.; Hutchinson, S. A. Optimal motion planning for multiple robots having independent goals. *IEEE Transactions on Robotics and Automation*, volume 14, no. 6, 1998: pp. 912–925.

[11] Erdmann, M.; Lozano-Perez, T. On multiple moving objects. *Algorithmica*, volume 2, no. 1, 1987: pp. 477–521.

[12] Latombe, J.-C. *Introduction and Overview*. Boston, MA: Springer US, 1991, ISBN 978-1-4615-4022-9, pp. 1–57, doi:10.1007/978-1-4615-4022-9_1. Available from: `https://doi.org/10.1007/978-1-4615-4022-9_1`

[13] Andreychuk, A.; Yakovlev, K. Two techniques that enhance the performance of multi-robot prioritized path planning. *arXiv preprint arXiv:1805.01270*, 2018.

[14] Bnaya, Z.; Felner, A. Conflict-Oriented Windowed Hierarchical Cooperative A*. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, 2014, pp. 3743–3748, doi:10.1109/ICRA.2014.6907401.

[15] Wang, K. C.; Botea, A. MAPP: a Scalable Multi-Agent Path Planning Algorithm with Tractability and Completeness Guarantees. *CoRR*, volume abs/1401.3905, 2014, `1401.3905`. Available from: `http://arxiv.org/abs/1401.3905`

[16] Luna, R.; Bekris, K. E. Push and swap: Fast cooperative path-finding with completeness guarantees. In *IJCAI*, 2011, pp. 294–300.

[17] de Wilde, B.; ter Mors, A. W.; et al. Push and rotate: cooperative multi-agent path planning. In *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*, 2013, pp. 87–94.

[18] Standley, T. Finding optimal solutions to cooperative pathfinding problems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 24, 2010.

[19] Wagner, G.; Choset, H. Subdimensional expansion for multirobot path planning. *Artificial Intelligence*, volume 219, 2015: pp. 1–24, ISSN 0004-3702, doi:https://doi.org/10.1016/j.artint.2014.11.001. Available from: `https://www.sciencedirect.com/science/article/pii/S0004370214001271`

[20] Sharon, G.; Stern, R.; et al. The increasing cost tree search for optimal multi-agent pathfinding. *Artificial Intelligence*, volume 195, 2013: pp. 470–495, ISSN 0004-3702, doi:https://doi.org/10.1016/j.artint.2012.11.006. Available from: `https://www.sciencedirect.com/science/article/pii/S0004370212001543`

[21] Surynek, P. Makespan Optimal Solving of Cooperative Path-Finding via Reductions to Propositional Satisfiability. *CoRR*, volume abs/1610.05452, 2016, `1610.05452`. Available from: `http://arxiv.org/abs/1610.05452`

[22] Sharon, G.; Stern, R.; et al. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, volume 219, 2015: pp. 40–66, ISSN 0004-3702, doi:https://doi.org/10.1016/j.artint.2014.11.006. Available from: `https://www.sciencedirect.com/science/article/pii/S0004370214001386`

[23] Ma, H.; Li, J.; et al. Lifelong Multi-Agent Path Finding for Online Pickup and Delivery Tasks. *CoRR*, volume abs/1705.10868, 2017, `1705.10868`. Available from: `http://arxiv.org/abs/1705.10868`

[24] Cáp, M.; Vokrínek, J.; et al. Complete Decentralized Method for On-Line Multi-Robot Trajectory Planning in Valid Infrastructures. *ArXiv*, volume abs/1501.07704, 2015.

[25] Li, J.; Tinka, A.; et al. Lifelong Multi-Agent Path Finding in Large-Scale Warehouses. 2021, `2005.07371`.

[26] Grenouilleau, F.; van Hoeve, W.-J.; et al. A multi-label A* algorithm for multi-agent pathfinding. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 29, 2019, pp. 181–185.

[27] Okoso, A.; Otaki, K.; et al. Multi-Agent Path Finding with Priority for Cooperative Automated Valet Parking. 10 2019, pp. 2135–2140, doi:10.1109/ITSC.2019.8917112.

