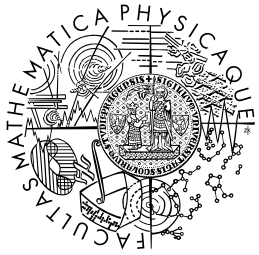


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Petr Šmíd

Algorithms for Multi-Agent Pickup-and-Delivery Problems

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis: prof. RNDr. Roman Barták, Ph.D.
Study programme: Computer science
Study branch: General Informatics

Prague 2023

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In Prague on July 20th, 2023 Petr Šmíd

I wish to thank my supervisor prof. Roman Barták for his valuable advice and for pointing me in the right direction. Also to my parents for letting me create this thesis under their roof.

Title: Algorithms for Multi-Agent Pickup-and-Delivery Problems

Author: Petr Šmíd

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: prof. RNDr. Roman Barták, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: In this thesis, we explore the world of Multiagent pickup and delivery algorithms. Basic definitions, as well as simple extensions, are introduced to the reader. State-of-the-art algorithms are thoroughly described, analyzed, and tested in various environments based on multiple conditions. We describe the scalability of the algorithms and demonstrate it in multiple scenarios. The thesis includes a short overview of explainable plans, their motivation, and their implementation. Support software was created for conducting experiments, visualization, and making explainable plans.

Keywords: MAPD MAPF Autonomous agents

Contents

List of Abbreviations **3**

Introduction **4**

1 Problem Definition **5**
1.1 MAPD Formalization . . . 5
1.1.1 Well formed MAPD Instance . . . 6
1.2 Additional settings . . . 7
1.2.1 Deadline for orders . . . 7
1.2.2 Capacities . . . 7
1.2.3 Multi-goal tasks . . . 7
1.2.4 Task ordering . . . 7
1.2.5 Agent charging . . . 7
1.2.6 Offline . . . 8
1.3 MAPF . . . 8
1.4 Solving methods . . . 8
1.4.1 Task assignment . . . 8
1.4.2 Path planning . . . 9
1.5 Example . . . 9

2 Related work **10**
2.1 MAPF research . . . 10
2.1.1 Hierarchical cooperative A* . . . 10
2.1.2 Increasing cost tree search . . . 10
2.1.3 Conflict based search . . . 11
2.1.4 Constraint programming . . . 11
2.1.5 Learning based algorithms . . . 11
2.2 MAPD research . . . 11
2.2.1 Offline settings . . . 11
2.2.2 Online settings . . . 12
2.2.3 Semi-online settings . . . 13

3 Algorithm description **14**
3.1 Token Passing . . . 14
3.2 Token Passing Task Swapping . . . 15
3.3 CENTRAL . . . 17
3.3.1 Hungarian Algortihm . . . 18
3.3.2 Path computing . . . 18
3.3.3 CBS . . . 18
3.3.4 Update . . . 19
3.3.5 Prunning . . . 19
3.4 Possible pitfalls . . . 20
3.4.1 TP and TPTS . . . 20
3.4.2 CENTRAL-A* . . . 20
3.4.3 CENTRAL-CBS . . . 21

4 Plan Validation **22**
4.1 Validation algorithm . . . 22
4.2 Properties . . . 22

5 Experiments **24**
5.1 Experiment descriptions . . . 24
5.1.1 MAPF benchmarks . . . 25
5.2 Experiment 1 . . . 25
5.2.1 Settings . . . 25
5.2.2 Results . . . 26
5.3 Experiment 2 . . . 31
5.3.1 Settings . . . 31
5.3.2 Results . . . 32
5.3.3 Conclusion . . . 36
5.4 Experiment 3 . . . 37
5.4.1 Settings . . . 37
5.4.2 Results . . . 37
5.5 Experiment 4 . . . 38
5.5.1 Settings . . . 38
5.5.2 Results . . . 42
5.5.3 Conclusion . . . 42

Conclusion **43**

Bibliography **44**

List of Figures **47**

List of Tables **48**

A Attachments **50**
A.1 Maps used in 4th experiment . . . 50
A.2 Mapd-visual software used for testing and visualization . . . 50
A.3 Experimental measurements . . . 50
A.4 Simulation.mp4, showing simulation of selected scenarios . . . 50

List of Abbreviations

MAPD: Multiagent pickup and delivery
MAPF: Multiagent path finding
TP: Token Passing algorithm
TPTS: Token passing tasks swapping algorithm
C-A^*: Central algorithm with A^* local search
CBS: Constraint based search
C-CBS: Central algorithm with CBS local search
CT: Constraint tree

Introduction

With the rapid increase in industry productivity in recent years comes the need for fast transportation, efficient storing of goods, and effective control over rapidly changing environments. The positive impact of Multi-agent pickup and delivery (MAPD) research is reflected especially in efficiency and costs. The potential of multi-agent system usage was demonstrated in large warehouses (Wurman et al. [2008]). The authors explained the typical workflow in the warehouse environment and the multi-agent solution's role. They argued that most modern automated warehouses were too expensive and provided ideas for cutting costs. Multi-agent roles in coordinated search and rescue were introduced by Kitano et al. [1999]. The authors proposed an extension of RoboCup-rescue activities, explained the problematics of a hostile environment and suggested significant issues for the rescue to be tackled. Multi-agent systems algorithms even made an appearance in video games (Silver [2005]). The author proposed a faster method of computing the paths of units in strategy games. Recently, intelligent warehouses have been established in Czechia by leading Tech companies such as Mall, Alza, and Amazon. Any of these use multi-agent solutions daily and would greatly benefit from faster or more efficient algorithms. It is desirable to research and utilize multi-agent systems and seize the opportunities they provide. For that reason, we will briefly visit the world of Multi-agent pickup and delivery algorithms.

This thesis aims to compare existing MAPD algorithms and test them in various environments, discuss known techniques, and experiment with the number of agents for some maps. Multi-agent and pickup delivery is a problem from the optimization field, and it belongs to the family of NP-Complete problems, and unlike others, it has not gained popularity until recently, with development in the fields of robotics and optimization.

Especially progress in robotics enabled the potential of MAPD and allowed for further extensive research. A set of agents, orders with pairs of locations, and environment is given, and the goal is to plan the most efficient non-collision paths while delivering all orders.

MAPD is commonly divided into two subproblems: task assignment and path planning. The goal of the task assignment is to decide the next location for the agents. Path planning then safely gets the agents to their designated locations.

1. Problem Definition

1.1 MAPD Formalization

MAPD problem instance is defined by the following input: map, n agents with respective positions, and k orders with respective positions. The map is usually warehouse-like or a replica of some real-world scenario, such as airports, industrial halls, etc. The goal is to deliver all orders by agents on non-conflict paths either with the shortest makespan or finding the most efficient solution by utility function chosen by the user. Commonly used is the cost efficiency function, where all agents and actions have their respective cost, and we seek the solution as low cost as possible. In this thesis, we will be working in a discrete environment. We will denote one tick of the environment as the time step and multiple ticks as the time frame. The other agents, along with map walls, are considered obstacles and must be avoided. Non-delivered orders are not considered obstacles.

Definition 1. An instance of a MAPD problem is triple (G, A, T) where $G = (V, E)$ is a non-oriented graph, in this thesis, also called map. The V set of all vertices is called locations. A is a set of all agents, where agent has A_i unique id, a_s starting location and a_c current location. T is a set of all tasks (orders) of a problem, where τ_i is denoted by triple (τ_t, τ_s, τ_g) where τ_t is the time task τ become available, τ_s is a starting position and τ_g is a goal location of task τ.

Definition 2. Neighbor locations of $v_i \in V$ are all locations v_j in the neighborhood of v.

Definition 3. Agents perform *actions* every time step. The available actions are as follows:
Move: Agent moves to a different neighbor location.
Wait: Agent stays in its previous location.
Pickup: Agent picks up an order
Deliver: Agents unload the order

Every action has its cost and duration. The duration has been set to 1 for simplicity in the experimentation part. The cost will be further discussed in the solution evaluation section. The basic idea is that it makes sense that the cost of movement will be more expensive than the cost of staying. This is motivated by the real world, where the movement of agents burns more fuel, drains more energy from batteries, etc., than staying still. Thus we divided this into two actions. In simple scenarios, the same cost can be assumed over all actions, then Wait actions equal Move actions.

Definition 4. By assigning task τ to the agent, it is meant that the agent's immediate goal location becomes the τ_s and once the agent reaches it, the new goal is τ_g.

Definition 5. By finishing task τ is understood that agent successfully moved to τ_s, picked up the order, moved to τ_g, and delivered the order.

Definition 6. The agent is *occupied* if it has been assigned a task that it is executing. Otherwise, the agent is *idle*. Once the agent finishes the task, it becomes *idle*. Once it is assigned a task, it becomes *occupied*.

Definition 7. The location of agent a in time t corresponds to its current location in time t. It is denoted as a_{c_t}

Definition 8. Vertex (location) collision is a situation where two agents $a \neq b$ share their current location at the time t. Formally $\exists a \neq b \exists t$ such that $a_{c_t} == b_{c_t}$.

Definition 9. Edge collision is a situation, where two agents $a \neq b$ share an edge uv simultaneously. Formally let a, b be agents and $a_{c_t}, b_{c'_t}$ be their current locations at time t. Then edge conflict occurs when at $t+1$, their respective locations become $a_{c'_{t+1}}, b_{c_{t+1}}$.

By creating non-colliding paths, we understand that no edge nor location collision takes place during the plan. This is a neccesseary condition for validity of a plan.

Definition 10. Path of an agent is $l_{1,t}, l_{2,t+1} \dots l_{i,t+i}$ sequence of locations the agents have been assigned with their respective time frames.

Definition 11. The plan is a sequence of actions of all agents.

1.1.1 Well formed MAPD Instance

According to Ma et al. [2017], the MAPD instance is well-formed iff it satisfies the following:

1. Number of tasks is finite
2. There are no fewer non-task endpoints than agents, where by task endpoint, we understand a designated location in which the idle agents may wait.
3. For any two endpoints, there exists a path such that it contains no other endpoint

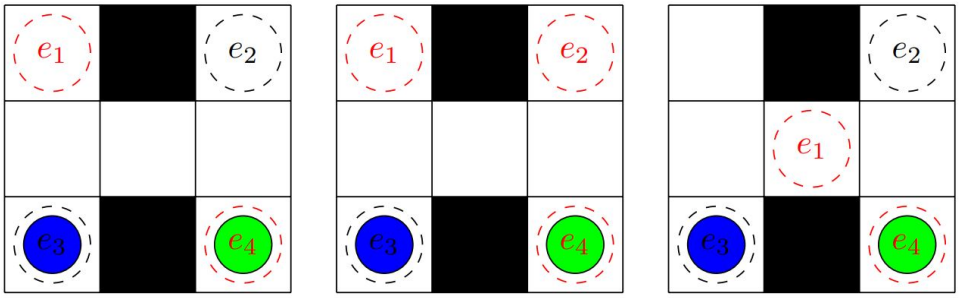


Figure 1.1: Ma et al. [2017] Example MAPD instances

In the Fig. 1.1, black cells are blocked, Blue and green circles are initial locations, red-dashed circles are task endpoints, and black-dashed circles are non-task endpoints.
The rightmost picture shows an example of a non-well-formed instance. All paths from e3 to a2 contain e1, which violates the third condition. The middle one violets the second property as there is one non-task endpoint and two agents. The leftmost one is an example of a well-formed instance.

The MAPD algorithms are constructed in such a way that they solve every well-formed instance. These instances, however, rarely reflect real-world scenarios. Therefore, algorithms are modified to tackle non-well-formed instances as well, in that case, losing the property of the guaranteed solution. All measurements in the experimental section are conducted solely on non-well-formed instances.

1.2 Additional settings

1.2.1 Deadline for orders

Every task can be assigned a deadline. Formally new variable is introduced $\tau_d \; \forall \tau$ which denotes the deadline. Consequentially, every task must be finished before the deadline. In case of missing the deadline τ_d for some task τ, the solution is either invalid or penalized based on the difference between the time of finishing the task and its deadline.

1.2.2 Capacities

Orders may be assigned volume or weight and agent's capacities. Formally new constraints are introduced $\tau_w \; \forall \tau$, which denotes the weight of a task τ. $A_c \; \forall A$, which denotes the capacities for each agent. This further extends the basic MAPD instance because the agent may carry multiple orders at once. The goal remains the same, but the agents must not exceed their respective capacity limit while processing the orders at any point in time during the solution.

1.2.3 Multi-goal tasks

This MAPD extension is called MGMAPD. The tasks have 1 to n (where n is at most the number of locations) goal locations. Order is denoted by triple (τ_t, τ_s, τ_G) where τ_t is the time task τ becomes available, τ_s is a starting position and τ_G is a set of goal locations for task τ. The task is finished if the order is delivered to either of these locations. MGMAPD is detailly described by Xu et al. [2022].

1.2.4 Task ordering

In the original MAPD instance, orders may be processed in arbitrary order. In many real-world scenarios, however, the order in which the tasks are processed is given and must be followed. For example, an agent may manipulate some order only after a different order has been completed, etc. The generalized version with precedence constraints was described by Zhang et al. [2022] for Multiagent path finding (MAPF) problem.

1.2.5 Agent charging

Real warehouses often force agents to charge their batteries or fill up the fuel in designated locations. This must be done either at predefined times or when the agent runs low on power. This MAPD extension introduces additional conditions and constraints for the plan based on agent charging.

1.2.6 Offline

The MAPD problem is considered an online problem. In some cases, solving its offline version holds important information or even practical uses. Liu et al. [2019]. By offline settings, it is understood that we know all the orders with their respective start times and initial and goal locations in advance, and we can include this information in our computation.

1.3 MAPF

Multiagent path finding is a subproblem of MAPD. The input consists of pairs of locations , and the goal is to find non-collision paths for agents to move to their respective target locations Stern et al. [2019]. Formally the classical MAPF is defined as follows. Input is triple $(G = (V, E), s, t)$ where G is undirected graph
$s : [1..k] \to V$ maps agents to source vertex
$t : [1..k] \to V$ maps agents to a target vertex.
Time is assumed to be discrete. Each time step agent might perform an action, wait, or move (the definition of these actions is identical to these of MAPD). The goal is again to find a valid Plan with non-collision paths.

