Multi-Robot Rearrangement in Large-Scale Warehouses

by

Baiyu Li

B.Sc., Northeastern University, China, 2020

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

SIMON FRASER UNIVERSITY
Summer 2023

Declaration of Committee

Name:	**Baiyu Li**
Degree:	**Master of Science**
Thesis title:	**Multi-Robot Rearrangement in Large-Scale Warehouses**
Committee:	**Chair:** Jason Peng Assistant Professor, Computing Science
	Hang Ma Supervisor Assistant Professor, Computing Science
	Mo Chen Committee Member Assistant Professor, Computing Science
	Angelica Lim Examiner Assistant Professor, Computing Science

Abstract

We introduce a new problem formulation, Double-Deck Multi-Agent Pickup and Delivery (DD-MAPD), which models the multi-robot shelf rearrangement problem in automated warehouses. DD-MAPD extends both Multi-Agent Pickup and Delivery (MAPD) and Multi-Agent Path Finding (MAPF) by allowing agents to move beneath shelves or lift and deliver a shelf to an arbitrary location, thereby changing the warehouse layout. We show that solving DD-MAPD is NP-hard. To tackle DD-MAPD, we propose MAPF-DECOMP, an algorithmic framework that decomposes a DD-MAPD instance into a MAPF instance for coordinating shelf trajectories and a subsequent MAPD instance with task dependencies for computing paths for agents. We also present an optimization technique to improve the performance of MAPF-DECOMP and demonstrate how to make MAPF-DECOMP complete for well-formed DD-MAPD instances, a realistic subclass of DD-MAPD instances. Our experimental results demonstrate the efficiency and effectiveness of MAPF-DECOMP, with the ability to compute high-quality solutions for large-scale instances with over one thousand shelves and hundreds of agents in just minutes of runtime.

Keywords: Multi-Robot Systems, Path Planning for Multiple Mobile Robots or Agents, Task Planning

Acknowledgements

First of all, I would like to pay my special regards to my supervisor Dr. Hang Ma, for introducing me to the areas of task assignment and path finding, which is exciting and inspiring. I would like to thank him for providing supports and patience throughout my Master's study. This thesis would not have been possible without his invaluable comments and guidance.

I would like to express my appreciation to my supervisory committee member Dr. Mo Chen for providing research discussions and answering my questions. Many thanks to my examiner Dr. Angelica Lim and my chair Dr. Jason Peng for devoting time to read my thesis and providing useful comments.

I would like to thank all the members of AIRob Lab: Xinyi Zhong, Qinghong Xu, Dingyi Sun, Danoosh Chamani, Qiushi Lin, Jiaqi Tan, Erwin Samuel, Zander Mao, Jingtao Tang, with whom I had lots of fun and inspiring conversations during both research and life.

Last but not least, I am grateful to my parents, Xiaodong Li and Quan Bai, for their unbounded support and encouragement throughout my life.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The real-world applications of multi-robot systems often require coordination between multiple agents to rearrange objects in shared environments. Warehouse robots in fulfillment centers, such as those described in [38], are one such example where agents are utilized to relocate inventory shelves. In these scenarios, it is crucial for agents to avoid collisions with each other and with the objects they are relocating. Other instances of multi-agent object transportation problems include automated container relocation and 3D automated warehouse fulfillment, where the objects can be manipulated and moved by agents. In these cases, it is imperative to prevent collisions between both agents and objects.

Much research has been done on the topic of multi-robot rearrangement of inventory shelves in automated warehouses, with a focus on the simplified Multi-Agent Path Finding (MAPF) problem [32]. In MAPF, each agent must move from its predefined start location to its predefined goal location quickly without colliding with others. Multi-Agent Pickup and Delivery (MAPD) [23] extends MAPF to a more realistic setting where there are more tasks than agents. In MAPD, each task has a predefined pickup location and a predefined delivery location. Each agent needs to get assigned a task and complete it by moving first to its pickup location and then to its delivery location. Task assignment and path finding are repeated until all tasks are completed. While MAPD is more practical for rearranging shelves than MAPF, it assumes a fixed storage layout for shelves and that shelves can only be picked up and delivered to designated locations. MAPD algorithms do not coordinate shelf movement explicitly, making them impossible to solve problems such as exchanging the locations of two shelves using a single agent.

Therefore, we introduce a novel problem formulation, Double-Deck Multi-Agent Pickup and Delivery (DD-MAPD), which extends the existing MAPF and MAPD techniques toward practical large-scale real-world warehouse autonomy. DD-MAPD addresses the multi-robot shelf rearrangement problem by allowing agents to either move beneath shelves or lift, carry, and place shelves at new locations. The name "Double-Deck" reflects the requirement to avoid collisions on two levels: on the high level, between shelves when they are being carried by agents, and on the low level, between agents themselves. The problem of DD-MAPD is

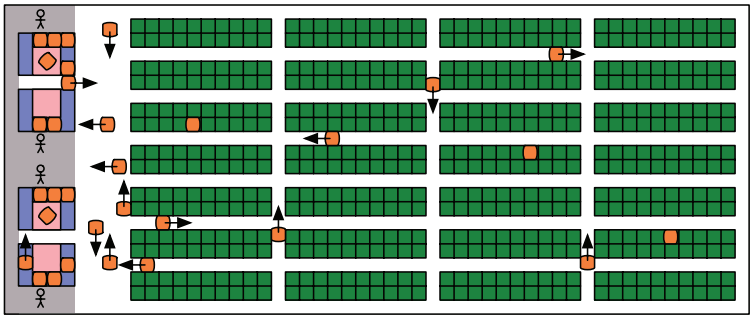

Figure 1.1: The typical layout of a fulfillment center [38].

to task N agents with moving M shelves from their given pickup locations to their given delivery locations, thereby changing the overall arrangement of the shelves.

From a practical perspective, the problem formulation of DD-MAPD enables the use of robots to dynamically adapt warehouse layouts in response to changing product demands. As shown in Figure 1, a typical layout of an Amazon fulfillment center features green cells representing designated storage locations for shelves, which are arranged in 4×7 blocks, each consisting of 10×2 storage locations. DD-MAPD allows for the creation of solutions that utilize various numbers of agents to efficiently (re)arrange shelves, such as those that are less frequently requested, into dense configurations or selectively "dig out" desired shelves from densely packed blocks by moving other shelves out of the way. This approach can significantly reduce land usage, ultimately reducing the cost of an automated warehouse.

1.1 Contributions

We propose a novel problem formulation, DD-MAPD, to model the multi-robot shelf rearrangement problem in a warehouse. Theoretically, our work generalizes existing inapproximability and NP-hardness results of MAPD to DD-MAPD and establishes a set of sufficient conditions, namely well-formedness, for solvability. Our main contribution is a new algorithmic framework, MAPF-DECOMP, which solves a DD-MAPD instance with N agents and M shelves by decomposing it into an M-agent MAPF instance, followed by a subsequent N-agent MAPD instance with task dependencies—an extension to MAPD that has received limited attention thus far. MAPF-DECOMP first solves the MAPF instance to plan collision-free trajectories for the shelves. It then converts the computed shelf trajectories into tasks and solves the N-agent MAPD instance to assign tasks and plan paths for the agents to complete all the tasks. This decomposition has a two-fold advantage. Firstly, it enables better scalability by reducing the state space and number of agents in the planning problem of $N \times M$ agent-shelf pairs. Secondly, it leverages the advancements of off-the-shelf MAPF solvers to speed up DD-MAPD solving. We also propose an optimization technique to improve the effectiveness of MAPF-DECOMP and a variant of it that

solves all well-formed DD-MAPD instances, a realistic subclass of DD-MAPD instances. Our experimental results show that MAPF-DECOMP can compute high-quality solutions for up to 1,843 shelves and 400 agents in minutes.

Chapter 2

Related Work

2.1 Single-Agent Path Finding

Single-agent path finding problem is an alias of single-source shortest path problem, where an agent is inside a graph $G = (V, E)$, whose vertices V represent locations and edges E represent the connection between these locations that the agent can move along. The agent has a start location and a goal location. The required solution is a shortest path from the start location to the goal one. It is optimally solvable with polynomial algorithms such as Dijkstra algorithm, and search-based algorithms such as A*.

2.1.1 A*

A* is a path search algorithm, which is used in many fields of computer science due to its completeness, optimality, and optimal efficiency[30]. A priority queue, named $OPEN$ is used to save vertices, also called search nodes during the search. Its pop function returns the node n with the minimum f-value: $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of the path from the start location to this vertex and $h(n)$ is a heuristics function that estimates the minimum cost from this vertex to the goal location.

$OPEN$ is initialized with only the start location. Then at each iteration, A* first pops the node with the minimum f-value from the $OPEN$. Next, this node is expanded to generate new nodes with its neighbor vertices. Lastly, new nodes are pushed into $OPEN$. Here neighbor vertices are ones that have a directed edge in G linked from the previous vertex. Finally, when the goal location is popped from $OPEN$, the shortest path's length is the g-value of the goal location's search node(the h-value of the goal location is usually 0).