[28] Felner, A.; Goldenberg, M.; et al. Partial-Expansion A* with Selective Node Generation. *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 26, no. 1, Jul. 2012. Available from: `https://ojs.aaai.org/index.php/AAAI/article/view/8137`

Appendix A

Windowed Priority CBS

We decided to implement the algorithm in Java based on this language general usage. The project has been built in NetBeans, and the source code is available in the enclosed CD.

A.1 Program input

The program takes a file as input. The file contains agents and tasks specifications, and a map. The file structure is following:

- Dimensions of the map
- Number of agents
- Initial locations of the agents
- Number of endpoints
- Locations of the endpoints
- Number of tasks
- Task specifications
  - Priority
  - Time
  - Start location
  - Finish location
- Map
  - 1 - feasible location
  - 0 - unfeasible location

A.2 Data storage classes

Data class

The input file is processed in class Data. This class stores the data from the input file; besides that, it contains methods to assign new tasks to agents and update agents.

Read File

This method takes $filename$ as a parameter and reads input data from the input file. It saves the agents, endpoint locations, tasks and map data. All the tasks are saved in an array, no matter at what time they enter the system. Priority queue fetches them out of the array according to their specified entering time.

Assign tasks

Parameters:

- Timestep number - number of the timestep for which the solution will be planned (parameter w)
- Current timestep

Based on the current timestep, tasks from the task array are moved to the priority queue. Tasks from the priority queue are assigned to the agents based on agents' distance from them, including the distance to fulfill preciously assigned tasks. Tasks are assigned either until the priority queue is empty or all agents have enough tasks for the next *window*.

Update agents

After each run (*window*), tasks in each agent are updated based on how many tasks were finished during that run. Finished tasks are removed from agents.

Agent class

Class variables:

- Tasks $\rightarrow$ tasks assigned to the agent
- Initial location $\rightarrow$ location at the beginning of the run
- Final goal $\rightarrow$ last goal of the agent
- Heuristic $\rightarrow$ heuristic from initial location to the final goal through all the task goals

- Label:
  - 0 → start location of the first task has not been reached yet
  - 1 → start location of the first task has been already reached

Add task

The method takes a task as a parameter. It adds the task to the tasks array, updates the final goal and heuristics.

Remove task

The method takes the task's position in the tasks array as a parameter. It deletes the task from the array. If it was the last task of the array, the final goal is set to the goal location of the previous task; if there are no tasks left, it is set to be the initial location. The heuristic is updated by subtracting the removed task from it.

Task class

This class is used to store information about tasks. Class variables:

- Priority → priority of the task
- Timestep init → timestep when task entered the system
- Start → pickup location
- Goal → delivery location
- Move to endpoint → boolean if the task is only to move the agent to its non-task endpoint
- Timestep start → timestep when start location has been reached

Dimensions

The sole purpose of this class is to save and compare locations as one object. It has two class variables: row and column.

A.3 Windowed CBS

There is one class that takes care of replanning the solution every w steps and another class that supports the high-level CBS and calls A* search for specific agents.

CBS class

This class implements the high-level CBS together with the lifelong approach of the program. It takes data class as an input. Parameter w is specified in this class and can be reset from here. The constructor of the class is running CBS for every w steps.

CBS first calls *data* to assign tasks to the agents, plans the tasks for all the agents and adds the solution to the constraint tree. Nodes of the constraint tree are stored in a priority queue based on the solution cost. While there are nodes in the priority queue and the optimal valid solution has not been found yet, agents are replanned.

At the beginning of each cycle, a node with the lowest cost is opened, and the program searches for conflicts in the joint solution. If no conflicts are found, the node contains the optimal valid solution, the program calls *data* to update the agents, and we can terminate the cycle.

If a conflict is found, two new nodes are created. In each node, we replan the tasks for an agent who has the conflict, count the joint solution cost and add it to the constraint tree.

CBS is repeated until all the tasks are finished (until there are no more tasks in the tasks array). In a real situation, the run would be lifelong.