1.4 Solving methods

The MAPD problem instance is solved by solving multiple subsequent MAPF instances. The solution method consists of two steps. Task assignment and path planning. Task assignment returns the pairs of the agent with its initial and goal location, thus creating a MAPF instance. This problem is then treated as a MAPF problem and solved as such. The environment is updated, and this process repeats as long as the simulation runs. We will briefly introduce the general methods. Concrete algorithms will be described in the Related Work chapter.

1.4.1 Task assignment

The task assignment problem is computing the matching of agents and their respective goal positions for the following MAPF subproblem. The locations are usually the start locations of some task, the end location of a task, or the result of a wait function call. The assignment should maximize the utility function as much as possible. For example, if the utility function is service time (difference between optimal time of delivery of the orders and the real one), then we want to assign locations such that the sum of their distances from their respective agent's current locations is the lowest possible.
Let A_{i_al} denote location assigned to the agent i then the aim is to minimize

$$\sum_{i}^{k} Dist(A_{i_al}, A_{i_c}) \tag{1.1}$$

1.4.2 Path planning

There are three main general approaches to solving the lifelong MAPF instance. All of them are utilized in different algorithms and solving techniques described in Li et al. [2021]

Complete non-collision paths

All paths for all agents are computed at once. This tends to be computationally more challenging, or the solution tends to be worse if we use some heuristic to speed up the computation. The scaling is poor.

Decompozing MAPF

Lifelong MAPF is decomposed into individual MAPF problems, and the paths of the agents are replanned every time step. Search from previous time steps can be utilized but fails to scale well in dense scenarios, as most of the paths must be recomputed.

Decompozing MAPF-v2

A method similar to the second one, but only the paths of agents, which delivered the order in $t-1$ are recomputed in t-time step. The scalability of this method is stable; however, other unpleasant drawbacks are typical, such as incompleteness.

1.5 Example

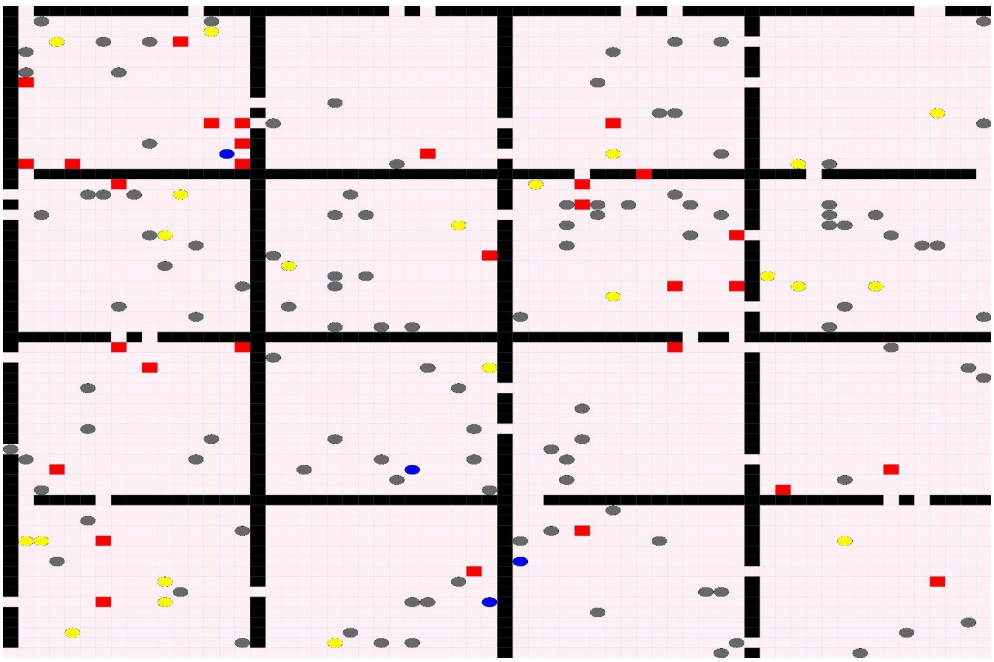


Figure 1.2: The figure shows MAPD instance from mapd-visual software

The above figure shows an example MAPD scenario with 30 agents and 200 orders. Agents are denoted by red tiles, active orders by blue circles, accessible locations are white tiles, and wall tiles are black. Inactive orders are grey (orders that have not yet been revealed), and yellow circles are goal locations of orders which are already loaded on agents. The initial location of all agents must be different, as well as the initial position of all orders must not be equal. Each order has an assigned target position, which is a completely arbitrary accessible tile.

2. Related work

While the MAPD problem drew attention quite recently, MAPF has been studied for a while. Provided that most modern MAPD algorithms use solving MAPF instances as subproblems, We list related work on MAPF algorithms, as well as research conducted on MAPD.

2.1 MAPF research

Algorithms for MAPF instances can be divided into two basic categories. Optimal and bounded. While the optimal solvers return solutions with optimal cost, their runtime is usually rather inefficient. On the other hand, Bounded suboptimal solvers return solutions with a maximum cost of k times the optimum cost, but their runtime performance far exceeds that of optimal solvers. In the real world, computational effectiveness is a trade-off for the cost of a solution and vice versa. Therefore excessive research has been conducted on both classes. They can both be subdivided into the same categories based on algorithms used by their solvers Stern [2019].

2.1.1 Hierarchical cooperative A*

The most intuitive of the listed methods is the extension of well known A* (Silver [2005]). In some given order, the agents find their respective shortest available paths. Once the path is found, the information about it is passed to other agents and the locations became unavailable for other agents. The upside is its simplicity and computational speed. Downsize is that the scaling in the number of agents is quite poor. In bigger scenarios, the problem often becomes unfeasible if solved by HCA*.
Improvement suggested WHCA* (windowed Hierarchical cooperative A*). The cooperative search is limited to a fixed depth specified by the current window. Each agent searches for a partial route to its destination and then begins following the route. At regular intervals (e.g. when an agent is halfway through its partial route) the window is shifted forwards and a new partial route is computed. Silver [2005]
Further research suggested CO-WHCA* (conflict-oriented windowed Hierarchical cooperative A*) Bnaya and Felner [2014]. Authors improve the reservation system of WHCA* resulting in a great decrease of runtime.

2.1.2 Increasing cost tree search

This algorithm was proposed by Sharon et al. [2013] . The idea of the algorithm is to traverse the search tree, however, unlike A*-based algorithms, the search is not driven by heuristic. It divides the search into two levels.
High-level search, which searches for a minimum cost solution, in the space of combinations of agents.
Low-lever search, which searches for a valid solution under the cost constraints. The low-level search can be viewed as a goal test for the High-level search.

Multiple pruning techniques for the lower-level search were later introduced in Sharon et al. [2011], leading to a speed-up of the goal test.

2.1.3 Conflict based search

This algorithm was proposed by Sharon et al. [2015]. There are two levels of search in CBS: high level and low level.
HighLevel: In the high level, we construct a constraint binary tree (CT) The low-level search is used to evaluate the node, then enumerate them in sorted order by price. Once we reach a valid, so-called goal node, we return it as a solution.
CBS algorithm is thoroughly described in the third chapter.
The Improved version is introduced in Boyarski et al. [2015]. Multiple improvements are suggested, such as search restarts, smart splitting, or conflict bypassing.

2.1.4 Constraint programming

The MAPF problem can be encoded into a set of constraints, for example, an agent occupies exactly one location at every time, no two agents occupy the same vertex at any time, etc. The modulation of these constraints into logic-based Picat language was presented by Barták et al. [2017]. Once the problem is encoded, different variants of SAT and MIP are utilized to obtain a solution.

2.1.5 Learning based algorithms

With its gained popularity, deep learning has crawled its way to previously purely symbolic AI problems. Abreu [2022] proposes a supervised learning approach for solving MAPF instances using smaller, less costly model for training. In Huang et al. [2022], learning is being used for guiding the LNS search and proved to be quite effective in certain scenarios.

2.2 MAPD research

The following section containts non exhaustive list of MAPD related research.

2.2.1 Offline settings

In fully offline settings, the main approach became solving life-long instances of MAPF problems. In Liu et al. [2019], multiple techniques are introduced.
Solving special TSP instance with constraints and penalty function. The solution appears to create a very effective task sequence assignment. **Prioritize Path planning.** Based on priority, agents plan their paths using A^* and knowledge about other agents' already planned paths. **TA-Prioritized path planning** Also takes into account idle agents, parking, and mock locations.
The second proposed algorithm,**Hybrid path planning** uses the ICBS algorithm to plan agents from pickup locations to goal locations, whereas, for free agents, it uses a min-cost max-flow polynomial algorithm in every time step.
Nguyen et al. [2017] works with the generalized target assignment and pathfinding problem. They address the limitation of TAPF by allowing an unequal number

of tasks and agents, assigning deadlines to each task, and introducing a checkpoint path, where every task is composed of a path of checkpoints that must be completed prior to completing the task. The problem is modeled as **Answer set programming**, however, solutions for only small-size scenarios are presented.

2.2.2 Online settings

The online settings for MAPD problems have rapidly become the default setting of MAPD problem instances. Exhausting research has been driven by real-world applications in recent years. The two different algorithms classes were initially introduced in Ma et al. [2017]. They were thoroughly studied and the algorithms improved bit by bit in follow up research. Both approaches, central and decoupled, will be further examined in the third chapter.

Decoupled

In this class, agents compute the paths themselves based on system knowledge about other agents. One agent computes one path at a time to ensure conflict-free paths.

State-of-the-art algorithm Token passing (TP) and Token passing with task swapping (TPTS) have been proposed by Ma et al. [2017]. Follow-up research on TP has been conducted by Nie et al. [2020]. The authors address the issues with the original definition of TP, such as problems with feasibility in some scenarios, and suggest solutions by introducing ITP (Improved token passing). The procedure for executing mock tasks is added to unblock some of the actual tasks and overall improve the flow of the algorithm.

Yamauchi et al. [2022] extends TP by adding standby-based deadlock avoidance for improved transportation efficiency. So-called standby nodes are locations, where agents wait before they continue with their task execution. These nodes can be dynamically selected. Pathfinding for the agent first decides whether the next location is a standby node or an actual task-goal location. Then proceeds with the slightly modified path-finding method of TP.

The MLA* was invented by Grenouilleau et al. [2021]. This algorithm extends the classic A* algorithm by computing a path through a set of ordered goal locations. The advantage over sequential A* is that additional constraints for the target initial and locations can be omitted. H-value-based task assignment is also introduced.

Central

In a centralized approach, the system itself computes agents' respective paths. It is done so in every time frame based on the current state Ma et al. [2017]. Using $A*$ as a pathfinding method in this manner seemed to be inefficient, therefore, CBS approaches has been introduced as a pathfinding method. The task assignment uses the Hungarian method for maximum matching.

The centralized approach proves to have far worse scaling, thus it lacks the popularity of TP.

Other

The authors Chen et al. [2021] explore a technique, which combines the task assignment step with the path planning step. The idea is to improve the task assignment by the knowledge of the cost of the paths. The algorithms MCA (Marginal cost-based task assignment) and RMCA (Regret marginal cost-based

task assignment) are introduced. The authors also demonstrate the extension of MAPD with capacities of the tasks and agents carrying multiple orders.

2.2.3 Semi-online settings

The motivation for researching this setting is quite clear. In many real-world scenarios, we have certain information about upcoming orders, but at the same time, other orders might be issued. The main algorithm of this type is Look ahead horizons, where the paths of the agents are recomputed every h time frame based on partial knowledge. This algorithm was proposed in Xu et al. [2022].

3. Algorithm description

This chapter is dedicated to detail describing the algorithms and techniques used in solving MAPD and MAPF problem instances.

3.1 Token Passing

This is an algorithm similar to cooperative A*. It was proposed by Ma et al. [2017]. For its simplicity, this is the only algorithm, that scales to thousands of orders and agents. It belongs to the Decoupled algorithms family.
Firstly, Agents are assigned states (idle, occupied). In every time step, in a selected order, agents are iterated. If the state of an agent is idle, it is sent a token and assigned an order. Orders are selected from a list of already visible but not yet assigned orders of the token. The agent usually receives the order with the closest pickup location. This depends on assigning function. Then the path for the agent is calculated. The token holds information about all the possible paths. Certain locations and edges are already blocked for other agents at different times, so they can not be used. To find an optimal path, A* is utilized with extra conditions on opening new locations based on their time occupancy. If A* selects position D in time t, and D has 3 neighbors: F, G, H where F is occupied at $t+1$, G at $t+2$, H at $t+3$. From D we cannot open location F as we would arrive at time $t+1$. Slightly modified but a similar idea is for edges. Checking for occupied locations is not sufficient enough, in that case, agents could swap their respective neighbor positions in one time step, but in reality, it would lead to a collision. When a path is found, it is assigned to the agent and blocked within the token, meaning the token will now hold information about this path and block location and edges at corresponding times.
If the path was not found, meaning given the current state, the agent can not arrive at the pickup location, there are two choices, either we try to compute the path to the second-best order or we run the waiting function on the agent.
If no order is selected (either there is no order to deliver or no paths have been found), the waiting function is called on the agent with the state "idle".
After that, the system time ticks, and the token is updated. Each agent has already been assigned movement, so the time in the token is updated, agents proceed with their respective assigned actions for this time step, and their current locations and states are updated. If the agent delivers an order, its state becomes idle and it will be assigned a new one in the next iteration.
This algorithm will terminate and finish all orders. One-by-one access to the token ensures collision-free paths. Once the path is blocked for a certain agent no other agent can use it and interrupt it. For the sake of simplicity, we assume, we will assume agents are immune to malfunctioning. We also assume all orders are reachable from anywhere on the map. This is a necessary condition that ensures solvability.
Waiting function: If the agent has to halt for there is no order or path to its assigned order, the target mock location is assigned instead, and the path to this location is computed. The easiest is to stay in the same location unless it is occupied next step. In that case, find a neighbor location that is not occupied

in the next step. The edge occupancy must be taken into account as well. By this logic, collision-free waiting is ensured. Different mock locations can be areas not covered with agents, meaning no occupied agent has a target location there and no idle agent is nearby. The advantage is that if a new order comes to this section of a map, it can be served faster. A disadvantage might be the increase in the cost function of the solution.