With an admissible heuristics function, which means it never overestimates the cost, A* is guaranteed to return a shortest path from start to goal.

2.2 Multi-Agent Path Finding

The multi-agent path finding(MAPF) problem is a generalization of single-agent path finding, and its goal is to find collision-free paths for multi agents. Each agent is assigned a start location and a goal location. Time is discrete and for each timestep, an agent can move either to an adjacent vertex or wait at its current vertex. One set of collision-free paths is a solution if and only if all agents are at their start locations at first and are at their goal locations at the end. The completion time of an agent is the earliest timestep when the agent has arrived its goal location and stopped moving. The quality of a solution is usually measured by flowtime, which is the sum of agents' completion time, or makespan, which is the maximum of agents' completion time.

There are two kinds of conflicts leading to collisions: vertex conflict and edge conflict. A vertex conflict happens when two agents are at the same vertex at the same timestep. An edge conflict happens when two agents are moving to each other's current vertex during the same timestep.

MAPF is NP-hard to solve optimally [40, 33] on general graphs, planar graphs [39], and even 2D 4-neighbor grids [2]. Recent MAPF solvers include reduction-based [14, 34, 8, 9], rule-based [28], and search-based [31, 37, 19, 18] methods.

2.2.1 Space-Time A*

Space-time A* is a single-agent path finding algorithm used by many MAPF solvers. Space-Time A* includes one pair of vertex and time into its search node. Node $\langle v, t+1 \rangle$ can be generated from $\langle u, t \rangle$ when $u = v$ or there is a directed edge from u to v in the map. Space-time A* can meet the requirements of MAPF solvers. Firstly, it can generate a time-minimal path, which is related to the quality of a MAPF solution. Secondly, it can add constraints before the search, prohibiting the agent from moving into one specific location at some specific timestep(s), to avoid collisions with other agents.

2.2.2 Prioritized Planning

Prioritized planning(PP)[6] is a basic but efficient solver for MAPF. It simply assigns priority to agents, and plans each agent individually one after one, instead of planning multi agents simultaneously. Each time one agent plans its path with space-time A* so that the path is time-minimal and does not have collisions with former agents that are already planned.

PP avoids collision by transferring former agents' paths into constraints before starting the space-time A* search. There are three kinds of constraints from former agents: vertex constraints, edge constraints, and goal constraints. Vertex constraints and edge constraints are set to avoid vertex conflicts and edge conflicts, preventing the agent either from being at some vertex or moving from one specified vertex to another specified one, at some timestep.

Goal constraints prohibit the agent from moving into any former agents' goal locations after these agents arrive at their goal locations.

Although prioritized planning is a suboptimal algorithm with no suboptimality guarantee, it is very efficient. It is also complete in solving well-formed MAPF instances[26]. Well-formed MAPF instances are practical instances with the map that is connected even after all the start and goal locations are removed from the map. Therefore each agent can stay indefinitely at its start location or goal location without blocking any other agents.

2.2.3 Push and Swap

The push and swap algorithm[21] is a MAPF solver that can solve most instances. The only requirement is that the map should have at least two unoccupied vertices. The algorithm first assigns priority to agents. Then iteratively, the algorithm chooses an agent and plans its path, any shortest one, to its goal location. When this agent is moving along the path, another agent may block its way. If the blocking agent is with lower priority, the blocking agent will be 'pushed' away along one shortest path to an unoccupied vertex. If it is with higher priority, a 'swap' operation will be executed by making use of the unoccupied vertex, to exchange the location of these two agents. Then the blocking agent will return to its original location and the moving agent can also continue to move.

2.2.4 Conflict-Based Search

Conflict-based search(CBS)[31] is an essential algorithm in MAPF literature. It is complete and optimal. Some later optimizations make it even more efficient and practical.

CBS is a two-level algorithm and both the high-level and low-level are A*-based. The low-level algorithm, usually space-time A*, plans the path for single agent. Algorithm 1 shows that for high-level, it generates the first search node by planning each agent's shortest path individually with the low-level algorithm. Then at each iteration, it selects the node from $OPEN$ with the minimum f-value, which is the flowtime. There would probably be conflicts between agents' paths of this node, otherwise, the solution is found. CBS then generates two new search nodes with the two agents involved in the first conflict. For each new node, a new constraint is added to avoid that conflict and the low-level will replan the path of the agent considering all constraints on it. The f-value of new nodes is updated accordingly and pushed into $OPEN$.

Optimizations surrounding CBS is an active research field and recent literature are making optimizations such as classifying conflicts[4], utilizing better heuristics[7], adding accurate constraints[17], considering agent priorities[26], identifying difficult conflicts[16], and applying neighborhood search on solutions[15].

Algorithm 1: CBS

```
1  R.constraints ← ∅;
2  R.solution ← find individual paths by LowLevel();
3  R.f ← FlowTime(R.solution);
4  insert R to OPEN;
5  while OPEN ≠ ∅ do
6      P ← node with minimum f from OPEN;
7      if P has no conflict then
8          return P.solution;
9      C ← first conflict (a_i, a_j, v, t) in P.solution;
10     foreach agent a_i in C do
11         A ← new node;
12         A.constraints ← P.constraints + (a_i, v, t);
13         A.solution ← P.solution;
14         Update A.solution by invoking LowLevel(a_i);
15         A.f ← FlowTime(A.solution);
16         insert A to OPEN;
```

2.2.5 Explicit Estimation Conflict-Based Search

Explicit estimation conflict-based search(EECBS)[19] replaces CBS's high level from A* into Explicit Estimation search(EES), to provide a bounded-suboptimal solution of MAPF instance while still guaranteeing completeness. With a proper suboptimality setting, EECBS can generate an acceptable solution faster.

EES maintains three lists during the search: $CLEANUP$, $OPEN$, and $FOCAL$. Firstly, $CLEANUP$ is the priority queue used in regular A*, with a regular admissible heuristic h, and $f(n) = g(n) + h(n)$. $OPEN$ is another priority queue with a potentially inadmissible heuristic h, and $f'(n) = g(n) + h'(n)$. $FOCAL$ saves the nodes n in $OPEN$ satisfying $f(n) \leq wf'(\text{best}_{f'})$, where w is the suboptimality, and is sorted by $h'(n)$. Each time the high-level selects the best node best_d in $FOCAL$ only if $f(\text{best}_d) \leq wf(\text{best}_f)$. Otherwise it selects the best node $\text{best}_{f'}$ in $OPEN$ only if $f(\text{best}_{f'}) \leq wf(\text{best}_f)$. Otherwise, lastly, it selects the best node best_f in $CLEANUP$ and in the next round the lower bound $wf(\text{best}_f$ is thus increased, allowing it to consider best_d or $\text{best}_{f'}$.

2.3 Multi-agent pickup and delivery

Multi-agent pickup and delivery(MAPD) is a generalization of MAPF. It decouples agents and tasks. One agent has a start location. One task is with a pair of pickup location and delivery location, and a start time which is the timestep when it can start to be executed. A task is executed when an agent picks the task's pickup location, and delivers it to its delivery location. One agent can only execute one task at the same time and cannot change its task before the current one has been executed. Similar to MAPF, the solution of MAPD is a set of agents' paths, and the quality of it is measured by the service time, which is the

average time between one task's start timestep and its executed timestep. It can also be measured by makespan, which is the earliest timestep that all tasks are executed. MAPD is more complex than MAPF and is also NP-hard to solve optimally[25].

Existing MAPD algorithms [23, 20] decompose a MAPD instance into a sequence of task-assignment and MAPF instances. They assign tasks and plan paths for the agents whenever there is a change in the system, such as agents finishing tasks or new tasks being added.

MAPD problems for tasks with temporal constraints [27] or predefined dependencies [5] have also been studied. However, these problems do not model shelves as movable objects that occupy locations, which is necessary for shelf rearrangements.

2.3.1 Multi-Label A*

Multi-label A*(MLA*)[10],an extension of space-time A*, is designed for the low level search of CBS-based MAPD solvers. It considers that in MAPD, an agent needs firstly move to one task's pickup location and then to its delivery location. It takes both destinations into account during one search procedure and thus optimizes paths more globally and accurately. Otherwise, if regular space-time A* is used, it needs calculate twice since there are two paths, and the concatenation of two optimal paths may not be optimal.

2.3.2 CENTRAL

CENTRAL[23] is a complete algorithm for solving well-formed MAPD, a practical subclass of MAPD. It iterates the timestep and for each timestep, it assigns distinct target locations, which can be either tasks' pickup locations or safe parking locations, to agents. Then it calls CBS to plan agents to their target locations, firstly for agents whose target locations are pickup locations and secondly for agents assigned parking locations.