Get Duplicate

This method search for vertex and edge conflicts in the joint solutions. Parameters:

- Number of agents
- Array of results → agents' paths
- Maximum timestep → last timestep in which it should search for conflicts

Vertex conflicts are found by using structure *set* of Dimensions, which returns false when we try to insert a value that has been inserted before. If false is returned, we go through the previous agents to find the agent with whom it conflicted.

Edge conflicts are computationally more challenging to find. We need to compare each agent with all the other agents, which results in three nested cycles; the first one to go through the agents, the second one to go through the agent's path and the third one to compare Dimensions of a single path location with the rest of the agents.

When a conflict is found, two constraints are created; one for each agent in the conflict. If there are more agents in the conflict, we only consider the first two; the rest will be taken care of on a deeper level of the tree.

Plan Agents class

High-level CBS uses this class to call low-level CBS to get the plans of the agents.

Plan All Agents

Parameters of this method are:

- Data → input data - for searching in *map*
- Constraints → array of Constraint object
- Agents → array of Agent objects

Method calls A* search for each agent using their constraints and returns an array of results.

Plan Agent

It is a very similar method to plan all agents but only plans specific agent.

Get Constraints for Agent

The method takes an array of constraints and agent ID (position of the agent in the agent array) as a parameter and returns an array of the constraints for the specific agent.

A.4 Constraint Tree

The following classes are implementing the constraint tree and handling its operations.

Constraint class

This class represents one constraint. Its class variables are:

- Agent Id → position of the agent in the agents' array
- Node 1 → Dimensions of the constraint node
- Node 2 → Dimensions of the second node if it is an edge conflict
- Time → time of the constraint

- Is node constraint → boolean that specifies whether it is a node or an edge constraint

Collides

This method is the essential method of the constraint class. It takes Dimensions of node (resp. nodes) and time as parameters and returns true if the input collides with the constraint.

Node class

This class represents one node of the constraint tree. Its class variables are:

- Constraint → each node saves one constraint
- Results → path of the agents
- Left → Node object - left child
- Right → Node object - right child
- Parent → Node object- parent of the node

Get Cost

Since the node is not saving the cost directly, we need to get it from the results. This method is using the *results* class to count the cost of the joint solution.

Goal Node

This method returns true if the node contains a valid solution. The solution is valid if the constraint is equal to null. The method returns false otherwise.

Constraint tree class

This class represents the constraint tree used for high-level CBS search. The implemented methods are: add node and get constraints.

Add node

This method adds a new node to the constraint tree. Input parameters:

- Current node
- Constraint
- Results

The method creates a new node based on the input parameters. It sets its parent to be the current node, sets the new node to be the left (resp. right) child of the current node, and returns the newly created node.

Get constraints

This method takes a node as an input parameter. It returns all the constraints in the path from the node to the root.

A.5 CBS Low Level

Low-level CBS takes care of the A* search. It searches for single-agent paths using constraints from the high-level constraint tree.

Element class

An instance of this class represents one element (node) in an A* search queue. The class variables are:

- Location → Dimensions of the element in the input map
- Path cost → cost to get to this location from the agent's initial location
- Heuristic → distance to the agent's final goal (including the passage through all the goals in the way)
- Time → timestep in which the location is reached
- Label
  - 0 → agent is going to the start location of the current task
  - 1 → agent is going to the goal location of the current task
- Task number → number of the task that is being processed (position in the agent's tasks array)
- Path → path to get to this location from the agent's initial location

The purpose of this class is to store essential data to be able to find the optimal path with the A* search.

A* search class

This class implements an A* search on a single agent with multiple sorted goals. We use manhattan distance as a heuristic multiplied by the priority of the task.

Element Comparator

This class was implemented to compare two *elements* between each other based on their path cost and heuristic.

Get Heuristic for Element

This function counts the heuristic for the specified element. The heuristic is calculated based on the task number and label in the element.

Dimensions Valid

This method checks if the newly generated dimensions are valid. If they are still on the map and whether or not they are feasible. Returns true if dimensions are valid, otherwise returns false.