Order heuristic: We can select the order in which we allow agents access to the token. This can be randomized as the simplest solution. Other options are a user-based priority. If we want to use some agents more than others, assigning higher priority allows the agent to complete more orders. A different approach is distance based, we compute the distance from all agents to their closest location and assign priority in descending order, meaning the agent that has the longest distance will be given priority in path selection. The distance can be calculated fast. Precomputed table with distance info for all locations is available, thus finding the order of agents is O(1).

Algorithm 3.1 TP

```
token ← InitializeToken(Agents, Orders, Locations)
while true do
    Add new orders to token.Orders
    for all a ∈ Agents do
        if a.state == idle then
            FindBestOrder(a, token)
            if a has assigned order then
                PlanPath(a, o, t)
                Remove o from token.Orders
            else
                Wait(a, t)
            end if
        end if
    end for
    Agents execute one time step and update state
    Time in token increases by one
end while

function FindBestOrder(Agent a, Token t)
    minDist ← int.MaxValue
    bestOrder ← null
    for all oint.Orders do
        if d ← dist(a.CurrLoc, o.OriginLoc) < minDist then
            minDist ← d, bestOrder ← o
        end if
    end for
    return bestOrder
end function

function PlanPath(Agent a, Order o, Token t)
    path ← findPath(a, o, t.Locations)
    token.UpdateLocations(path)
    a.Path ← path
end function
```

3.2 Token Passing Task Swapping

Next decoupled algorithm proposed by Ma et al. [2017]. The idea is similiar to the TP algorithm with Task Swapping extension. While in TP, once an order is assigned to the agent it cannot be reassigned to a different one, even if it were beneficial for the overall performance. This is where Task swapping comes into play. When computing the optimal order of idle agents, we will assume not only unassigned orders but also orders that have not been picked up yet. First, we will find the best order amongst the unassigned ones. Then in ascending order by distance, we will look at the assigned but not picked up. If the distance to the unassigned one is closer than any of the assigned ones, we immediately return it. Otherwise, we have to check the overall behavior.

Assume we are finding the best order for agent A_i. The optimal order is O_i, currently assigned to the agent A_j but not yet picked up. The algorithm will save the state of the token, and unassign order from agent A_j, so that the remaining path blockings are cleared from the token, we calculate the path for agent A_i to order O_i, block it within the token and immediately send the token to the agent A_j. In case agent A_j finds an order O_j, such that it was worth it based on distance, we make the swap and save a new token. A_j can repeat the process of A_i and find order assigned to agent A_k etc. This process eventually terminates, thanks to the triangular inequality. If even one of the steps fails though, one of the agents cannot find an order, or the trade becomes overall ineffective, the changes are rolled out, we take back the saved token, and agent A continues with the second-best order.
This process is very computationally demanding. Even though the result is guaranteed to exceed the performance of TP, the time complexity is much worse and the algorithm is not scalable for larger scenarios.
Distance-based heuristic: Given the assumption we have access to the precomputed distance tables, we can use a local distance heuristic instead of a full path search. When evaluating, whether a task swap is efficient in a given moment, instead of recalculating all the paths for all agents to their new orders, we assume only distance. We search for all agents the orders in ascending order and we only calculate the distance save/gain in theory based on real distance. This saves a nontrivial amount of computational time, however, creates space for nonefficient behavior.
Let A_i be the agent, and let O_i be his optimal order. We compute raw distance, let's call it $rd(A, O)$. Then we compute distance under the token, meaning the fastest way to travel from A_i to $O_i.pickupLocation$ given path blockages td(A,O). It holds that rd(A,O) $\geq$ td(A,O). For low agent density scenarios, the difference will be negligible. However, in high-density agent scenarios, the difference can cause the overall performance to drop below the one of TP.
This heuristic is very effective for high-order density and lower agent density scenarios. It scales much better than the original TPTS and mostly outperforms the TP algorithm, as demonstrated in the experimental chapter.

Algorithm 3.2 TPTS

```
token ← InitializeToken(Agents, Orders, Locations)
while true do
    Add new orders to token.Orders
    for all a ∈ Agents do
        if a.state == idle then
            Order o ← FindBestOrder(a, token)
            if o ≠ null then
                PlanPath(a, o, t)
                Remove o from token.Orders
            else
                Wait(a, t)
            end if
        end if
    end for
    Agents execute one time step and update state
    Time in token increases by one
end while


function FindBestOrder(Agent a, Token t)
    Orders ← t.Orders
    while Orders! = null do
        bestOrder ← null
        minDist ← int.MaxValue
        for all o ∈ Orders do
            if d ← dist(a.CurrLoc, o.OriginLoc) < minDist then
                minDist ← d, bestOrder ← o
            end if
        end for
        Remove o from Orders
        if o has no assigned agent a_i then
            Assign o to A
            PlanPath(a, o, t)
            return true
        else
            Save the token state
            Unassign o from a_i
            Remove the a_i's path in token
            PlanPath(a,o,t), assign o to a
            if new dist to o is shorter a FindBestOrdera_i, t == true then
                return true
            else
                Revert changes to saved token state
            end if
        end if
        return bestOrder
    end while
end function


function PlanPath(Agent a, Order o, Token t)
    path ← findPath(a, o, t.Locations)
    token.UpdateLocations(path)
    a.Path ← path
end function
```

3.3 CENTRAL

This algorithm was proposed by Ma et al. [2017]. Unlike in the Token Passing algorithm and its derivates, in CENTRAL, as the name suggests, agents do not calculate their respective paths based on system knowledge, they are assigned one.

CENTRAL recomputes all the paths of all agents in every time step, which leads to better performance of the algorithm, yet much worse computational performance and ineffective scaling. Because the amount of shared information amongst the agent and systems is the largest among already mentioned algorithms, the usual expectation is that it will outperform them. However, due to frequent recomputation of all paths, for larger environments, CENTRAL does not manage to compute the agent actions on time thus it is not suitable for bigger warehouses or other complex MAPD scenarios.

Firstly, CENTRAL assigns tasks to each agent. This can be either the pickup location of some order, the target location of the order in case the agent is loaded or mock position in case we need to get the agent out of the way.

Assignments are computed using Hungarian algorithm.

3.3.1 Hungarian Algortihm

This optimization method was developed by Kuhn [1955]. By original idea, we are given a matrix of n workers and n jobs. Element a_{ij} denotes the cost of job j done by worker i. The goal is to minimize the cost and finish all the jobs. The Hungarian method outputs minimal perfect matching in polynomial time, $O(n^3)$ to be exact.
We discard all agents that are currently loaded with orders and assign them their target location. CENTRAL usually does not work with the same number of available tasks and agents, thus we have to create either mock agents or tasks with arbitrarily large values in the cost matrix. As a result, they will not be prioritized, Hungarian method outputs minimal max matching in the matrix without them and we will omit them in result processing. In case the agent is assigned a mock location, based on settings, the wait function is called. In case the task is assigned a mock agent, we ignore it and try to reassign the Task in the next time step.

3.3.2 Path computing

Each agent has been assigned a task with its location. Based on priority, CENTRAL computes the fastest path for the agents one by one using CBS.

3.3.3 CBS

This algorithm was proposed by Sharon et al. [2015]. The state space spanned by A* in MAPF is exponential in k (the number of agents). By contrast, in a single-agent pathfinding problem, k = 1, and the state space is only linear in the graph size. CBS solves MAPF by decomposing it into a large number of constrained single-agent pathfinding problems.
There are two levels of search in CBS: high level and low level.
High level: In the high level, we construct a constraint binary tree (CT) where each node has:
Agents: List of all agents
Constraints: List of all constraints for given agent.
Constraint format: Constraint is a location-time or edge-timestep pair. It prevents the agent from entering said location (edge) in said timestep.
Price: Evaluation of a node. Usually denoted by the sum of all costs of all paths of agents.
Low level: The low-level search is used to evaluate the node, then enumerate them in sorted order by price. Once we reach a valid, so-called goal node, we return it as a solution.
Low-level search computes the fastest path for all agents given their constraints independently using A*. The second part is the evaluation of a node. This simply checks for any conflicts, edge, and vertex, between agents given their plans. If no conflict is found, this node is a goal node. Because the nodes are enumerated in sorted order, the first time a goal-node is found, it is always the best solution.
If a conflict is found between two agents a_i and a_j, it is returned. The High-level search then splits the node into 2 children, where in one child, the constraint for a_i is added, and in the other, the constraint for a_j is added. The search then continues.

In case of a conflict given more than two agents, there are generally options. Either the node is split into n children, where n is the number of conflicting agents or the first two agents in conflict generate the children of a CT node, and the remaining conflicting agents will be resolved in the lower branch of a CT. Both are valid and equally fast options, so this choice has little to no effect on the performance.

3.3.4 Update

At the end of the time step, all agents proceed with their respective assigned actions. In case they loaded/unloaded an order, their status is updated. Unless the stop message has been passed down to the Algorithm, the cycle repeats.

Algorithm 3.3 CENTRAL

```
while true do
    Add new orders to system
    TaskAssignments(system)
    CBS(system)
    UpdateSystem(system)
end while

function CBS(system)
    Initialize root
    Add root to nodes
    while true do
        for all node ∈ nodes do
            Evaluate node cost
        end for
        nodes ← nodes.SortByCost
        for all node ∈ nodes do
            conflict ← Validate(node)
            if conflict is empy then
                return node.Plan
            else
                Create two new constraints from conflict
                Split node into two, assign each one constraint
                Add children of node to nodes, remove node from nodes
            end if
        end for
    end while
end function


function TaskAssignments(system)
    Create mock agents/tasks
    M ← distances from agents to tasks
    result ← Hungarian(M)
    Assign Task to agents based on results
end function


function UpdateSystem(System s)
    for all Agent a ∈ s  do
        Execute action
        UpdateStatus
    end for
end function
```

3.3.5 Prunning

So far, we have treated idle agents and active agents alike. In certain scenarios, however, we would like to prioritize the active agents to achieve lower makespan and overall performance. The downside might be a slight increase in cost.
The basic branching idea is simple. If a conflict is detected by high-level search where one agent is idle and one active, we create just one child CT node where we add the constraint of the idle agent.
By using this method, in certain time steps, a feasible solution does not exist and we lose the main strength of CBS. To prevent this, when creating a child CT node for the idle agent, we also create a node for the active agent, but instead of

a CT, we insert it into a list and we do not explore it further. If the CT empties without finding a solution, we build a new tree using these saved nodes with lower priority. If we were to use the secondary tree every time step, the runtime would worsen, but for a reasonable number of agents, this happens extremely rarely, and the overall performance increases as demonstrated in the experimental chapter (chap. 5.4).

3.4 Possible pitfalls

Each of the mentioned algorithms has its pitfalls, the worst impact have the deadlocks. Deadlock is any situation in which no agent can proceed because each waits for another agent to move. Deadlock can be resolved only by brute force, meaning some entity must order some agent to move and free a way for the others. On maps containing the dead-end, the classical MAPD algorithms may fall into deadlock even with smart wait function modification. To prevent deadlock, a semafor-like entity needs to be instantiated and control the traffic flow through the dead-end segment. A possible solution is using the mentioned standby deadlock avoidance (Yamauchi et al. [2022]).
Assuming dead-end free maps, as all of the experiments conducted are on the maps, where every location has at least 2 neighbors, We will explain how each algorithm can fall into a deadlock state and how can we prevent it. The mentioned methods are inspired by general deadlock avoiding and solving techniques and modified for our experimental environment.

3.4.1 TP and TPTS

For both of these algorithms, deadlock can happen only if the agent calls a wait function and zero locations are returned. It means, that agent cannot stay in his current position nor it can move to a different one, as all of them are occupied (or edges to them) in the next time step. We can try to prevent this situation with a conflict heat map. We can run the simulation in our designated environment, measure the collision heat map, ie score locations based on collision (Zhao et al. [2023]) and we can set TP/TPSP to preferably the fewer conflict locations. This, however, does not prevent deadlock completely. Upon stumbling on deadlock, we have to compute paths of involved (and possibly other) agents again. We take the paths of these agents, unblock their paths in the token, permutate their order (ideally we want to begin with the agent that discovered the deadlock) and run the updates on them again. This will almost likely solve the issue, if not, another agent must have caused the deadlock again. We can either repeat this process starting with all the problem agents or we can recount all occupied agents. This happens very rarely so it does not affect the computational time of these algorithms.

3.4.2 CENTRAL-A*

This case is very similar as A* algorithm is used for path search. The advantage of CENTRAL algorithm is that when deadlock happens, we do not need to unblock paths in the system as only one step every time frame is computed. We

simply repeat the computation for this time step with permutated order, which is computed in the same fashion as above.

3.4.3 CENTRAL-CBS

C-CBS should be deadlock-free (assuming a solvable instance). The reason is that CBS always return feasible solution, ie non-conflict paths. Two agents' possible deadlock is resulted automatically, as creating subnodes with constraints for agents gives one of the agents augmented priority. For multiple agents, it may rarely happen, that one of the agents cannot under any scenario find a valid path to its task. The wait function is called and in the next frame, the attempt is repeated. In the worst case, this agent has to wait for all other agents to finish their tasks and become idle. For the Idle-Active agent possible deadlock, an elegant solution exists. If the wait function assigns the idle agent its current position, the CBS only finds a path of length 1, its location. This can, in fact, lead to deadlock or loop.
Instead, a sequence of wait function calls is assigned, where the number of calls is the length of the longest path of any active agent computed by low-level search. The conflict on Xth frame will be recognized, constrained created, and a feasible path of length n computed for all agents. This transforms the problem into original CBS (Sharon et al. [2015])and the problem with deadlock is voided.
Alternatively, the active agent prioritizing over idle agents leads to deadlock avoidance.