2.3.3 TA-hybrid

TA-hybrid[20] is also a complete algorithm for well-formed MAPD. Its task assignment procedures are similar to CENTRAL, while it proposed to use dummy paths during path planning. For agents assigned with pickup locations, they are planned with the path from their current locations, via the pickup locations to their safe parking locations, by replacing CBS's low-level with MLA*. And such paths are called dummy paths because agents will usually be assigned delivery locations after reaching pickup locations and their paths will be re-planned. The usage of the dummy path improves the algorithm performance and guarantees the completeness.

2.3.4 Goal Sequencing and Configurable Environments

Recent studies [35, 41, 29] have explored MAPF variants where each agent must visit multiple goals and computes the order in which it visits the goals. DD-MAPD also requires sequencing tasks, but the task locations must be computed dynamically. [3] studies a MAPF variant where the warehouse layout can be changed by a blackbox, but DD-MAPD requires computing a plan for agents to change the warehouse layout.

2.3.5 Plan Execution

MAPF and MAPD plans can be executed by real robots, using a dependency graph that respects the precedence constraints [12]. The plan execution is guaranteed to be collision-free, even with unmodeled kinematic constraints [11] or delay uncertainties [22]. Unlike our proposed framework for solving DD-MAPD, each agent has its own path in the computed plan and cannot execute other paths.

Chapter 3

Problem Definition

A DD-MAPD instance consists of N agents $a_1, \ldots, a_N$, M shelves $s_1, \ldots, s_M$, and a connected directed graph $G = (V, E)$, whose vertices V represent locations and edges E represent the connections between these locations that the agents can move along. We consider only interesting DD-MAPD instances where $M \geq N$ since, otherwise, one can use $N = M$ agents to each carry a unique shelf.

Let $\pi_i(t)$ denote the location of agent a_i at timestep t. Each agent a_i starts at its start location at timestep 0 and moves to an adjacent location or waits in its current location at each timestep. Each shelf s_j starts in its pickup location p_j at timestep 0 and is given a delivery location d_j. If a shelf s_j does not need to be relocated, then $p_j = d_j$. An agent can move beneath a shelf when not carrying any shelf, and it can lift a shelf when it is in the same location as the shelf, carry the shelf from then on, and place (put down) the shelf when it arrives in another location. Agents are *active* when they are carrying shelves, and *free* when they are not. We assume that the time required to perform a lift or place action is 0, but our framework can easily be generalized to accommodate lift and place actions with non-zero time costs. Let $\eta_j(t)$ denote the location of shelf s_j at timestep t. Shelves can only move when carried by agents. There should be no collisions either between agents (on the virtual low-level deck) or between shelves (on the virtual high-level deck). A vertex collision between agents a_i and $a_{i'}$ occurs iff $\pi_i(t) = \pi_{i'}(t)$; an edge collision occurs iff $\pi_i(t) = \pi_{i'}(t+1)$ and $\pi_i(t+1) = \pi_{i'}(t)$. Similarly, a vertex collision between shelves s_j and $s_{j'}$ occurs iff $\eta_j(t) = \eta_{j'}(t)$; an edge collision occurs iff $\eta_j(t) = \eta_{j'}(t+1)$ and $\eta_j(t+1) = \eta_{j'}(t)$.

The problem of DD-MAPD aims to compute collision-free paths for the agents to transport all shelves from their pickup locations to their delivery locations. The completion time of agent a_i is the earliest timestep when the agent has arrived in the last location of its path and stopped moving. We use two metrics to measure the effectiveness of a DD-MAPD algorithm: the *makespan*, defined as the maximum of the completion times of all agents, and the *flowtime*, defined as the sum of the completion times of all agents.

We use the DD-MAPD instance shown in Figure 3.1 as our running example. Figure 3.1 (Top) demonstrates a solution with makespan 7 and flowtime 14 ($= 7 + 7$).

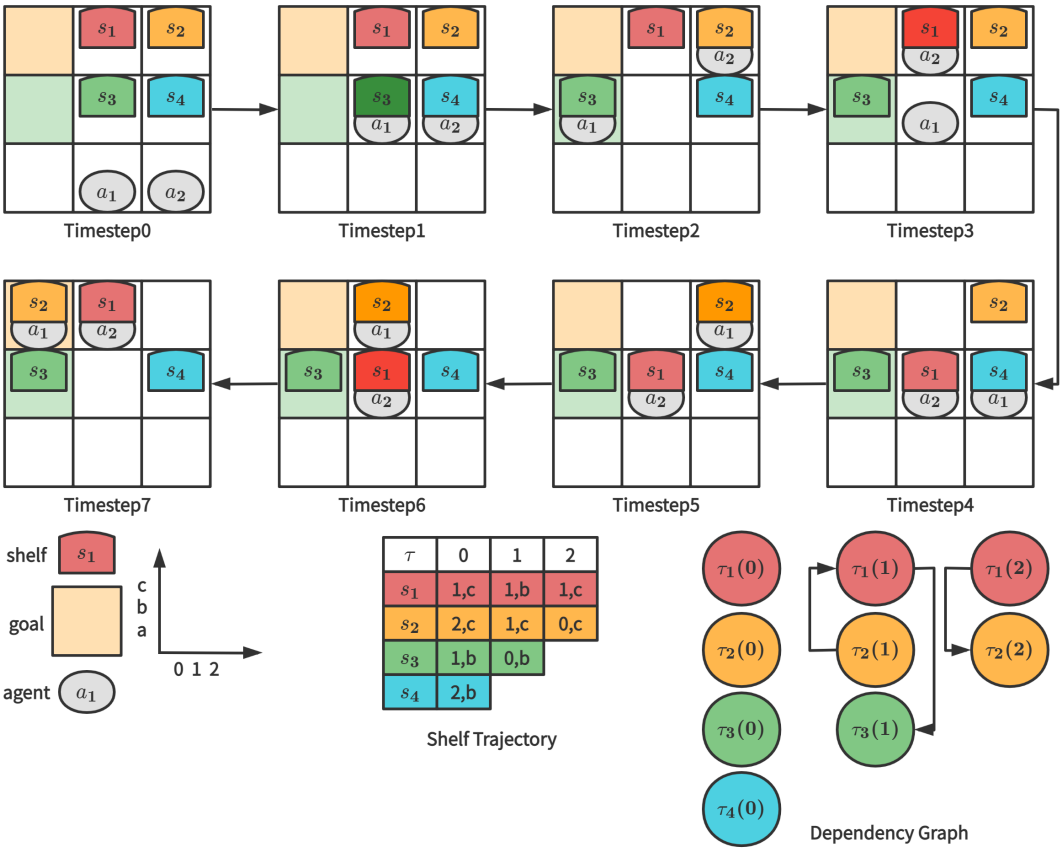

Figure 3.1: Example DD-MAPD instance with two agents and four shelves on a 2D 4-neighbor grid. Shelves s_2 and s_3 need to be relocated to the orange and green cells, respectively. Other shelves do not need to be relocated. Top: Locations of agents and shelves at each timestep. Bottom: Shelf trajectories and the resulting dependency graph.

Chapter 4

Complexity and Solvability

We now generalize existing complexity results for MAPD to DD-MAPD and identify sufficient conditions that make DD-MAPD instances solvable.

4.1 Complexity

We first show a constant-factor inapproximability result for DD-MAPD with respect to makespan minimization by reusing the reduction [25] from the NP-complete $\leq$3,=3-SAT problem [36] to MAPD, similar to that for MAPF [24]. The reduction in [25] proves that for any $\epsilon > 0$, it is NP-hard to find a $(4/3 - \epsilon)$ approximate solution to MAPD for makespan minimization.

Theorem 1. *For any $\epsilon > 0$, it is NP-hard to find a $(4/3 - \epsilon)$-approximate solution to DD-MAPD for makespan minimization.*

Proof Sketch. We use the same reduction as that used in the proof of Theorem 4.5 in [25] to construct a DD-MAPD instance with $M = N = 2\mathcal{N} + \mathcal{M}$ shelves and agents for a given $\leq$3,=3-SAT instance with $\mathcal{N}$ variables and $\mathcal{M}$ clauses. The arguments for MAPD still hold for DD-MAPD since each constructed agent carries only its corresponding constructed shelf. The constructed DD-MAPD instance has a solution with makespan three iff the $\leq$3,=3-SAT instance is satisfiable, and always has a solution with makespan four, even if the $\leq$3,=3-SAT instance is unsatisfiable. □

The constructed DD-MAPD instance in the above proof has the property that the completion time of each agent is at least three. Therefore, if the makespan is three, then every agent completes in exactly three timesteps, and the flowtime is $3N$ $(= 3(2\mathcal{N} + \mathcal{M}))$. Moreover, if the makespan exceeds three, then the flowtime exceeds $3M$, yielding the following corollary:

Corollary 2. *It is NP-hard to find an optimal solution to DD-MAPD for flowtime minimization.*

4.2 Baseline Complete Algorithm for Well-Formed Instances

We characterize a subclass of solvable DD-MAPD instances, called *well-formed* DD-MAPD instance, that generalize well-formed MAPD instances [23], even though we often need to solve non-well-formed DD-MAPD instances in practice. We first define that a MAPF solution for shelves (by treating shelves as agents that can move by themselves) is *1-robust* [1] iff, at any time step, a shelf is not allowed to (move to and) occupy a location at the next time step if the location is currently occupied. We note that the requirement for the existence of a 1-robust MAPF solution is not overly restrictive in practice since a sufficient condition for it is that at least two vertices of the MAPF graph are unoccupied [21].