Constraint Check

Method Constraint Check takes dimensions *start* and *goal* and *time* as an input. It checks whether the agent can move from location *start* to location *goal* in a timestep *time* or a there is a constraint that would forbid the agent from performing said action.

Visited Element

This method checks whether an Element with the same location and time has already been visited before. If yes, it checks for its path cost and heuristics and compares the elements. If the path cost and heuristic of the input element is lower than the one of the already visited element, the method returns false. If the path cost and heuristic of the new element is higher than the previously visited element, the method returns true.

Search

This method implements the A* search for a single agent. It takes an agent as an input parameter. In the beginning, it creates a search queue for elements and adds the first element to the queue, which is the initial location of the agent in timestep zero. The label of the element is set according to the label in the agent. The task number is zero, and the path is null.

While the search queue is not empty, A* is searching for the solution. The element with the lowest path cost and heuristic is processed and removed from the queue in each cycle.

First of all, we need to check whether or not the element's location equals the current goal. This is done by comparing the location to the element's task number start if the element's label equals zero or to the goal if the element's label equals one. If a goal is found and the label is zero, a new element is added to the search queue. The new element has the same properties as the current element; the only difference being the label, which is set to one. If a goal is found and the label is one, a new element is also added to the search queue, but this time we increase the task number by one and set the label to zero.

The cycle continues by creating discovering neighbouring locations of the current element. Neighbouring locations are validated using methods *dimensionsvalid*, *constraintcheck* and *visitedelement*. If the new elements pass the check, they are added to the search queue.

When the final goal is reached, path, path cost and timesteps in which each task was started and finished are saved into the results and returned.

Results class

Results class is used for storing the results of a single-agent search. The main class variables are:

- Final path cost - the cost of the path
- Path - an array of Dimensions
- Finished tasks - table of integers

Finished tasks

The finished tasks table has two rows. The first row represents the start location, and the second row represents the goal location. It stores the timestep number in which each of the locations was reached in the path.

As an example, we can look at table A.1. We can see three tasks there, the start location of the first task was reached in timestep 2, the goal in timestep 10, and so forth, for the second and the third task. The second task's start location was reached in timestep 12, and the goal location was reached in timestep 16.

Table A.1: Finished tasks table

	0	**1**	**2**
0	2	12	19
1	10	16	25

A.6 Solution properties

In this section, we will discuss the implementation of the measurement of the optimality of the solution. All the solution optimality data are stored in the Result optimality class.

Result optimality class

This class is used to evaluate the returned solution. The class contains four arrays:

- Average waiting time
- Average run time
- Number of finished tasks
- Sum of costs

Each position in an array represents one priority. For example, suppose we want to know the solution evaluation of the tasks with priority 1. In that case, we can look at position one of the array *sum of costs* or any other array of the mentioned ones to give us more information on how the program performed for this priority.

Appendix B

Acronyms

CA*-Pri	Cooperative A* with priority
CBS	Conflict-based search
CBS-Pri	Conflict-based search with priority
COBRA	Continuous Best-Response Approach
CT	Constraint tree
EPEA*	Partial-Expansion A* with Selective Node Generation
ICTS	Increasing cost tree search
MAPF	Multi-agent pathfinding
MAPD	Multi-agent pickup and delivery
MAPP	Multi-Agent Path Planning Algorithm
TP	Token passing

Appendix C

Contents of enclosed CD

```
readme.txt ....................... the file with CD contents description
├── experimental_results ...... the directory with result tables and graphs
├── src ..................................... the directory of source codes
│   ├── doc ................... the directory with javadoc of the source codes
│   ├── example_input_files ... the directory with example input files for the
│   │   program
│   ├── thesis ............. the directory of LaTeX source codes of the thesis
│   └── windowed_priority_CBS .. the directory with Windowed Priority CBS
│       source code
└── text ......................................... the thesis text directory
    └── DP_Dvorakova_Klara_2021.pdf ........ the thesis text in PDF format
```