4. Plan Validation

The work on plan validation for MAPD was introduced in Almagor and Lahijanian [2020].
With the increasing trend of machine learning, solutions to various problems are somewhat black boxes. In the area of multi-agent systems, it is vital to be able to verify incorrect and potentially dangerous solutions. Even though we can programmatically validate any plan, for the sake of safety, the humans (for example, warehouse controllers) should be able to verify them as well. There are multiple approaches to the plan validation, also called "Solution explanation." The crucial point is the ability to verify the non-conflict nature of the paths. The validation process must be effective; the warehouse or other multi-agent environment is rapidly changing, and identifying problems must be possible within a few moments.
The already proposed technique slices the plan into specific time frames containing only non-conflicting paths. These segments must be as large as possible to speed up the validation process. The agents can be visualized by color lines with the location either by grid or actual map used for the plan. In the case of a small number of agents, different colors can be used to visualize them individually and for easier identification. This method does not scale well to a larger amount of agents. Therefore it is recommended to implement a mono-color scheme where we lower the ability to identify the agents, but the overall picture remains easily readable. Some other identification might be implemented, for example, small digits next to the start/end of each agent's paths.

4.1 Validation algorithm

The most straightforward approach is the most direct one. We iterate through all paths of all agents one by one in each time frame. We remember every visited location, and once we try to visit an already seen location, we recognize the positions of agents, index on search in every path, and return the segment. Then the search continues at the same position. We search until all paths have been covered. This approach will end after viewing every step of each agent exactly once, so in time it's linear in the length of a plan. The memory needed for such an algorithm is linear in the number of locations because once we try to load one location twice, the process repeats.

4.2 Properties

Every plan is decomposable in n non-colliding time frames. The proof of this is trivial if we assume a valid (which implies non-colliding) plan. From the beginning to the end of a plan, there is no collision between any agents. Therefore we can certainly take all time steps and present them as a valid non-colliding time frame. For practical use, the number of frames should be as low as possible for quick validation. However, once the plan is computed, the exact non-colliding time frames are determined and can not be changed. So the question arises, can we

find a valid plan such that it is decomposable into m time frames, given MAPD problem instance? The answer is yes, but the hardness of a problem increases significantly. This problem is arguably harder than regular MAPD; therefore, it is at least NP-HARD. The solution contains finding disjoint paths in a graph, which is also NP-HARD. Thus this new problem is also NP-COMPLETE, but much harder in a sense (Almagor and Lahijanian [2020]).

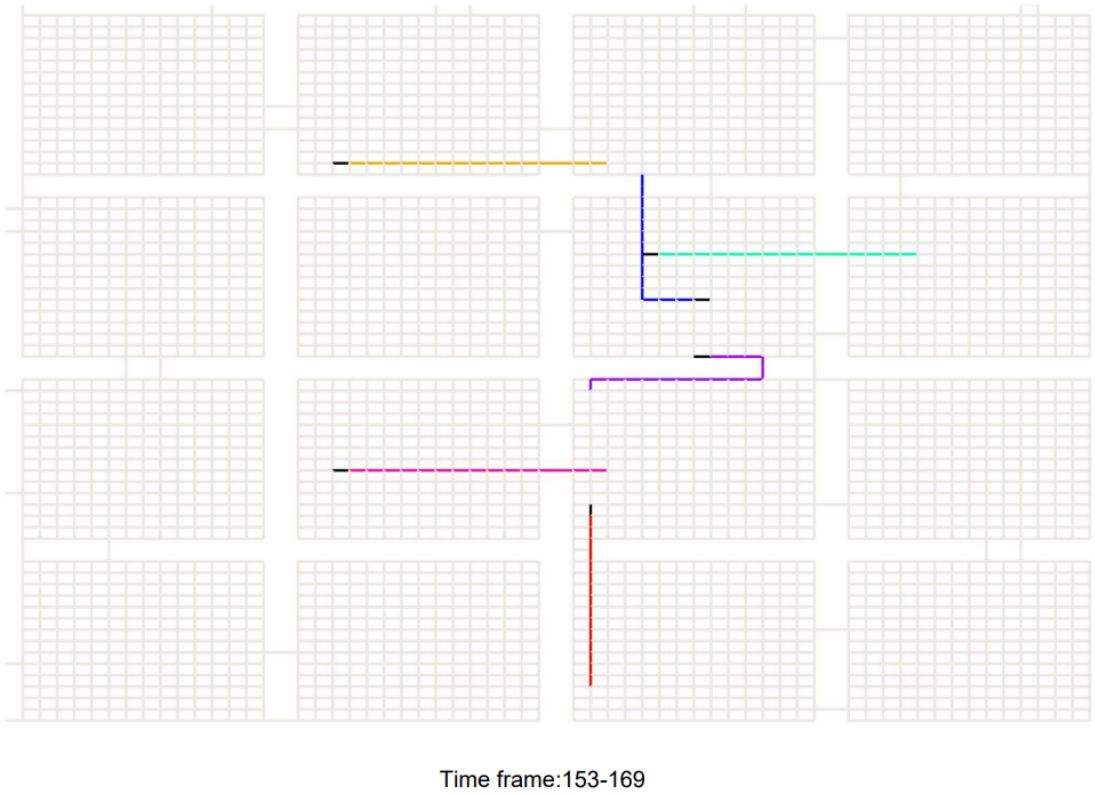


Figure 4.1: The figure shows plan segment visually explained in MAPD-visual software.

5. Experiments

In this section, we describe conducted experiments and share the results. We have implemented multiple MAPD algorithms and their version with various heuristics and conducted various tests and analyses based on thesis guidelines.

5.1 Experiment descriptions

Implementation of algorithms is in C# language along with WPF interactive application. Due to the usage of objects and suboptimal data structures, some experiment runs are slower, than if implemented by language as PICAT.
The basic MAPD problem is expanded by actions of pickup, delivery, and wait. This extension was motivated by the idea of a generalization of the MAPD problem from the cost point of view. In some of the experiments, the cost of the solutions will be compared as an additional tool of measurement to runtime and makespan. It also has a real-life motivation, where pickup and deliver actions both have their respective duration, as well as wait action is cheaper than the movement. Nevertheless, Actions pickup and delivery can be omitted by setting their duration to 0. Similarly, action wait can be turned into action move by setting its cost to 1, then we create a basic instance of the mapd problem. Because we do not directly compare results to the ones of other thesis/articles on base MAPD problem, this will not create any bias or measurement error. We compare overall results to the ones already mentioned in cited articles based on their ratios, and order based on MS or RT, however, given this extension is used across all our algorithms, produced values do not invalidate the measurements.
Our implementation contains a dictionary for locations and distances for first the "room" map. While some of the articles work with distances as a heuristic based on their location on the grid and ignoring the obstacles, we precompute the exact distance. This is advantageous because it allows for a faster run of path-finding algorithms. It is also motivated by real-world warehouses, where we can divide them into squares and precompute their respective distances quite easily, thus this assumption makes sense in the context of the MAPD problem. For other maps, the Manhattan heuristic is used. Although, with enough memory, it could be just as easily precomputed, leading to better results.
All agents are considered an obstacle. Some MAPF and MAPD instances are unfeasible using only basic algorithms because some agents might finish in a choke area and prevent other agents from finishing their tasks. This implementation always calls the "wait" function on idle agents. This function evaluates whether it is beneficial to move or stay in the same location. Across our experimental measurements, we observed that sending idle agents to the nearest location, which has 3-4 accessible neighbor locations was the most effective makespan-wise. The reason is, that if agents moved only when needed, it can create a "soft deadlock". This means that the optimal path is blocked within the algorithm, and before it is found, the algorithm finds a different, suboptimal one. The difference in timeframes between the optimal and suboptimal path is usually far greater, that sending the agent preemptively to the open space where it can be easily dodged. This also has real-life motivation. idle agents might be sent to some designated

locations, to shorten the distance between them and possible future tasks to minimize makespan. Every map then can have calculated these locations precisely. Another motivation is charging the agents. Based on real warehouses, agents must be charged after a certain amount of steps. These charging stations can be in the middle of the warehouse to further minimize the timespan or somewhere behind the main warehouses for a more practical yet a little less effective cause.

5.1.1 MAPF benchmarks

We tested said algorithms on well-established MAPF benchmarks: Stern et al. [2019]. All maps and scenarios come from this benchmark set. For the transformation from map to MAPD scenario, it had to be modified. For the sake of this thesis, we transformed individual agents with their start and goal positions to the orders. Agents in the context of MAPD can be added to the scenario or generated later. Orders can also be assigned an arrival time with the scenario file itself or later added to the program.
The maps consist of 2 types of tiles, black and white ones. Black tiles symbolize obstacles and are not accessible. No agent nor order has their current (or target) location in the wall tile. All free locations have an edge to their neighbor white location and their neighbors are called "accessible locations" in this thesis.
All scenarios will be from the section "random", meaning that the start and finish locations of all tasks had been generated randomly.

5.2 Experiment 1

5.2.1 Settings

This experiment is heavily inspired by testing TP, TPTS, and CENTRAL in Ma et al. [2017].
We tested 4 algorithms. TP, TPTS, CENTRAL-A^*, and CENTRAL-CBS on the following maps. There are 200 tasks with frequencies of 0.2,0.5,1,2,5,10 per time frame meaning that based on their order, their respective start times are calculated based on frequencies. There will be 10,15,20,25,30 agents for each run. The goal of this experiment is to compare makespan and runtime across main MAPD algorithms. This experiment will be conducted on the following map from mentioned MAPF benchmark:

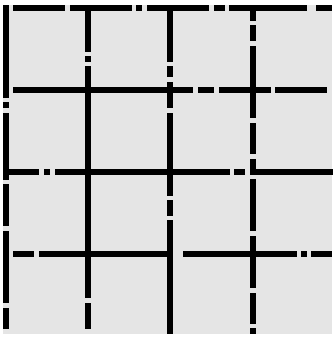


Figure 5.1: The figure shows Room-64-64-16 map

And its ”random” benchmark scenarios modified to MAPD scenarios. Given the information based on related work, the order of algorithms based on makespan should be the following: CENTRAL(CBS, A*), TPTS, TP where CENTRAL should heavily outperform the other 2. However, we expect the same order for runtime per timeframe values, where CENTRAL runtime is significantly slower than the other 2.
We further evaluate the depth of the CBS constrain tree throughout the runs. This measurement should yield clear results, why CBS does not scale well with the increasing number of agents.

5.2.2 Results

The shortcuts for the result tables are as follows:

TF: Task frequency
A: Number of agents
MS: makespan
RT / TF: Runtime per timeframe in milliseconds

Settings		TP		TPTS		C-A*		C-CBS	
TF	A	MS	RT / TF	MS	RT / TF	MS	RT / TF	MS	RT / TF
0.2	10	1762	1.717	1762	2.103	**1723**	4.623	1769	98.312
0.2	15	1363	0.523	1325	0.887	**1280**	8.097	1317	56.560
0.2	20	1225	0.971	1168	1.461	1142	2.453	**1110**	19.902
0.2	25	1166	1.55	1124	1.675	1118	2.315	**1110**	13.776
0.2	30	1190	1.360	1124	1.880	1126	2.313	**1111**	12.649
0.5	10	1696	0.788	1669	0.395	**1613**	2.131	1673	81.643
0.5	15	1238	1.259	1186	0.734	1142	3.092	**1133**	57.505
0.5	20	995	1.170	928	1.018	**920**	2.750	**920**	45.168
0.5	25	838	1.103	801	1.764	783	3.591	**762**	116.83
0.5	30	775	1.384	736	1.927	689	4.393	**631**	371.70
1	10	1712	0.657	1675	0.460	**1598**	2.860	1613	67.755
1	15	1201	1.383	1158	0.645	**1092**	3.097	1172	61.001
1	20	977	1.073	948	1.053	889	2.941	**830**	53.034
1	25	827	1.153	795	1.959	749	3.688	**716**	180.65
1	30	712	1.720	667	2.890	639	4.477	**609**	353.03
2	10	1764	2.194	1673	0.376	**1633**	1.648	1638	42.534
2	15	1199	2.791	1176	0.619	1163	2.428	**1099**	30.884
2	20	966	0.908	899	1.136	864	2.990	**839**	44.241
2	25	850	1.305	777	1.443	743	3.769	**687**	111.09
2	30	741	1.508	683	5.853	647	4.298	**653**	258.38
5	10	1622	2.354	1662	0.471	**1590**	1.848	1656	24.301
5	15	1184	0.950	1129	0.840	**1110**	3.146	1130	48.595
5	20	956	1.054	925	1.039	**861**	3.017	871	51.884
5	25	803	1.598	757	1.863	**728**	3.793	744	259.56
5	30	696	1.982	644	1.684	630	4.758	**618**	1067.4
10	10	1648	2.341	1750	0.432	**1585**	2.488	1616	27.002
10	15	1129	1.024	1147	0.599	1158	4.846	**1100**	44.017
10	20	896	0.940	873	1.342	873	3.141	**827**	33.702
10	25	786	1.350	755	1.738	727	4.039	**691**	316.51
10	30	679	2.106	666	2.226	605	4.804	**567**	733.63

Table 5.1: Experiment 1 results.