Definition 1 (Well-formedness and safe 1-robustness)**.** A DD-MAPD instance is *well-formed* iff all agents start at different locations, graph G remains connected if the start locations of any $N-1$ agents are removed, and there exists a *safe* 1-robust MAPF solution for shelves, defined as one that does not use the start location of any agent.

We then sketch a baseline algorithm that is complete for all well-formed DD-MAPD instances as follows: It first generates trajectories for all shelves by computing a safe 1-robust MAPF solution for them; It then uses any single agent to execute all shelf trajectories in locked steps, namely proceeding to the next step only after executing one step of the MAPF solution for all shelves, while letting all other agents wait at their start locations. It is a straightforward observation that all well-formed DD-MAPD instances are solvable, and this baseline algorithm solves all of them. Our framework, MAPF-DECOMP, is similar to this baseline algorithm in that it first computes the shelf trajectories, but it differs in that it utilizes multiple agents to execute them.

Chapter 5

MAPF-DECOMP

Algorithm 2 shows the pseudocode of MAPF-DECOMP. The algorithm starts by calling a MAPF solver to compute collision-free trajectories τ for all shelves from their pickup locations to delivery locations [Line 1]. These trajectories represent the intended path for each shelf, but the actual path of each shelf is executed by the agents. To distinguish the trajectories from the actual paths of the shelves or agents, we refer to τ as *trajectories*.[1] Next, MAPF-DECOMP converts the trajectories into a dependency graph that implicitly partitions each trajectory into segments [Line 2]. Each segment represents a portion of the trajectory that can be executed by an agent by carrying the shelf and following the segment. Finally, MAPF-DECOMP solves a specialized MAPD instance with task dependencies by assigning shelves (segments) to agents and planning paths for them to complete all executable segments at each timestep where an agent changes from active to free or vice versa, while respecting the dependencies between segments [Lines 3-11]. Unlike existing MAPD algorithms, MAPF-DECOMP does not plan paths for active agents but lets them follow the planned trajectories of the shelves they carry. The paths of active agents are thus the unexecuted portion of the trajectories.

To solve the specialized MAPD instance resulting from the dependency graph, MAPF-DECOMP maintains the current state of each agent a_i in the variable $States_i$ that consists of two attributes: $States_i.type$ is either *free* or *active* and initially set to *free*; $States_i.shelf$ is the shelf assigned to agent a_i and initially set to *null* [Line 4]. In addition, each shelf s_j is assigned a step $s_j.step = k$, which represents its current location $\tau_j(k)$ according to its trajectory τ_j. The step $s_j.step$ is initially set to 0, corresponding to the pickup location $\tau_j(0) = p_j$ [Line 6]. We recall that a shelf is completed if it has arrived at its delivery location and remains there. At each timestep where there are uncompleted shelves, MAPF-DECOMP updates the states of all agents based on the dependency graph and the steps

[1]This thesis adopts a non-standard use of the terms "paths" and "trajectories" by reversing their conventional definitions. The purpose of this is to maintain consistency with the usage of "paths" in the MAPF literature, whereas in robotics, a "trajectory" typically refers to the path/executed followed by an agent as a function of time.

Algorithm 2: MAPF-DECOMP

```
1  τ ← trajectories of shelves by MAPF;
2  G = (V, E) ← BuildDep();
3  foreach agent a_i do
4      States_i.type ← free, States_i.shelf ← null;
5  foreach shelf s_j do
6      Steps_j ← 0;
7  while there are uncompleted shelves do
8      Update(States, G);
9      if States has changed then
10         FreePaths ← AssignAndPlan(States);
11     System proceeds to the next timestep;
12     Move(States, τ, FreePaths);
```

Algorithm 3: BuildDep($\tau, \mathcal{G}$)

```
1  foreach vertex pairs τ_j(k), τ_j'(k') in τ with k > k' do
2      if j ≠ j' and τ_j(k) = τ_j'(k') then
3          AddEdge(τ_j(k), τ_j'(k' + 1));
```

of all shelves [Line 8]. If any agent changes its state, MAPF-DECOMP (re)assigns shelves or safe locations to free agents and calls a MAPF solver to compute collision-free paths for them to reach their assigned destinations [Line 10]. The algorithm then proceeds to the next timestep and moves all agents and the shelves they carry to their next locations [Lines 11-12].

5.1 Building Dependency Graph

Given the trajectories τ as the result of calling a MAPF solver, MAPF-DECOMP calls Function BuildDep() to construct a dependency graph. The trajectories τ define a total order on their entries $\tau_j(k)$, which the dependency graph relaxes to a partial order. The resulting dependency graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a directed graph whose vertices $\mathcal{V}$ are the trajectories entries $\tau_j(k)$. For all j, j', k, k' with $j \neq j'$, $k > k'$, and location $\tau_j(k) = \tau_{j'}(k')$, an edge $\langle \tau_j(k), \tau_{j'}(k'+1) \rangle$ exists, which represents that shelf s_j should be in step k, namely the location specified by $\tau_j(k)$ (that follows $\tau_j(k)$, which specifies the same location as $\tau_{j'}(k')$), no earlier than shelf $s_{j'}$ is in step $k'+1$, namely the location specified $\tau_{j'}(k'+1)$. That is, whether an agent is allowed to execute the segment starting from step k or not depends on whether shelf $s_{j'}$ has been in step $k'+1$. By construction, cycles can only exist for entries $\tau_j(k)$ with the same step k. The above construction of the dependency graph is similar to the one used in the previous work [11, 22] and guarantees that the execution of the collision-free trajectories of the shelves is also collision-free. It differs from the one used in the previous work in not constructing edges between entries of the same trajectory. Figure 3.1 shows the dependency graph for our running example and Figure 5.1 shows how the dependency

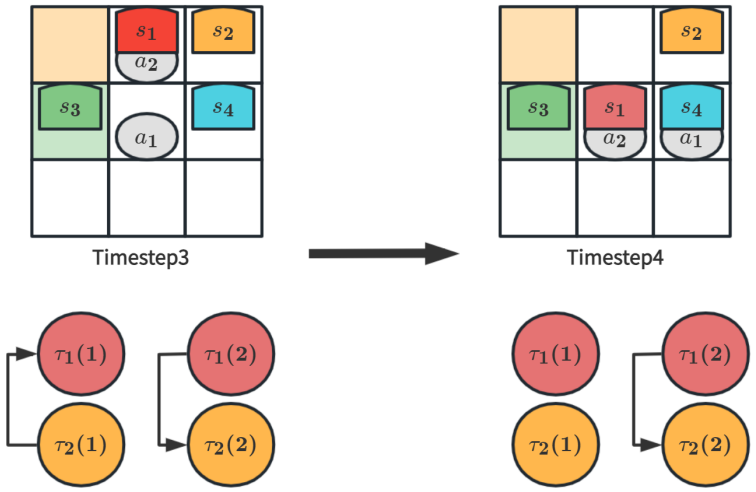

Figure 5.1: Usage of the dependency graph

graph constrains the movement of shelves: In timestep 3, there is an edge directed from $\tau_2(1)$ to $\tau_1(1)$, prohibiting s_2 from executing its next movement, which is moving left. And in timestep 4, since s_1 is moved for one step, the edge is removed and s_2 can move in the following timesteps.

5.2 Updating Agent States

At each timestep, MAPF-DECOMP calls Function Update() to update the states of agents. Update() serves three purposes: (1) It checks whether a free agent located at its assigned shelf's location can execute the shelf's trajectory. (2) It verifies if an active agent can continue executing the trajectory of its assigned shelf or if the shelf's trajectory is completed. (3) If the shelf trajectories are not 1-robust, the function checks whether the next trajectory entries of the assigned shelves of multiple agents form a cycle or path and whether the shelves can be executed simultaneously.

We define a (simple) *path* on $\mathcal{G}$ as a sequence of vertices that has an outgoing edge from each vertex in the sequence to its successor in the sequence, with no repeated vertices and edges. We define a (simple) *cycle* on $\mathcal{G}$ as a (simple) path except that the first and last vertices are the same.