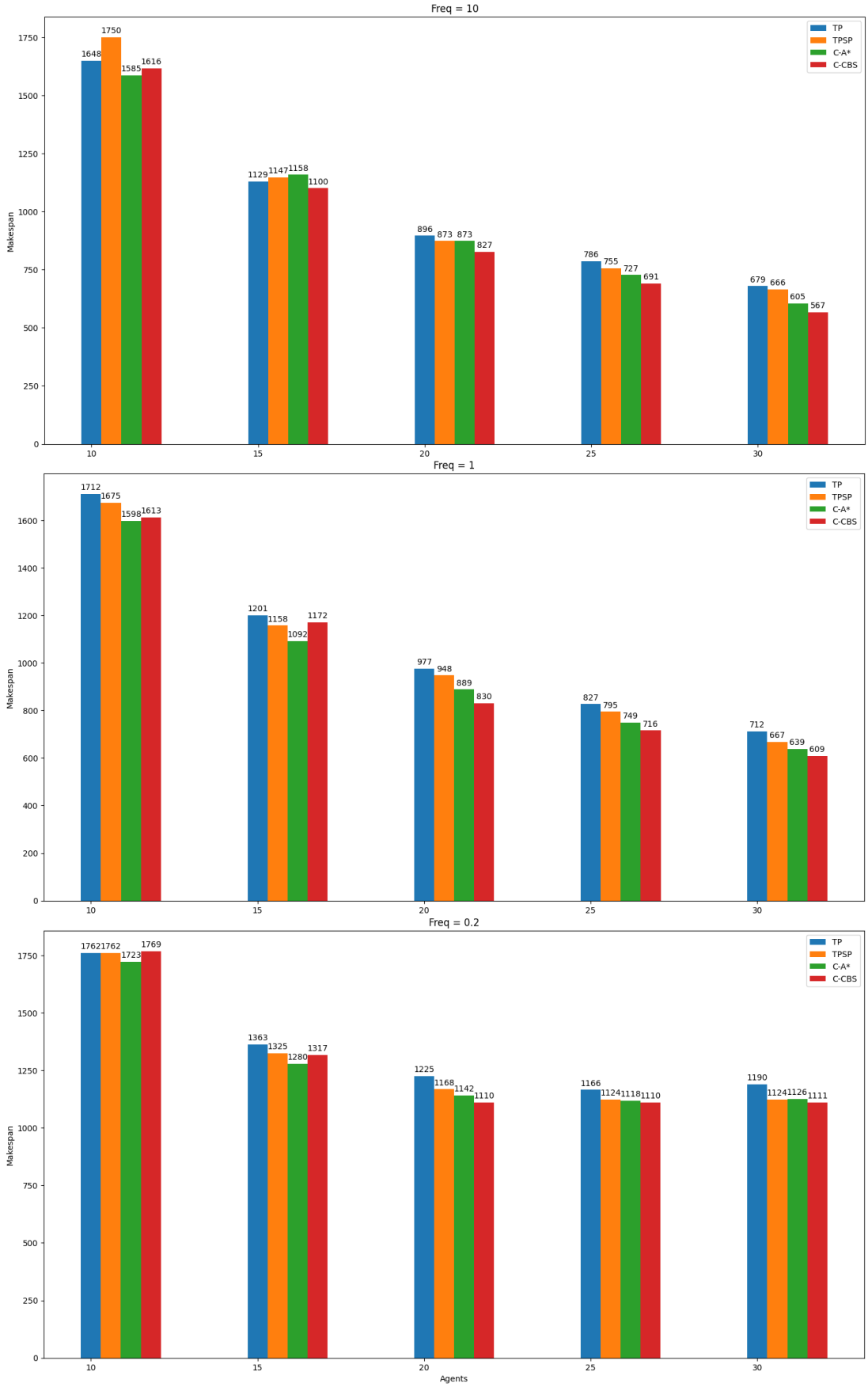


Figure 5.2: The figure shows selected makespan results for the first experiment

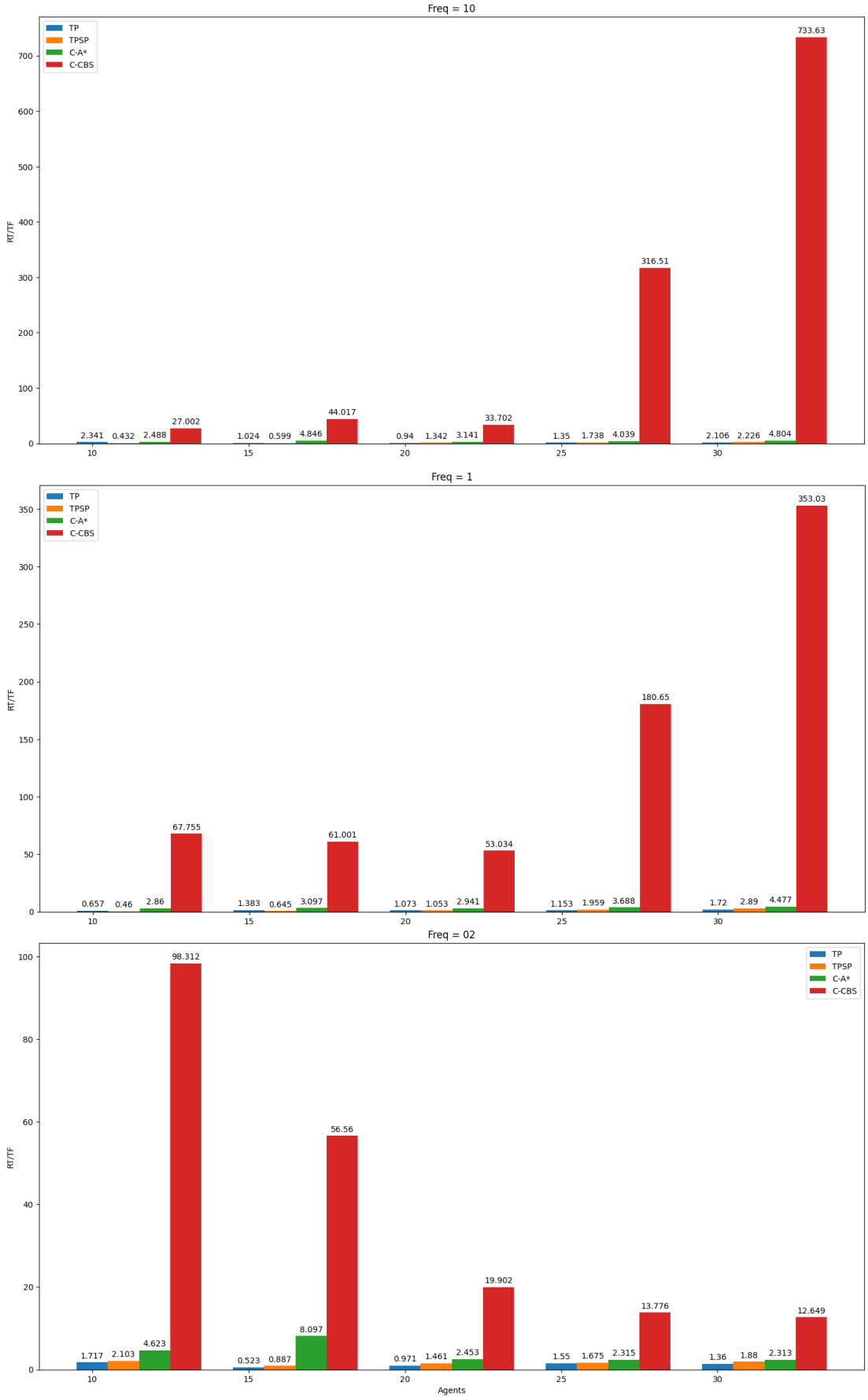


Figure 5.3: The figure shows selected runtime results for the first experiment

Makespan

The difference of makespan peaks in the largest agent and frequency scenarios. We can see, that algorithms in ascending order of their makespans in largest scenarios are TP, TPSP, C-A*, and C-CBS. On the other hand, the smallest difference is in the low-frequency scenario. The makespans for the smaller number of agents and small frequencies are negligible. The reason for this is that due to a lack of conflicts, the orders are finished at nearly optimal times across all algorithms. The differences start to build up as the number of agents increases. We can see that the CBS-based algorithm outperforms all others.

There is a clear correlation between makespan and frequencies. The scenario with 0.2 frequency has a vastly greater makespan simply because agents are not able to finish orders before the last order appears + its time to distance. On the other hand, makespan for high-frequency scenarios is far lower as agents work with most of the orders right away.

Runtime

We can see that the runtimes on algorithms are ascending by the following order: TP, TPSP, C-A*, and C-CBS. This does not come as a surprise, it follows the same order as the performance makespan-wise. We can see the neverending tradeoff between performance and computational time.

Runtime per timeframe measures the ability of algorithms to perform well in real-life MAPD scenarios. The logical bound for accepting an algorithm as suited for live operation is one second per time frame. We can observe from the measured data, that all algorithms satisfy this condition. We can assume they are all suitable for the scenario of this size.

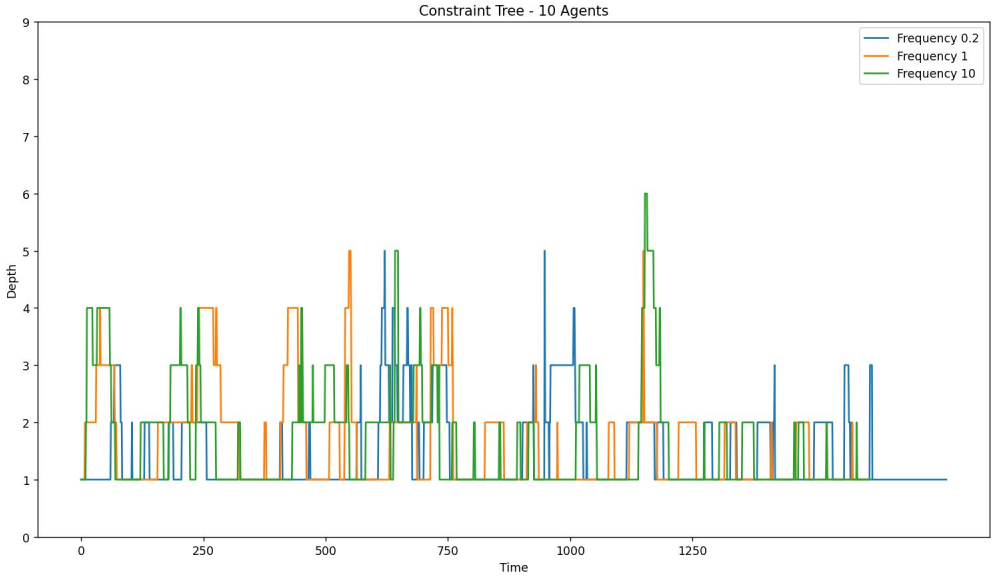
There is little to no correlation between frequencies and runtime for TP and TPTS. CENTRAL benefits greatly from lower-frequency scenarios. As most of the agents are assigned mock locations, fewer conflicts are created thus the whole algorithm funs much faster. More about this is in the CBS Constrain tree subsection.

Little surprising is the fact, that CENTRAL-A* is only slightly worse in runtime, than Token Passing algorithms yet its performance is better. The reason for this hides in our settings, in distance precomputing to be exact. We are working with pre-counted distances for all locations, thus the A* works with exact distance values, not heuristics. This increases the speed of A* dramatically. The result is that running multiple instances of A* every time frame is not computationally demanding thus the resulting time is not greatly increased.

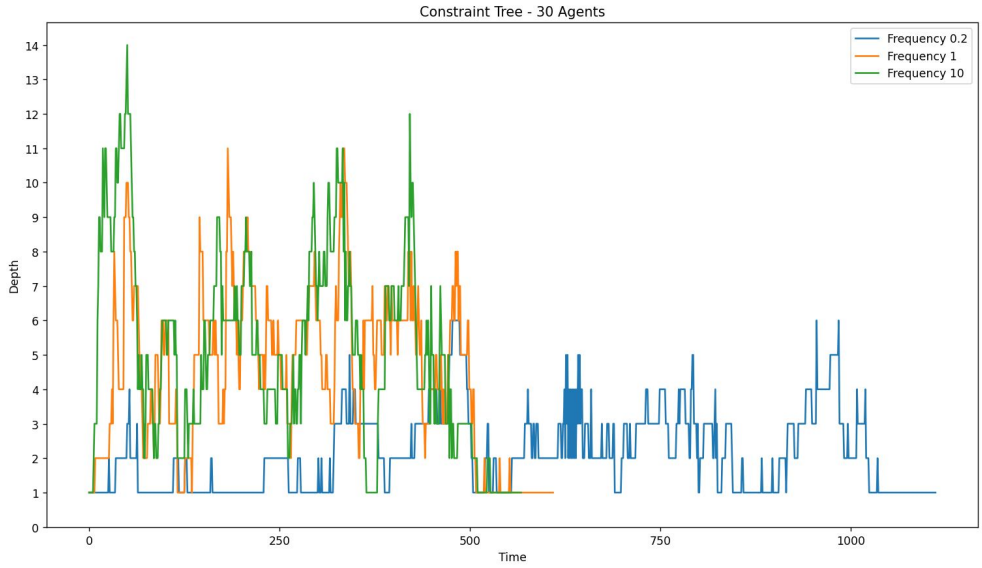
CBS Constrain tree

The first figure shows the graph of depths of the CBS constraint throughout the run of the algorithms. for 10 agents. The second figure shows the same for the maximal agent number of 30. We can observe that the busiest part of the algorithm creates deeper trees for such small differences in the number of agents. For 10 agents, the average depth is around 2, this effectively means that the algorithm runs very fast. Given that CT is binary, the computational hardness doubles for every level. If the runtime of C-CBS scaled linearly with the number of agents, we would observe the average depth of a CT for 30 agents around 3.5. This is however not the case. For higher-frequency runs, the average depth of a tree is around 6.5. That is about 16 times the computational hardness for just 3 times the agents. Even though the number of time frames is halved compared to the

10-agent scenario, we can conclude that this algorithm is not very suitable for a large number of agent scenarios.



(a) CT-10A



(b) CT-30A

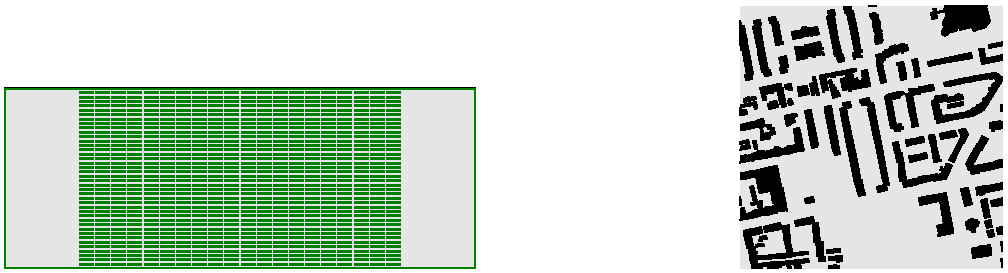
Figure 5.4: The figure shows CBS-CT depth

5.3 Experiment 2

5.3.1 Settings

In this experiment, we aim to compare the TP algorithm in a large warehouse environment with various numbers of agents and heuristics. TPTS and CENTRALs do not scale well enough and did not manage to solve such a high number of agent scenarios on this large map.
The first goal is to compare various TP algorithms based on the order of the

agents, in which they are given access to the token. In Experiment 1, random order was assigned in the beginning and never changed. We denote this as "Fixed". We add two more types of orders. The first one is a random permutation of idle agents every time frame. This is denoted as "random". The second one is the distance to the closest order. The idea is that the agent with the longest route should be prioritized with the access to create as small a makespan as possible. The agents are ordered in descending order based on their Manhattan distance to their closest order. This is denoted as "mdist".
The second goal is to compare the efficiency of the solution based on the number of agents and frequency. This will be measured in makespan as well as cost and service time. The cost is calculated as a sum over all agents and their movements where the move action has a cost of 1, the wait function has a cost of 0.1. This should somewhat simulate real-world usage where paying for extra agents would become inefficient at some point.
Service time is a ratio of the ideal time of delivering orders (measured by heuristic) and the real one. It is calculated as an average of ideal times for all orders and the real ones. The closer to 1, the more effective the algorithm.
The algorithm will be compared separately on two different maps. One map is a large warehouse, and the second one is "Berlin" and its purpose is to simulate a complex environment, for example, an airport, with its irregular structure.



(a) Warehouse-321x143 (b) Berlin-256x256.map

Figure 5.5: The figure shows "Warehouse" and "Berlin" maps respectively

5.3.2 Results

Warehouse
The TP algorithm managed to solve all instances quite quickly, so the assumption that it does scale well was confirmed.

Settings		TP-random			TP-MDIST		
TF	A	MS	cost	ST	MS	cost	ST
2	100	**2766**	**220268**	**0.18**	2867	229665	0.17
2	150	**2094**	**233990**	**0.27**	2219	246245	0.24
2	200	**1653**	**241586**	**0.35**	1870	262153	0.30
2	250	1690	**262506**	**0.42**	**1687**	279734	0.35
2	300	**1441**	**275576**	**0.47**	1623	298601	0.39
2	350	2088	**275576**	**0.51**	**1511**	315393	0.41
2	400	**1128**	**287326**	**0.56**	1231	320849	0.44
2	450	**2068**	**335817**	**0.63**	—	—	—
2	500	**1156**	**322320**	**0.59**	1262	325279	**0.59**
50	100	**2614**	**212459**	**0.16**	2746	214429	**0.16**
50	150	**1866**	**215531**	**0.24**	2029	221322	0.23
50	200	**1524**	**220051**	**0.30**	1555	225793	0.28
50	250	**1357**	**228294**	**0.36**	1450	238426	0.33
50	300	1294	**239914**	**0.40**	**1278**	246530	0.37
50	350	**1043**	**233211**	**0.45**	1451	310332	0.37
50	400	**1067**	**247932**	**0.49**	1094	260654	0.44
50	450	**1216**	**275511**	**0.46**	—	—	—
50	500	**941**	**261304**	**0.56**	966	280976	0.49

Table 5.2: Experiment 3 results on large warehouse.

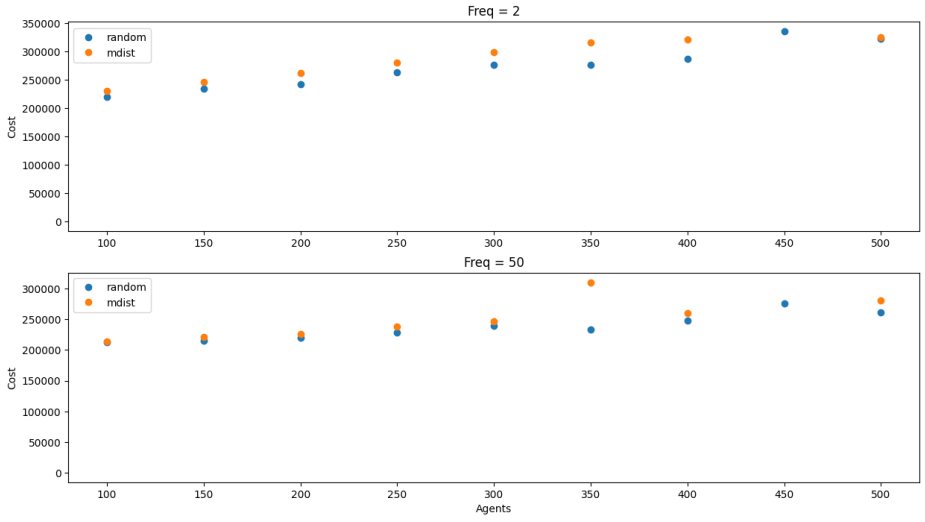


Figure 5.6: The figure shows results in cost

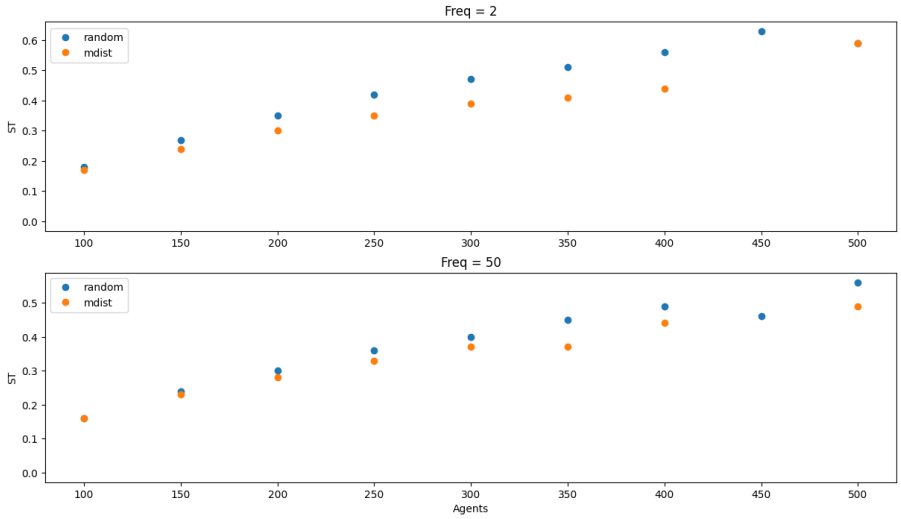


Figure 5.7: The figure shows results in service time

The overall better-performing heuristic is ”random”, but just slightly and with exceptions. In bigger scenarios. it is always better to have some probability involved because determinism might lead to unwanted deadlocks (viz Fig 5.2, 450 agents) or converge to the local optimum too fast.
Better efficiency was achieved for scenarios with low order frequency. This is logical because agents do not become overwhelmed with the number of orders and manage to resolve the order quickly compared to the high-frequency scenarios, where a lot of the orders have to wait for some agent to finish their task, so the service times become worse. Service times become more efficient with the increased number of agents for the same reason.
In a slower pace scenario, the optimal number of agents seems to have an upper bound of about 400. The increase in makespan as well as a spiking increase in cost suggests that 500 agents is way past the optimal number. The smaller cost difference along with meaningful improvement in makespan suggest the lower bound around 330 agents.
For the faster pace, we can observe that the most costly decrease in makespan is between 400 and 500 agents. Even though the makespan does still decrease, based on the cost-ST ratio, we can argue that the lower recommended number of agents is about 400 whereas the upper bound might be slightly past 500.

Berlin

Settings		TP-random			TP-MDIST		
TF	A	MS	cost	ST	MS	cost	ST
2	100	**2625**	**222259**	**0.16**	2794	230033	0.15
2	150	**2103**	**235927**	**0.24**	2108	243885	0.22
2	200	**1766**	**248708**	**0.31**	1819	260375	0.27
2	250	**1498**	**255541**	**0.37**	1635	276405	0.31
2	300	1514	304344	**0.38**	**1494**	**292343**	0.35
2	350	**1302**	**281321**	**0.45**	1518	307712	0.38
2	400	**1250**	**293299**	**0.48**	1282	317658	0.40
2	450	**1343**	**306379**	**0.51**	1392	390408	0.36
2	500	**1226**	**323119**	**0.51**	—-	———	—-
50	100	**2338**	**209114**	**0.15**	2615	217059	0.14
50	150	**1835**	**218557**	**0.21**	1929	223492	0.20
50	200	**1577**	**224832**	**0.26**	1819	232768	0.25
50	250	**1409**	**229876**	**0.31**	1483	244368	0.29
50	300	**1221**	**235718**	**0.35**	1303	248792	0.33
50	350	1193	**241994**	**0.40**	**1135**	254717	0.36
50	400	**1051**	**247215**	**0.42**	1217	270815	0.38
50	450	**1069**	**255098**	**0.46**	1122	276768	0.40
50	500	**1343**	**259028**	**0.49**	—-	———	—-

Table 5.3: Experiment 3 results on complex map

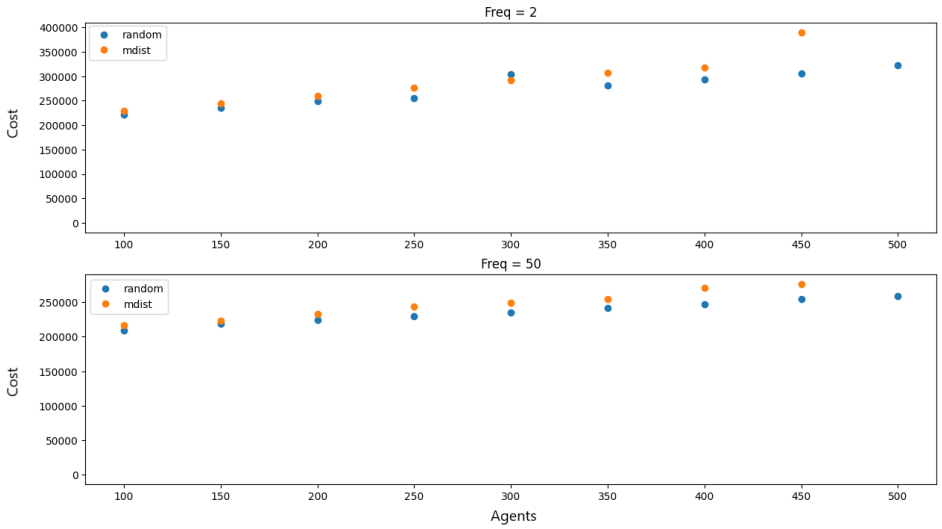


Figure 5.8: The figure shows results in cost

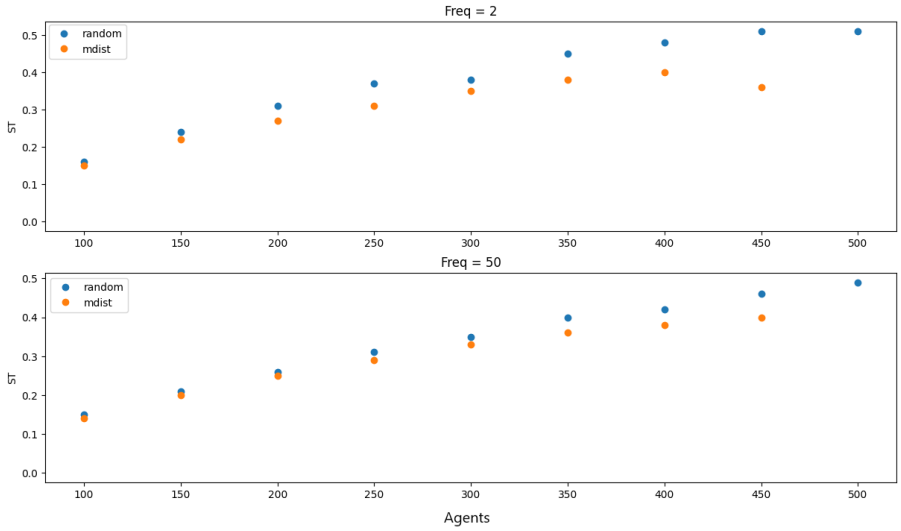


Figure 5.9: The figure shows results in service time

We can observe that the random heuristic vastly outperformed the distance-based one. This is likely due to the irregular structure of the map, in the warehouse map, this difference was not as significant.
The pattern with better service time for low-frequency scenarios continues in this experiment as well. On the other hand, better makespan results are for the fast pace scenarios, this is again due to agents being able to process orders in a much shorter time overall. For the highest number of agents, however, we can see that the efficiency as well as makespan either decreases only slightly or even increase, whereas the price increases significantly. Based on the results of TP with random heuristic and cost-ST ratio, we can deduct that for lower density scenarios, the optimal number of agents will be around 350-400. For high-density scenarios, the optimal number should be within the range of 375-425.
For the mdist based tp, this range is lower for both scenarios. Without the randomness, the algorithm starts to struggle greatly with an increased number of agents.

5.3.3 Conclusion

In this experiment, we have shown the difference between distance-based and random heuristics based on multiple criteria. We estimated the optimal number of agents for a given map and order frequency and showed the relation between order frequency, agent number, and makespan as well as cost and service time. We can also suggest, that for structurally demanding maps, randomness improves the algorithm, leads to better runtime, and decreases the danger of deadlocks.
We will use some of the data, knowledge, and observations for the final experiment, which aim is to estimate the ideal number of agents given the map, order count, and order frequency.

5.4 Experiment 3

5.4.1 Settings

This experiment aims to compare the CENTRAL-CBS algorithm with and without the pruning mentioned in the third chapter. The algorithms will be compared in the Room-64-64-16.map environment with two different frequencies: 0.5 and 5 simulating slower and faster pace of knowledge about the orders. They will be compared on two different criteria: makespan and runtime. The goal is to empirically show, that the pruning version is faster whereas the change in makespan is negligible.