Algorithm 4 shows the pseudocode of Update(). $\mathcal{A}_{sc}$ is the set of possible agents whose assigned shelves have dependencies that might be released simultaneously in the next step. $\mathcal{A}_{sc}$ is set to empty initially [Line 1]. For each agent a_i, *numDeps* stores the number of dependencies of the next step of its assigned shelf s_j [Line 4]. If $numDeps = 1$, let the only dependency be $\langle \tau_j(s_j.step+1), \tau_{j'}(k'+1) \rangle$ (with $\tau_j(s_j.step+1) = \tau_{j'}(k')$ and $step+1 > k'$). In this case, Function SoftDep() returns *true* iff $Steps_{j'} = k'$ (namely, shelf $s_{j'}$ is in step k' that specifies the same location as the next step of shelf s_j) and stores the result in the

Algorithm 4: Update($States, \mathcal{G}$)

```
1  A_sc ← ∅;
2  foreach agent a_i do
3      s_j ← States_i.shelf;
4      numDeps ← |outgoing edges of τ_j(s_j.step + 1) in G| ;
5      isSoftDep ← numDeps = 1 ? SoftDep(States_i.shelf, G) : false;
6      if States_i.type = free and agent a_i at the same location as States_i.shelf then
7          if isSoftDep then
8              A_sc ← A_sc ∪ a_i;
9          else if numDeps ≥ 1 then
10             States_i.shelf ← null;
11         else
12             States_i.type ← active;
13     else if States_i.type = active then
14         if States_i.shelf is completed then
15             States_i.shelf ← null;
16             States_i.type ← free;
17         else if isSoftDep then
18             A_sc ← A_sc ∪ a_i;
19         else if numDeps ≥ 1 then
20             States_i.shelf ← null;
21             States_i.type ← free;
22 A_noMove ← FindNoMove(A_sc);
23 foreach agent a_i ∈ A_sc do
24     if a_i ∈ A_noMove then
25         States_i.shelf ← null;
26         States_i.type ← free;
27     else
28         States_i.type ← active;
```

Boolean variable *isSoftDep*. In this case, we say that shelf s_j is *softly constrained*, or has a *soft dependency*, because both shelves s_j and $s_{j'}$ can move one step forward simultaneously, which includes cases where $\tau_j(s_j.step + 1)$ and $\tau_{j'}(k' + 1)$ are in a cycle or path. For each free agent a_i that is in the same location as its assigned shelf [Line 6], if the shelf has a soft dependency, then agent a_i is added to $\mathcal{A}_{sc}$ [Lines 7-8]. Otherwise, if the shelf has dependencies (that are thus *hard*), then it is unassigned from agent a_i [Lines 9-10]. Otherwise, the shelf has no dependency, and agent a_i changes from free to active and starts executing the trajectory of the shelf [Lines 11-12]. All other free agents do not change their states (remain free). For each active agent a_i, if its assigned shelf is completed, then the shelf is unassigned from it, and it changes from active to free [Lines 13-16]. Otherwise, if the shelf has a soft dependency, then agent a_i is added to $\mathcal{A}_{sc}$ [Lines 17-18]. Otherwise, if the shelf has dependencies (that are thus hard), then it is unassigned from agent a_i, and the agent changes from active to free [Lines 19-21]. All other active agents do not change their states and continue executing the trajectories of the shelves they carry.

Function Update() then calls the procedure FindNoMove() to identify the set $\mathcal{A}_{noMove}$ of any agents in $\mathcal{A}_{sc}$ that cannot move to the locations specified by the next step of the trajectories of their assigned shelves. We recall that each such shelf s_j is softly constrained, namely, its next trajectory entry depends on the next trajectory entry of another shelf $s_{j'}$ (not necessarily assigned to an agent in $\mathcal{A}_{sc}$) and it can thus move one step only no earlier than shelf $s_{j'}$ has moved one step. FindNoMove() considers the subgraph $\mathcal{G}_{sc}$ of $\mathcal{G}$ induced by the next trajectory entries of shelves assigned to all agents in $\mathcal{A}_{sc}$. Each such trajectory entry is in either a cycle or a path on $\mathcal{G}_{sc}$ since it has one outgoing edge in $\mathcal{G}_{sc}$ except for the case that it is the last vertex on some path and its (only) outgoing edge in $\mathcal{G}$ points to a trajectory entry of some shelf $s_{j'}$ that is not assigned to any agent in $\mathcal{A}_{sc}$ (the edge thus does not belong to $\mathcal{G}_{sc}$). In this case, if shelf $s_{j'}$ is assigned to either a free agent (not in $\mathcal{A}_{sc}$) or no agent at all, then the outgoing edge (dependency) is not released since shelf $s_{j'}$ is not carried by any agent, The shelves with their next trajectory entries on this path cannot move, and the agents that they are assigned to are thus added to $\mathcal{A}_{noMove}$.

5.3 Shelf Assignment and Path Planning

MAPF-DECOMP calls Function AssignAndPlan() if Update() changes an agent's type from free to active or from active to free and unassigns its assigned shelf. AssignAndPlan() then assigns *executable* shelves, namely ones that are not constrained at their current steps, to free agents and plan their paths, unless the shelves are already carried by active agents. The function operates in multiple rounds, with the aim of making more shelves executable in each subsequent round, as constraints are lifted by the paths planned in the previous round. To do so, AssignAndPlan() procedure follows these steps in each round: (1) It constructs a candidate set of unassigned and executable shelves. (2) If no such shelves exist, the function simulates Function Move() that lets all agents (including the free agents that got assigned shelves in the previous rounds and all active agents) follow their paths. During simulation, each shelf moves together with its assigned active agent until it reaches a hard dependency in its next trajectory entry. AssignAndPlan() then identifies any newly executable unassigned shelves and adds them to the candidate set. (3) If no such shelves become executable from the simulation, the function identifies all unassigned shelves whose next trajectory entries form a cycle in $\mathcal{G}$ and adds them to the candidate set since they can be executed simultaneously. Once the candidate set consists of one or more shelves, AssignAndPlan() calls the Hungarian algorithm [13] to find a minimal-cost assignment of shelves to free agents and plans their paths. The cost of assigning a shelf to an agent is calculated as the maximum of the shortest-path distance between the agent and the shelf and the timestep when the shelf first becomes executable. AssignAndPlan() then calls a MAPF solver to compute paths for these agents from their current locations to the current locations of their assigned shelves, avoiding collision with paths of active agents and any paths of free agents

Algorithm 5: Move($States, \tau, FreePaths$)

```
1 foreach agent a_i do
2     if States_i.type = active then
3         s_j ← States_i.shelf;
4         Update the location of a_i and s_j according to τ_j;
5         s_j.step ← s_j.step + 1;
6         Remove incoming edges of τ_j(s_j.step) from G;
7     else
8         Update the location of a_i according to FreePaths_i;
```

already planned in the previous rounds. When all executable shelves have been assigned, for each free agent that remains unassigned, the function assigns it a unique location closest to it and not on any active agent's path. The function then calls a MAPF solver to compute paths for these unassigned agents, avoiding collisions with the paths of all other agents.

5.4 Moving Agents and Shelves

When the system proceeds to the next timestep, MAPF-DECOMP calls Function Move() to move agents and shelves one step forward. Algorithm 5 shows the pseudocode of Move(). Each active agent a_i and the shelf it carries move one step according to the trajectory τ_j of the shelf [Lines 4-5], which, as a result, releases the dependencies of the corresponding entry of τ_j (by removing all incoming edges of the entry from $\mathcal{G}$) [Line 6]. Each free agent moves one step according to $FreePaths_i$ [Line 8].

5.5 Running Example

Figure 3.1 (Top) shows the execution of the shelf trajectories for our running example. For ease of presentation, we point out only the most insightful details. At timestep 0, a_1 and a_2 are assigned s_3 and s_1 since s_2 is not executable in Round 1. At timestep 1, a_1 changes from free to active. At timestep 2, a_1 completes s_3 and becomes free. In Round 1, only s_1 is executable and is assigned to a_2. In Round 2, s_2 is assigned to a_1. At timesteps 4 and 5, Update() sets a_2 to free even though it is in the same location as its assigned shelf s_1 since $\langle \tau_1(2), \tau_2(2) \rangle$ is a hard constraint. At timestep 6, Update() adds only a_2 to $\mathcal{A}_{sc}$ and changes it to active since $\langle \tau_1(2), \tau_2(2) \rangle$ is a soft constraint.

5.6 Optimization: Involving Future (IVF)

We now present an optimization technique called Involving Future (IVF) that aims to enhance the effectiveness of the shelf assignment procedure in MAPF-DECOMP. IVF improves AssignAndPlan() by considering not only the currently free agents but also those that will become free in K timesteps for shelf assignment. This allows for a larger pool of agents to

be considered for shelf assignment, potentially resulting in a better overall assignment. To achieve this, IVF simulates both Update() and Move() for K iterations and includes active agents that change to free during the simulation in the shelf assignment and path planning process. The number of timesteps elapsed when each such agent becomes free is recorded and added to the shortest-path distance between the agent and any shelf when calculating the cost of assigning the shelf to the agent.

Chapter 6

Prioritized Planning (PP) for Completeness

MAPF-DECOMP is not guaranteed to solve all well-formed DD-MAPD instances due to two reasons: (1) If the shelf trajectories are not 1-robust, multiple shelves with soft dependencies may form a cycle (as detailed in Section 5.3), which cannot be resolved if the total number of agents is smaller than the number of shelves in the cycle. (2) Function AssignAndPlan() plans paths for free agents to locations different from the start locations of agents, which does not guarantee collision-free paths from those locations exist.

Thus, we propose a variant of MAPF-DECOMP, MAPF-DECOMP(PP), that is complete for all well-formed DD-MAPD instances. This variant differs from the original in the following ways: (1) It uses a MAPF solver to compute safe 1-robust shelf trajectories. (2) In each assignment round, AssignAndPlan() assigns only one shelf from the candidate set to an agent by selecting the pair with the smallest assignment cost, and then uses a multi-label A* search [10] to compute a time-minimal path for the agent, avoiding any collisions with the paths of other agents. The path first moves the agent from its current location to the current location of the shelf, then follows the trajectory of the shelf without waiting until the shelf is constrained, and finally moves the agent to its start location. Any free agent that remains unassigned when all executable shelves have been assigned keeps its current path that ends in its start location. Therefore, each agent maintains the invariant that its path always ends in its start location, inspired by the "reserving dummy paths" deadlock-avoidance technique for MAPD [20].

Theorem 3. *MAPF-DECOMP(PP) solves all well-formed DD-MAPD instances.*

Proof. Since the dependency graph for 1-robust trajectories is acyclic, Function AssignAndPlan() adds at least one unassigned shelf that is or (in simulation) will become executable to the candidate set and assigns it to a free agent. The multi-label A* search guarantees to find a collision-free path for the agent to execute the shelf trajectory segment since such a path always exists. For example, the agent can first follow its old path and wait at its start location until all other agents have completed their paths and stay at their start

locations indefinitely. Without passing through the start locations of other agents, the agent can then move to the current location of its assigned shelf, follow the trajectory of the shelf without waiting or causing shelf collisions until the shelf is constrained, and finally return to its start location. Thus, all shelf trajectories will eventually be executed since at least one additional trajectory segment is assigned and executed each time AssignAndPlan() is called. □

Chapter 7

Experiments

We conduct our experiments on a 3.1GHz Intel Core i5 laptop with 16GB RAM. We implement three variants of MAPF-DECOMP: MAPF-DECOMP without the IVF optimization, MAPF-DECOMP with IVF, and MAPF-DECOMP(PP) (labeled **`NIVF`**, **`IVF`**, and **`PP`**, respectively). They all use EECBS [19] to compute both trajectories for shelves and paths for agents and are implemented in C++. We adapt EECBS to compute safe 1-robust shelf trajectories for PP. We label IVF executions that use 1-robust shelf trajectories returned by EECBS as **`IVF-R`**. The suboptimality factor ω of EECBS ranges from 1.2 to 1.8 for different settings to balance the effectiveness and efficiency. We also implement two baseline algorithms: one that uses a single agent to execute the 1-robust shelf trajectories returned by EECBS in locked timesteps (labeled **`BASE`**), and another that used a single agent to execute 1-robust shelf trajectories returned by Push and Swap [21] (labeled **`PAS`**). Unlike the trajectories returned by EECBS, the sequential 1-robust shelf trajectories returned by Push and Swap often have segments longer than one step, and PAS interprets the trajectories as a total order on the segments.

7.1 MAPF-DECOMP with IVF

We construct DD-MAPD instances on $n \times n$ square 2D 4-neighbor grids of different sizes (labeled **`size`** n) by randomly sampling blocks of 2×2 cells as pickup locations of shelves until the pickup locations reach a certain density (percentage of all cells, labeled **`den`**). We sample $0.1 \cdot n^2$ shelves from all the shelves as the only ones that need relocation and sample their delivery locations from non-pickup cells. We randomly sample the start locations of the agents. We evaluate 50 random instances per setting and report the mean over all solved instances.

Optimization and numbers of future timesteps. Figure 7.1 and Figure 7.2 show the results for IVF when different values of K (numbers of future timesteps) are used, where $K = \inf$ means that all active agents are included in the shelf assignment. The makespan tends to be smaller for larger K. $K = \inf$ does not necessarily result in the

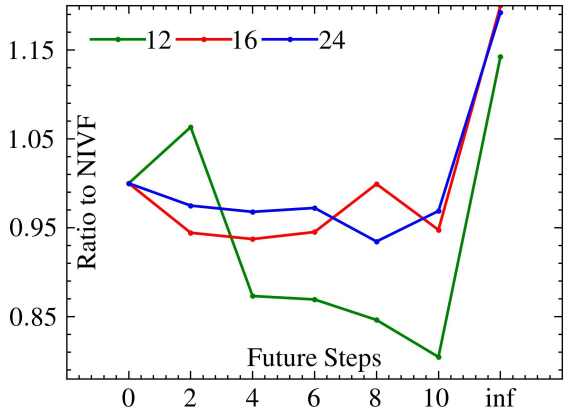

Figure 7.1: Agent time ratios of IVF against NIVF for different K. Legends show grid sizes.

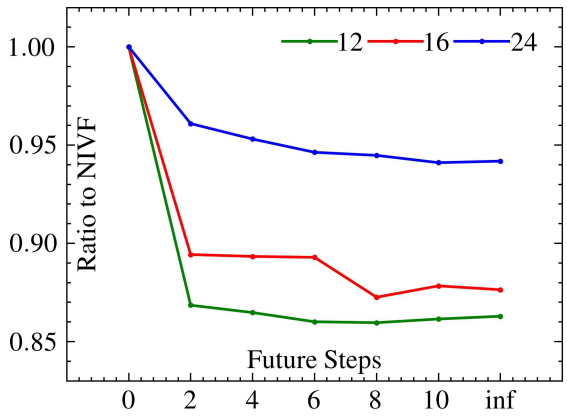

Figure 7.2: Makespan ratios of IVF against NIVF for different K. Legends show grid sizes.

smallest makespan, where the last call of EECBS returns paths for a few agents that are much longer than for other paths since EECBS optimizes only the flowtime. As K grows, the runtime excluding shelf trajectory computation (labeled **`agent time`**) (1) first drops since the makespan also drops and fewer calls to EECBS are made and (2) can then go up since each call to EECBS involves more agents. We use $K = 8$ for IVF in all the following experiments to balance the runtime and the effectiveness. IVF (with $K > 0$) is always more effective than NIVF due to better shelf assignments as a result of involving more agents.

Grid sizes, agent numbers, and shelf densities. Table 7.1 (Top) shows that IVF solves all instances in seconds for small numbers of agents and that doubling the number of agents sometimes increases the flowtime, which sums up the completion times of all agents, but always reduces the makespan significantly. The agent times are large for large makespans due to the large numbers of calls of EECBS by AssignAndPlan(). Table 7.1 (Middle) shows that IVF achieves high success rates (labeled **`succ`**) for hundreds of agents and more than one thousand shelves. The total runtimes remain in a few minutes even though the task-assignment and path-planning functions must be executed many times for makespans of

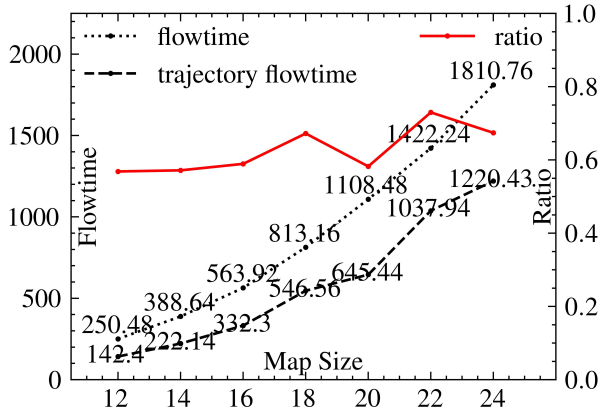

Figure 7.3: Results for IVF (den=20%): flowtime ratios against trajectory flowtimes for 4 agents and different sizes

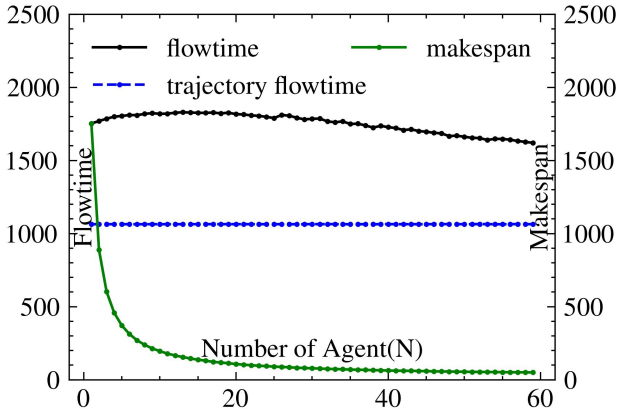

Figure 7.4: Results for IVF (den=20%): effectiveness for different numbers of agents and size 24.

thousands of timesteps. Table 7.1 (Bottom) categorizes the reasons for failed instances: (a) A timeout (1 min) for EECBS to compute shelf trajectories (typically for very large M); (b) Not enough agents to simultaneously move all shelves in a soft dependency cycle (typically for large M and small N), which can be addressed by computing a 1-robust MAPF solution for shelves or using more agents by setting N to be the number of shelves in the largest cycle; (c) The incompleteness of path planning for free agents in AssignAndPlan() (typically for very large N), which can be addressed by using deadlock-avoidance techniques if the given DD-MAPD instance is well-formed.