5.4.2 Results

The results are shown in the Table 5.4. We can observe that for the higher as well as the lower frequencies, the makespans do not differ. However, the runtime is slightly decreased for the pruning version, as noted in the ratio column.
The main difference happens at the beginning and towards the end of the problem instance. At no other time, do we have to deal with idle agents, so the runtime difference is not as significant. The basic idea on CT pruning can be expanded for higher level agent scenarios based on either local heuristic of predefined agent priorities (viz algorithm description).
The version of CBS without pruning did not manage to find a solution for some of the instances. We had to implement the local search described in the CT pruning chapter to guide the final steps of the algorithm. This is another advantage of using the pruned version, in a state with multiple idle agents and only a few orders to deliver, the occupied agents solving these few orders are always given priority. The difference in runtimes is negligible for the 10 agents' scenario. This

Settings		C-CBSp		C-CBS		
TF	A	MS	RT / TF	MS	RT / TF	ratio
0.5	10	1673	81.643	1673	81.643	1
0.5	20	920	45.168	920	65.945	1.46
0.5	30	631	371.70	631	416.30	1.12
5	10	1656	24.301	1656	24.301	1
5	20	871	51.884	871	63.298	1.22
5	30	618	1067.4	618	1195.4	1.12

Table 5.4: Experiment 3 results.

is to be expected because as shown in the first experiment (chap. 5.2), there are almost no conflicts so the search space is seldom increased in the nonpruned version. The main difference is for 20 agent scenarios. The overall runtime is not so high that the part with idle agents makes up a nontrivial part of it. There are also enough agents to create obstacles so that the actual search space is visibly increased. For the 30 agents scenario, the difference is still visible but gets smaller and the assumption is, with the increased number of agents, for this optimization, the difference would go near zero. The main part of the computation happens when all agents are occupied. This state is computationally the hardest as the

number of conflicts increases significantly. Part with the idle agents again makes up only the trivial part of it.
The overall difference is lower for high-frequency scenarios. This is yet again expected, as the part with idle agents is much shorter. To be exact, for 20 agents and an order frequency of 5, by the time step 5, all agents have become occupied, whereas, for 0.5 order frequency, this happens only after 41 time steps. Towards the end of a solution, more and more agents become idle without assigned orders. This is expected to be about the same for all scenarios so that the biggest difference happens at the beginning. Therefore the optimization is more efficient in low-frequency scenarios.

5.5 Experiment 4

5.5.1 Settings

For our last experiment, we attempt to find a formula, which would give us an approximate ideal number of agents for a given map. This would be very useful if we wanted to see how efficient may certain warehouses or different locations get. It is also a good first estimate of how expensive is the operation on a certain map going to get without prior testing and measurements which can be quite costly.
Observations:
Observation 1: From previous experiments, we can deduce, that if we only increase the number of agents, at some point, it will become slower and more costly as they block each other's paths.
Observation 2: With an increased number of orders, an increased number of agents is needed. This however stops at some point and it does not matter how many orders the scenario has, the optimal number of agents is approximately the same.
Observation 3: The same holds for frequencies. In faster pace scenarios, more agents are needed, but on most maps, it barely matters if the frequencies are 100/tf or 50/tf, the ideal number of agents is about the same.
Observation 4: The shape of a map has a great impact on number of agents needed. The first important parameter is the dimensions, then available locations compared to the wall locations, and finally the structure itself. Trivial observation is that if we have a map with 10x10 free locations square, we may utilize more agents than in a maze which has 100 available locations in total. This is the most important, yet difficult aspect of this experiment. We need to come up with a way to systematically analyze maps and use the results in the prediction itself. For this experiment, we will make several assumptions under which the experiment will be conducted. We will base the assumption on the observations as well as previous experiments and knowledge about the algorithms.
Assumptions:
Number of orders: We take the number of orders as a fraction of available locations on each map. Based on previous experiment we set this number to $\frac{free-locations}{10}$
Equal frequency: We make frequency for each scenario such that all orders will be available at time $t = 100$.
Used algorithm: For this experiment, we only use TP with a random ordering

heuristic, the best-performing algorithm on larger maps.
The goal: If we use makespan as our utility function, we may not obtain desirable results. In previous sections, some of the scenarios are solved in smaller makespan for an extremely high price, this is undesirable. Working with price as the sole utility function does not work either, based on the experiments, with an increased number of agents, the price rises even though the makespan lowers. This is because even idle agents contribute to the cost function. In reality, companies try to minimize price and maximize value. We can say that they value efficiency by some cost. We will do the same so that we can decide whether one result is better than the other based on the price of the efficiency increase of the solution. The used formula for evaluation is:

$$\frac{1}{log(price * makespan)} \tag{5.1}$$

Because we try to minimize price as well as makespan, we will be interested in the maximum values of the evaluation function.
Map analysis:
The hardest part is to analyze the map and obtain meaningful info, which can be utilized in approximating the ideal number of agents. We will simplify the map analysis to four factors. The first parameter p is the size of the maneuver space. This area determines whether agents can comfortably dodge each other after crossing the choke or while waiting for it. We can observe that in the room-16 map, this is rather high as all the rooms create huge maneuver space. In a maze with choke sizes of 1, however, there is no such area, which effectively decreases the optimal number of agents to units at most. The q is going to be the number of chokes and it will be determined by the number of pairs of locations that must cross the choke to arrive from one another. If this number of pairs is sufficient enough, the size of the chokes will determine the second parameter. The "sufficient enough" is a vague requirement, we would need exhaustive measurements to set this parameter to the exact value. For this experiment, we just take the category of the smallest repeating choke. The d parameter is related to the density of the chokes. And fourth parameter is the throughput of the map, meaning the ability of the agents to comfortably move between different areas of the map.
We omit some of the measurements in graphical or table visualization for the sake of keeping this section short. They are all available as attachments. We begin with testing on a plain map without any obstacles to receive base case info. The results are the following:

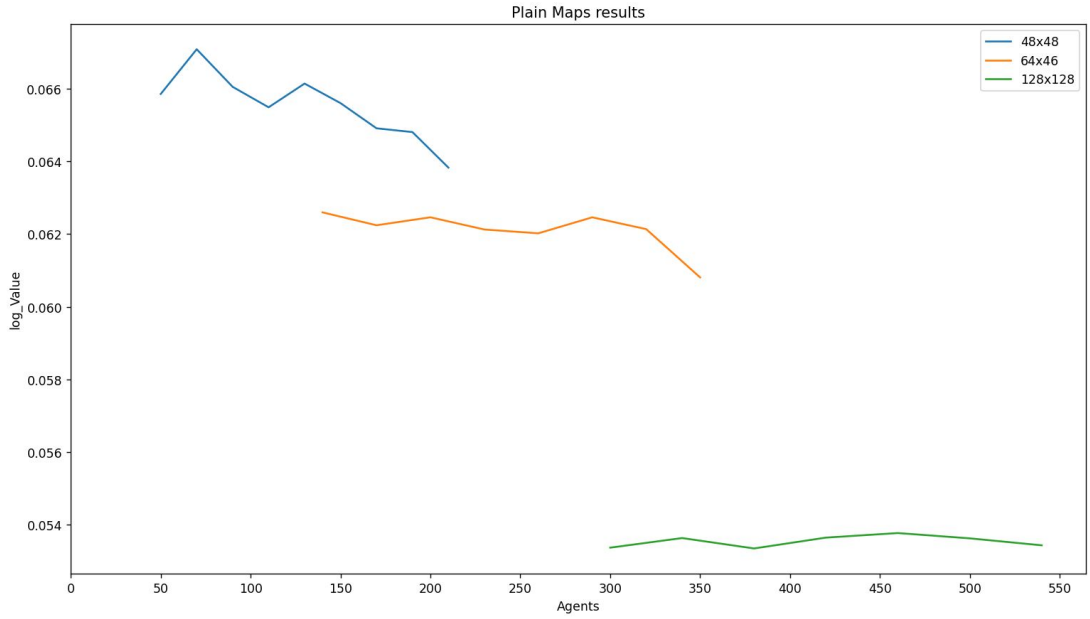


Figure 5.10: The figure shows results of plain maps tests

Result Description: We have highlighted the optimal range for each of the plain maps. We can see that the efficiency falls off with the greater number of agents, except for the largest map. The range is also by far the biggest for the 128x128 map. We can assume that the optimal range widens with the increased length of the map. That is after all logical conclusion. Based on the results, we roughly predict the optimal number of agents for plain maps as follows:

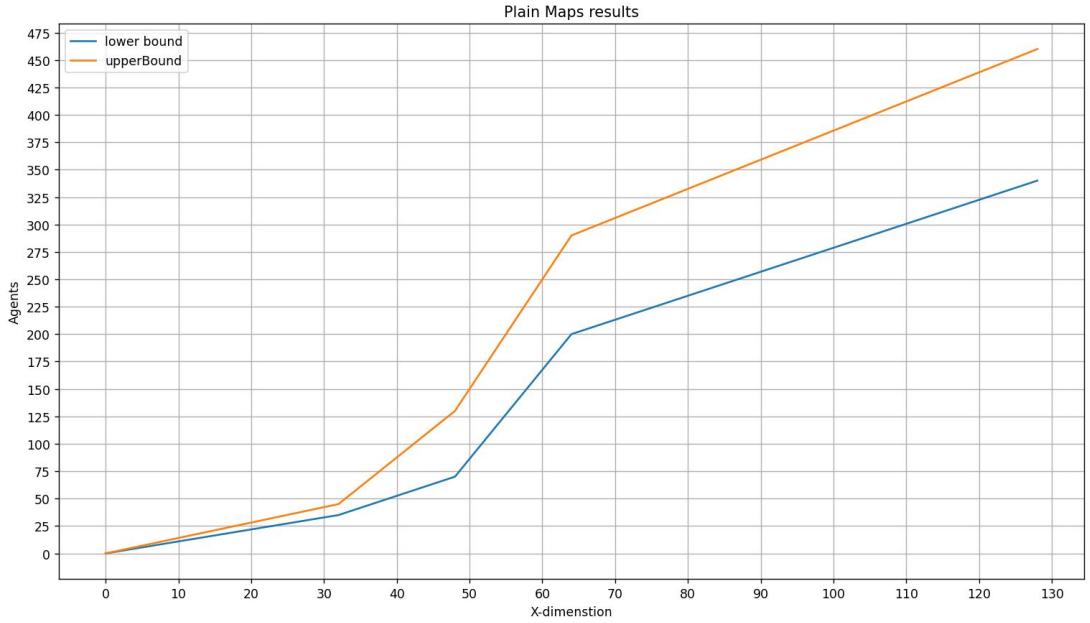


Figure 5.11: The figure shows prediction on plain maps

$p, q, d \in [0, 1]$ for simplification, $p, q, d \in \{S = 0.1, M = 0.4, L = 0.75\}$.
The d will be set somewhat experimentally based on measurements and then

accordingly set on the same map types. In the case of a maze, with constant corridor sizes, the d must go towards 1. In maps with barely any chokes or areas with deadlock potentials, the d will be rather low, and for plain maps, it is zero whereas throughout is set to one.
Room 64x64-16: The chokes are the size of 1 and all the locations in different rooms are connected through the chokes. We set the choke parameter to S. The sub rooms however are large in terms of the rest of the map, so we declare the first parametr as L. The throughput is rather worse, we set it to S.
In a similar fashion, we set parameters for multiple maps and conduct measurements. The base optimal range is given by the earlier estimate and the size of non-wall locations. For example, if we have 14000 free locations, we take x as a square root and look up the corresponding value on the base case predictions.

Map	Parameters				Base opt range	Opt range
Name	p	q	t	d	bounds	bounds
ROOM-64-16	L	S	S	0.2	170-250	90-130
RAND-64-20	M	S	L	0.1	150-225	70-90

Table 5.5: Experiment 4 map measurements results

Simple formula estimation:
Let n be ideal number of agents, let n_b be base case predicted number, let p,q be first, seconds and parameter respectively. Let t be throughput and d be density-related parameter. Then

$$n \cong n_b - n_b * (1 - p^{1-t} * q^{1-d}) \tag{5.2}$$

The p is the main parameter and is always bigger than q. The reason is that the maneuver space must be greater or equal to the chokes. It is modified by the throughput parameter. This is because some maps might have small open maneuver space, but they have multiple different paths to choose from to various map sections. A great example is map Random-64x64x20. The q parameter denotes the choke areas and frequency controls the impact of this parameter. If we look at the Maze-128x128-10 map, we can see that the frequency of the chokes is constant and equal to the maneuver space, thus making the second parameter irrelevant. The frequency is set to one and the choke parameter is omitted. By applying the formula to the base case maps, we obtain the expected:

$$n \cong n_b - n_b * (1 - 1^{1-1} * 1^{1-0}) = n_b \tag{5.3}$$

The formula is based on empirical measurements and observations of argument correlation throughout the experimentations.
We will test the formula on three different maps. The d parameter for ROOM-64-8 is set to 0.1 because the density is twice the density of ROOM-64-16. For the same reason, the d parameter of RAND-64-10 is set to 0.2 as the density is half of the RAND-64-20. The d parameter in the maze is set to one because the size of all corridors is equal to the size of all chokes and thus manoeuver spaces, so the q parameter is irrelevant.

Map	Parameters				Base opt range
Name	p	q	t	d	bounds.
ROOM-64-8	M	S	M	0.1	150-225
RAND-64-10	M	M	L	0.2	170-250
MAZE-128-10	M	M	M	1	325-430

Table 5.6: Experiment 4 map prediction parameters

5.5.2 Results

Map	Estimate	Opt range
RAND-64-10	65-95	70-150
ROOM 64-8	11-17	50-70
MAZE-128-10	190-300	210-250

Table 5.7: Experiment 4 map prediction results

5.5.3 Conclusion

We have roughly introduced the problematics of an optimal number of agents on an unfamiliar map. We suggested multiple arguments and factors that contribute to this value and we conducted small-scale experiments. With knowledge of prior measurements on a map of a similar type, we can very leniently estimate the number of agents required to solve the scenarios on different maps in terms of best value using the suggested formula. It has, however, many issues. Setting the parameters correctly is very difficult and the simplified domain yields inaccurate results (see ROOM 64-8 results). The optimal range gets narrow as the number of agents decreases which is not accounted for in the formula accurately, etc. The "correct" formula is likely much more complex.
This problem is rather going to be tackled by exhaustive measurements combined with machine learning analysis. This topic would be adequate for the paper on its own.