Effectiveness. Figure 7.3 shows that the flowtime of the shelf trajectories (the sum of all trajectory lengths), which is a (trivial) lower bound on the flowtime and for which MAPF-DECOMP does not optimize, consistently contributes to a large portion ($\geq$60%) of the flowtime across different instance sizes, which indicates that the decomposition and the execution of the trajectories of our framework are both effective. Figure 7.4 shows that to

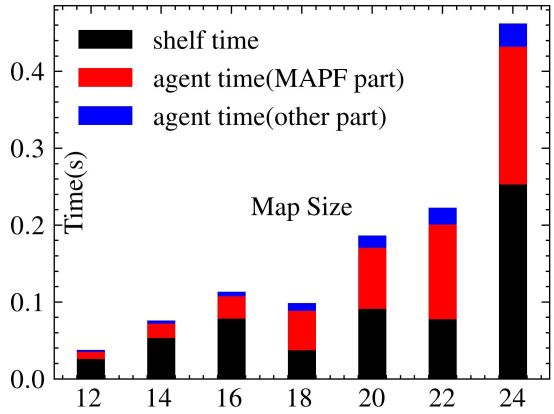

Figure 7.5: Results for IVF (den=20%): time breakdown for different sizes and 8 agents.

execute the same trajectory, as the number of agents increases, the flow time drops for large numbers of agents and the makespan drops in all cases. This is so because shelves are often assigned to close-by agents when there are many agents.

Runtime breakdown. Figure 7.5 confirms that the runtime used to compute the trajectories of shelves (labeled **`shelf time`**) contributes to a large portion of the total runtime. It also suggests that an improvement in the efficiency of the MAPF solver would directly result in an improvement in the efficiency of our framework since most of the runtime is used by the MAPF solver.

7.2 Comparison of Algorithms on Well-Formed Instances

We follow a similar procedure to construct random DD-MAPD instances, except that we do not place shelves along the perimeter of the grids but sample start locations of agents from cells on the perimeter, excluding the corners, to guarantee well-formedness. Table 7.2 shows that IVF-R with one agent is more effective than the two baselines and that using more agents results in further improvement. For example, the makespan for 100 agents is only 1.1% of that for a single agent for instances of size 96. IVF-R and IVF tend to be more effective but are less efficient than PP since PP assigns tasks and plans paths for agents one at a time. PP solves all well-formed instances as expected. IVF-R has higher success rates than IVF since it does not fail for Reason (b) observed in the previous experiments but still fails for three instances in total due to the incompleteness of AssignAndPlan(). All algorithms do not time out with a one-minute runtime limit for each call to the MAPF solver. Overall, PP appears to be the the optimal choice of algorithm in practice if well-formedness is guaranteed since it strikes a good balance between efficiency and effectiveness.

7.3 Warehouse Rearrangement Demo

We construct 50 well-formed instances on 2D 4-neighbor grids of size 27×27, with 32 agents starting along the perimeter, based on the layout of a fulfillment center comprising 8×4 blocks of 5×2 shelves. The delivery locations of these 320 shelves are arranged in a diagonally symmetrical configuration and randomly shuffled. IVF-R successfully solves all instances with an average total time of 30.68s and an average agent time of 4.85s using EECBS with $\omega = 1.8$. A demo video showcasing the execution on one of the instances is available at: `https://youtu.be/WFPl3wKDXXY`.

size	den	M	N	makespan	flowtime	total time (s)	agent time (ms)	ω	succ
8	40%	25	4	31.10	109.32	1.81	2.82	1.2	100%
10		40	4	55.02	204.10	3.41	5.83	1.4	
			8	31.83	209.26	3.39	8.97		
12		57	4	89.18	339.53	0.30	11.81	1.6	
			8	50.27	347.39	0.31	17.52		
16		102	8	105.68	772.60	0.92	55.83	1.8	
16	20%	51	4	140.98	534.94	0.10	22.51	1.2	100%
			8	74.34	530.60	0.11	34.82		
24		115	4	452.69	1,775.31	0.38	134.82	1.2	
			8	236.65	1,786.88	0.46	209.49		
32		204	4	1,072.18	4,247.16	0.67	487.65	1.4	
			8	546.39	4,249.12	0.92	737.61		
40		320	8	1,072.35	8,444.20	2.62	2,119.00	1.6	
large-size instances, agent time reported in seconds							(s)		
48	20%	460	8	1,850.22	14,629.29	6.24	5.04	1.6	98%
			32	487.78	14,578.45	13.09	11.83		98%
64		819	8	4,451.22	35,425.22	32.02	25.68	1.8	98%
			32	1,148.41	35,476.73	106.00	99.26		98%
			100	397.68	33,648.55	88.95	82.57		94%
			400	153.31	26,969.08	182.46	176.72		96%
96		1,843	32	3,909.00	123,383.24	302.92	270.27	1.8	92%
			100	1,264.05	118,965.28	1,060.16	1,019.93		80%

reasons for failed large-size instances

size	den	M	N	instances	(a) timeout	(b) small N	(c) incomplete
48	20%	460	8	1/50	0	1	0
			32	1/50	0	1	0
64		819	8	1/50	0	1	0
			32	1/50	0	1	0
			100	3/50	0	3	0
			400	2/50	0	0	2
96		1,843	32	4/50	1	3	0
			100	10/50	8	2	0
sum				23/400	9	8	2

Table 7.1: Results for IVF in different settings.

size	M	N	algo.	makespan	flowtime	total time (s)	agent time (s)	ω	succ
48	460	1	BASE	190,243.52	-	1.20	-	1.6	100%
			PAS	14,707.44	-	2.31	-		100%
			IVF-R	13,439.26	-	2.55	1.35		100%
		8	PP	1,757.33	13,910.77	2.11	0.93		100%
			IVF-R	1,750.96	13,862.56	6.35	5.20		100%
			IVF	1,745.44	13,812.10	6.86	5.80		96%
		32	PP	463.71	13,836.77	2.94	1.58		100%
			IVF-R	460.75	13,705.56	10.68	9.48		100%
			IVF	461.35	13,740.19	10.05	9.13		96%
64	819	1	BASE	605,647.28	-	4.34	-	1.8	100%
			PAS	34,642.68	-	9.17	-		100%
			IVF-R	32,560.80	-	9.06	4.72		100%
		8	PP	4,258.04	33,877.76	7.66	3.18		100%
			IVF-R	4,254.92	33,839.10	25.64	21.01		100%
			IVF	4,256.45	33,865.43	24.43	20.98		98%
		32	PP	1,105.00	34,090.16	10.69	5.38		100%
			IVF-R	1,099.69	33,943.73	41.29	36.64		100%
			IVF	1,096.71	33,835.06	40.50	37.01		98%
		100	PP	384.81	32,695.28	17.78	11.00		100%
			IVF-R	381.58	32,419.40	63.19	58.75		98%
			IVF	384.00	32,530.45	66.11	62.59		98%
96	1,843	1	BASE	3,027,935.92	-	37.13	-	1.8	100%
			PAS	112,282.98	-	71.02	-		100%
			IVF-R	111,258.48	-	72.10	34.97		100%
		32	PP	3,817.24	120,368.60	79.49	33.06		100%
			IVF-R	3,817.67	120,410.38	301.82	264.46		98%
			IVF	3,807.28	120,134.72	294.06	271.03		92%
		100	PP	1,235.37	115,719.63	123.86	64.49		100%
			IVF-R	1,228.53	115,274.80	623.54	585.69		98%
			IVF	1,229.35	115,037.24	708.07	684.56		92%

Table 7.2: Results on large-size well-formed instances.

Chapter 8

Conclusions and Future Work

In this thesis, we propose a novel problem formulation, DD-MAPD, and a new algorithmic framework, named MAPF-DECOMP for solving DD-MAPD, to make MAPF and MAPD applicable to multi-robot shelf rearrangement problems in large-scale warehouses. MAPF-DECOMP decouples the problem by firstly generating collision-free trajectories for shelves, and secondly planning paths for agents to carry shelves following the planned trajectories. In this thesis, we focus on the efficiency of our framework without sacrificing much of its effectiveness. Experiment results show that this framework can generate high-quality solutions for 1,843 shelves and 400 agents in minutes. We also propose a variant of MAPF-DECOMP that can solve all well-formed DD-MAPD instances efficiently.

We propose two future extensions to our framework: (1) We intend to improve its effectiveness by making its MAPF solving for shelf trajectories aware of the subsequent decomposition and planning. (2) We propose to plan paths for active agents instead of letting them follow shelf trajectories. (3) We may accelerate large-scale instances' execution by splitting agents and tasks into multiple groups by their start locations, and parallelize the computation of the whole algorithm framework.

Bibliography

[1] Dor Atzmon, Roni Stern, Ariel Felner, Glenn Wagner, Roman Barták, and Neng-Fa Zhou. Robust multi-agent path finding and executing. *Journal of Artificial Intelligence Research*, 67:549–579, 2020.

[2] Jacopo Banfi, Nicola Basilico, and Francesco Amigoni. Intractability of time-optimal multirobot path planning on 2d grid graphs with holes. *IEEE RA-L*, 2(4):1941–1947, 2017.

[3] Matteo Bellusci, Nicola Basilico, and Francesco Amigoni. Multi-agent path finding in configurable environments. In *AAMAS*, pages 159–167, 2020.

[4] Eli Boyarski, Ariel Felner, Roni Stern, Guni Sharon, David Tolpin, Oded Betzalel, and Solomon Eyal Shimony. ICBS: Improved conflict-based search algorithm for multi-agent pathfinding. In *IJCAI*, pages 740–746, 2015.

[5] Kyle Brown, Oriana Peltzer, Martin A Sehr, Mac Schwager, and Mykel J Kochenderfer. Optimal sequential task assignment and path finding for multi-agent robotic assembly planning. In *ICRA*, pages 441–447, 2020.

[6] M. Erdmann and T. Lozano-Perez. On multiple moving objects. In *Proceedings. 1986 IEEE International Conference on Robotics and Automation*, volume 3, pages 1419–1424, 1986.

[7] Ariel Felner, Jiaoyang Li, Eli Boyarski, Hang Ma, Liron Cohen, TK Satish Kumar, and Sven Koenig. Adding heuristics to conflict-based search for multi-agent path finding. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 28, pages 83–87, 2018.

[8] Graeme Gange, Daniel Harabor, and Peter Stuckey. Lazy cbs: Implicit conflict-based search using lazy clause generation. In *ICAPS*, pages 155–162, 2019.

[9] Rodrigo N. Gómez, Carlos Hernández, and Jorge A. Baier. Solving sum-of-costs multi-agent pathfinding with answer-set programming. In *AAAI*, pages 9867–9874, 2020.

[10] Florian Grenouilleau, Willem-Jan van Hoeve, and John N Hooker. A multi-label A* algorithm for multi-agent pathfinding. In *ICAPS*, pages 181–185, 2019.

[11] W. Hönig, T. K. S. Kumar, L. Cohen, H. Ma, H. Xu, N. Ayanian, and S. Koenig. Multi-agent path finding with kinematic constraints. In *ICAPS*, pages 477–485, 2016.

[12] Wolfgang Hönig, Scott Kiesel, Andrew Tinka, Joseph W Durham, and Nora Ayanian. Persistent and robust execution of mapf schedules in warehouses. *IEEE RA-L*, 4(2):1125–1131, 2019.

[13] H. W. Kuhn and Bryn Yaw. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, pages 83–97, 1955.

[14] Edward Lam, Pierre Le Bodic, Daniel Harabor, and Peter. Stuckey. Branch-and-cut-and-price for multi-agent pathfinding. In *IJCAI*, pages 1289–1296, 2019.

[15] Jiaoyang Li, Zhe Chen, Daniel Harabor, Peter J Stuckey, and Sven Koenig. Mapf-lns2: fast repairing for multi-agent path finding via large neighborhood search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 10256–10265, 2022.

[16] Jiaoyang Li, Graeme Gange, Daniel Harabor, Peter J Stuckey, Hang Ma, and Sven Koenig. New techniques for pairwise symmetry breaking in multi-agent path finding. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, pages 193–201, 2020.

[17] Jiaoyang Li, Daniel Harabor, Peter. Stuckey, Ariel Felner, Hang Ma, and Sven Koenig. Disjoint splitting for conflict-based search for multi-agent path finding. In *ICAPS*, pages 279–283, 2019.

[18] Jiaoyang Li, Daniel Harabor, Peter. Stuckey, Hang Ma, Graeme Gange, and Sven Koenig. Pairwise symmetry reasoning for multi-agent path finding search. *Artifical Intelligence*, 301:103574, 2021.

[19] Jiaoyang Li, Wheeler Ruml, and Sven Koenig. EECBS: Bounded-suboptimal search for multi-agent path finding. In *AAAI*, pages 12353–12362, 2021.

[20] Minghua Liu, Hang Ma, Jiaoyang Li, and Sven Koenig. Task and path planning for multi-agent pickup and delivery. In *AAMAS*, pages 2253–2255, 2019.

[21] R. Luna and K. E. Bekris. Push and Swap: Fast cooperative path-finding with completeness guarantees. In *IJCAI*, pages 294–300, 2011.

[22] H. Ma, T. K. S. Kumar, and S. Koenig. Multi-agent path finding with delay probabilities. In *AAAI*, pages 3605–3612, 2017.

[23] H. Ma, J. Li, T. K. S. Kumar, and S. Koenig. Lifelong multi-agent path finding for online pickup and delivery tasks. In *AAMAS*, pages 837–845, 2017.

[24] H. Ma, C. Tovey, G. Sharon, T. K. S. Kumar, and S. Koenig. Multi-agent path finding with payload transfers and the package-exchange robot-routing problem. In *AAAI*, pages 3166–3173, 2016.

[25] Hang Ma. *Target assignment and path planning for navigation tasks with teams of agents*. PhD thesis, University of Southern California, 2020.

[26] Hang Ma, Daniel Harabor, Peter Stuckey, Jiaoyang Li, and Sven Koenig. Searching with consistent prioritization for multi-agent path finding. In *AAAI*, pages 7643–7650, 2019.

[27] V. Nguyen, P. Obermeier, T. C. Son, T. Schaub, and W. Yeoh. Generalized target assignment and path finding using answer set programming. In *IJCAI*, pages 1216–1223, 2017.

[28] Keisuke Okumura, Manao Machida, Xavier Défago, and Yasumasa Tamura. Priority inheritance with backtracking for iterative multi-agent path finding. In *IJCAI*, pages 535–542, 2019.

[29] Zhongqiang Ren, Sivakumar Rathinam, and Howie Choset. Conflict-based steiner search for multi-agent combinatorial path finding. In *RSS*, 2022.

[30] Stuart J Russell. *Artificial intelligence a modern approach*. Pearson Education, Inc., 2010.

[31] G. Sharon, R. Stern, A. Felner, and N. Sturtevant. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, 219:40–66, 2015.

[32] Roni Stern, Nathan Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, T. K. Satish Kumar, Eli Boyarski, and Roman Bartak. Multi-agent pathfinding: Definitions, variants, and benchmarks. In *SoCS*, pages 151–159, 2019.

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

[34] Pavel Surynek. Unifying search-based and compilation-based approaches to multi-agent path finding through satisfiability modulo theories. In *IJCAI*, pages 1177–1183, 2019.

[35] Pavel Surynek. Multi-goal multi-agent path finding via decoupled and integrated goal vertex ordering. In *AAAI*, pages 12409–12417, 2021.

[36] C. Tovey. A simplified NP-complete satisfiability problem. *Discrete Applied Mathematics*, 8:85–90, 1984.

[37] G. Wagner and H. Choset. Subdimensional expansion for multi-robot path planning. *Artificial Intelligence*, 219:1–24, 2015.

[38] P. R. Wurman, R. D'Andrea, and M. Mountz. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI Magazine*, 29(1):9–20, 2008.

[39] J. Yu. Intractability of optimal multi-robot path planning on planar graphs. *IEEE RA-L*, 1(1):33–40, 2016.

[40] J. Yu and S. LaValle. Structure and intractability of optimal multi-robot path planning on graphs. In *AAAI*, pages 1444–1449, 2013.

[41] Han Zhang, Jingkai Chen, Jiaoyang Li, Brian Williams, and Sven Koenig. Multi-agent path finding for precedence-constrained goal sequences. In *AAMAS*, pages 1464–1472, 2022.