Conclusion

The Multiagent pickup and delivery (MAPD) problem has risen in popularity in recent years along with robotization and automatization. We have covered multiple different approaches to the MAPD and some of their follow-ups. The performance of multiple algorithms and heuristics was compared based on various goal functions and scenarios. Furthermore, discussed the problematics of optimal range of agents on different maps and scenarios.

The basic pointer of the efficiency of the algorithm is makespan or service time. The sophisticated algorithms outperformed the simple ones, makespan-wise even by 15-20%. Even the "agent swapping heuristic" technique significantly improved and outperformed Token passing in certain scenarios by as much as 10%. Further improvement was achieved using central algorithms with local search. We have also introduced a heuristic for active agent prioritization, which improved the C-CBS runtime as well as helped prevent deadlocks. On the other hand, the depth of a CT was shown to demonstrate the inability of C-CBS to scale. To measure, whether an algorithm is suitable for real-world planning, we described the requirement of runtime per frame. No other algorithm, than TP manage to consistently finish its calculation within the limit, therefore we only considered TP in large scenario experiments. We measured the effectivity as well as the cost of the TP in a large warehouse and complex scenarios for various numbers of agents, described the scenarios, and based on the results suggested the number of agents required to maximize the utility on different maps.

The significance of the result explanation was described. Especially for high pace environments, humans need to understand and verify the result of algorithm computation. The example methods for MAPD explanation were shown and demonstrated on testing software.

In conclusion, we demonstrated the results of MAPD algorithms on small as well as large scenarios. We have identified the weakness of the robust C-CBS algorithm as well as the strength in the simplicity of the TP algorithm and its improved version. Theoretical extensions of TP for concrete situations were described and argued as promising. With the increasing trend of automatical warehouses, in Czechia namely Alza, Mall, and Amazon, the research on multiagent pickup and delivery problems proves to be relevant and quite significant. Given the hardness of the problem, it is unlikely that an optimal general algorithm will be found, rather we ought to find improvements for concrete situations.

Bibliography

Natalie Abreu. Efficient deep learning for multi agent pathfinding. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36(11):13122–13123, Jun. 2022. doi: 10.1609/aaai.v36i11.21697. URL `https://ojs.aaai.org/index.php/AAAI/article/view/21697`.

Shaull Almagor and Morteza Lahijanian. Explainable multi agent path finding. In *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '20, page 34–42, Richland, SC, 2020. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 9781450375184.

Roman Barták, Neng-Fa Zhou, Roni Stern, Eli Boyarski, and Pavel Surynek. Modeling and solving the multi-agent pathfinding problem in picat. In *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 959–966, 2017. doi: 10.1109/ICTAI.2017.00147.

Zahy Bnaya and Ariel Felner. Conflict-oriented windowed hierarchical cooperative a*. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3743–3748, 2014. doi: 10.1109/ICRA.2014.6907401.

Eli Boyarski, Ariel Felner, Roni Stern, Guni Sharon, Oded Betzalel, David Tolpin, and Eyal Shimony. Icbs: The improved conflict-based search algorithm for multi-agent pathfinding. In *Proceedings of the International Symposium on Combinatorial Search*, volume 6, pages 223–225, 2015.

Zhe Chen, Javier Alonso-Mora, Xiaoshan Bai, Daniel D. Harabor, and Peter J. Stuckey. Integrated task assignment and path planning for capacitated multi-agent pickup and delivery. *IEEE Robotics and Automation Letters*, 6(3):5816–5823, 2021. doi: 10.1109/LRA.2021.3074883.

Florian Grenouilleau, Willem-Jan van Hoeve, and J. N. Hooker. A multi-label a* algorithm for multi-agent pathfinding. *Proceedings of the International Conference on Automated Planning and Scheduling*, 29(1):181–185, May 2021. doi: 10.1609/icaps.v29i1.3474. URL `https://ojs.aaai.org/index.php/ICAPS/article/view/3474`.

Taoan Huang, Jiaoyang Li, Sven Koenig, and Bistra Dilkina. Anytime multi-agent path finding via machine learning-guided large neighborhood search. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36(9):9368–9376, Jun. 2022. doi: 10.1609/aaai.v36i9.21168. URL `https://ojs.aaai.org/index.php/AAAI/article/view/21168`.

H. Kitano, S. Tadokoro, I. Noda, H. Matsubara, T. Takahashi, A. Shinjou, and S. Shimada. Robocup rescue: search and rescue in large-scale disasters as a domain for autonomous agents research. In *IEEE SMC'99 Conference Proceedings. 1999 IEEE International Conference on Systems, Man, and Cybernetics (Cat. No.99CH37028)*, volume 6, pages 739–743 vol.6, 1999. doi: 10.1109/ICSMC.1999.816643.

H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955. doi: https://doi.org/10.1002/nav.3800020109. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/nav.3800020109.

Jiaoyang Li, Andrew Tinka, Scott Kiesel, Joseph W Durham, TK Satish Kumar, and Sven Koenig. Lifelong multi-agent path finding in large-scale warehouses. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 11272–11281, 2021.

Minghua Liu, Hang Ma, Jiaoyang Li, and Sven Koenig. Task and path planning for multi-agent pickup and delivery. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '19, page 1152–1160, Richland, SC, 2019. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 9781450363099.

Hang Ma, Jiaoyang Li, TK Kumar, and Sven Koenig. Lifelong multi-agent path finding for online pickup and delivery tasks. *arXiv preprint arXiv:1705.10868*, 2017.

Van Nguyen, Philipp Obermeier, Tran Cao Son, Torsten Schaub, and William Yeoh. Generalized target assignment and path finding using answer set programming. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 1216–1223, 2017. doi: 10.24963/ijcai.2017/169. URL https://doi.org/10.24963/ijcai.2017/169.

Zhenbang Nie, Peng Zeng, and Haibin Yu. Effective decoupled planning for continuous multi-agent pickup and delivery. In *2020 Chinese Control And Decision Conference (CCDC)*, pages 2667–2672, 2020. doi: 10.1109/CCDC49329.2020.9164394.

Guni Sharon, Roni Stern, Meir Goldenberg, and Ariel Felner. Pruning techniques for the increasing cost tree search for optimal multi-agent pathfinding. In *Proceedings of the International Symposium on Combinatorial Search*, volume 2, pages 150–157, 2011.

Guni Sharon, Roni Stern, Meir Goldenberg, and Ariel Felner. The increasing cost tree search for optimal multi-agent pathfinding. *Artificial Intelligence*, 195:470–495, 2013. ISSN 0004-3702. doi: https://doi.org/10.1016/j.artint.2012.11.006. URL https://www.sciencedirect.com/science/article/pii/S0004370212001543.

Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, 219:40–66, 2015. ISSN 0004-3702. doi: https://doi.org/10.1016/j.artint.2014.11.006. URL https://www.sciencedirect.com/science/article/pii/S0004370214001386.

David Silver. Cooperative pathfinding. In *Proceedings of the aaai conference on artificial intelligence and interactive digital entertainment*, volume 1, pages 117–122, 2005.

Roni Stern. *Multi-Agent Path Finding – An Overview*, pages 96–115. Springer International Publishing, Cham, 2019. ISBN 978-3-030-33274-7. doi: 10.1007/978-3-030-33274-7_6. URL https://doi.org/10.1007/978-3-030-33274-7_6.

Roni Stern, Nathan R. Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne T. Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, T. K. Satish Kumar, Eli Boyarski, and Roman Bartak. Multi-agent pathfinding: Definitions, variants, and benchmarks. *Symposium on Combinatorial Search (SoCS)*, pages 151–158, 2019.

Peter R. Wurman, Raffaello D'Andrea, and Mick Mountz. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI Magazine*, 29(1): 9, Mar. 2008. doi: 10.1609/aimag.v29i1.2082. URL https://ojs.aaai.org/aimagazine/index.php/aimagazine/article/view/2082.

Qinghong Xu, Jiaoyang Li, Sven Koenig, and Hang Ma. Multi-goal multi-agent pickup and delivery. In *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 9964–9971. IEEE, 2022.

Tomoki Yamauchi, Yuki Miyashita, and Toshiharu Sugawara. Standby-based deadlock avoidance method for multi-agent pickup and delivery tasks. *CoRR*, abs/2201.06014, 2022. URL https://arxiv.org/abs/2201.06014.

Han Zhang, Jingkai Chen, Jiaoyang Li, Brian C. Williams, and Sven Koenig. Multi-agent path finding for precedence-constrained goal sequences. In *Proceedings of the 21st International Conference on Autonomous Agents and Multiagent Systems*, AAMAS '22, page 1464–1472, Richland, SC, 2022. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 9781450392136.

Luman Zhao, Sai Rana Thattavelil Sunilkumar, Baiheng Wu, Guoyuan Li, and Houxiang Zhang. Toward an online decision support system to improve collision risk assessment at sea. *IEEE Intelligent Transportation Systems Magazine*, 15 (2):137–148, 2023. doi: 10.1109/MITS.2022.3190965.

List of Figures

1.1 Ma et al. [2017] Example MAPD instances 6
1.2 The figure shows MAPD instance from mapd-visual software . . . 9

4.1 The figure shows plan segment visually explained in MAPD-visual software. 23

5.1 The figure shows Room-64-64-16 map 25
5.2 The figure shows selected makespan results for the first experiment 28
5.3 The figure shows selected runtime results for the first experiment 29
5.4 The figure shows CBS-CT depth 31
5.5 The figure shows "Warehouse" and "Berlin" maps respectively . . 32
5.6 The figure shows results in cost 33
5.7 The figure shows results in service time 34
5.8 The figure shows results in cost 35
5.9 The figure shows results in service time 36
5.10 The figure shows results of plain maps tests 40
5.11 The figure shows prediction on plain maps 40

A.1 Maps used in 4th experiment 50
A.2 Maps used in 4th experiment II 50

List of Tables

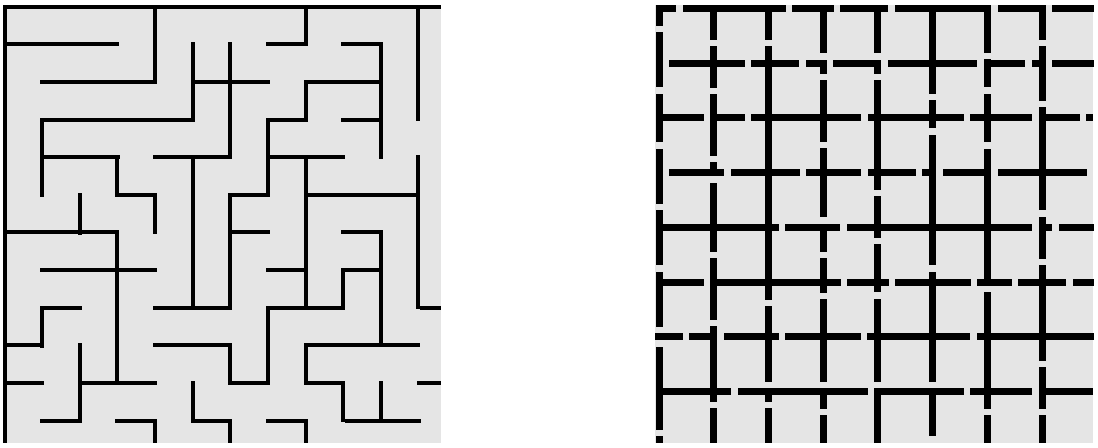
5.1 Experiment 1 results. 27
5.2 Experiment 3 results on large warehouse. 33
5.3 Experiment 3 results on complex map 35
5.4 Experiment 3 results. 37
5.5 Experiment 4 map measurements results 41
5.6 Experiment 4 map prediction parameters 42
5.7 Experiment 4 map prediction results 42

List of Algorithms

3.1 TP 15
3.2 TPTS 17
3.3 CENTRAL 19

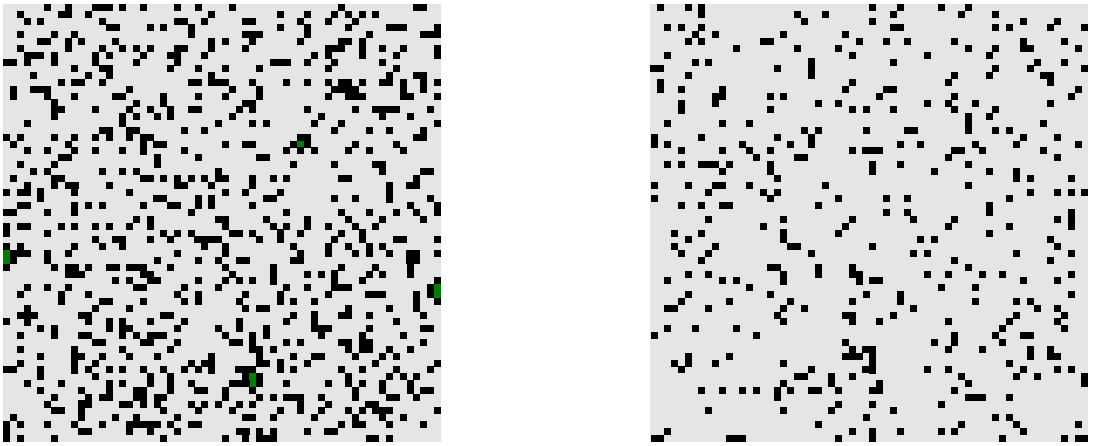
A. Attachments

A.1 Maps used in 4th experiment



(a) maze-128-128-10 (b) room-64-64-8

Figure A.1: Maps used in 4th experiment



(a) random-64-64-20 (b) random-64-64-10

Figure A.2: Maps used in 4th experiment II

A.2 Mapd-visual software used for testing and visualization

A.3 Experimental measurements

A.4 Simulation.mp4, showing simulation of selected scenarios