Optimal and Bounded-Suboptimal Multi-Goal Task Assignment and Pathfinding

by

Xinyi Zhong

B.Sc., Carleton University, 2019

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
Fall 2021

Declaration of Committee

Name:	**Xinyi Zhong**
Degree:	**Master of Science**
Thesis title:	**Optimal and Bounded-Suboptimal Multi-Goal Task Assignment and Pathfinding**
Committee:	**Chair:** Angelica Lim Assistant Professor, Computing Science
	Hang Ma Supervisor Assistant Professor, Computing Science
	Mo Chen Committee Member Assistant Professor, Computing Science
	David G. Mitchell Examiner Associate Professor, Computing Science

Abstract

We formalize and study the multi-goal task assignment and pathfinding (MG-TAPF) problem from both theoretical and algorithmic perspectives. The MG-TAPF problem is to compute an assignment of tasks to agents, where each task consists of a sequence of goal locations, and plan collision-free paths for agents that visit all goal locations of their assigned tasks in sequence. Theoretically, we prove that the MG-TAPF problem is NP-hard to solve optimally. We present algorithms that build upon algorithmic techniques for the multi-agent pathfinding problem and solve the MG-TAPF problem optimally and bounded-suboptimally. We experimentally compare these algorithms on a variety of different benchmark domains.

Keywords: Multi-Agent Pathfinding, Task Assignment and Pathfinding, Multi-Goal Tasks, Multi-Robot System

Acknowledgements

First of all, I would like to pay my special regards to my senior supervisor Dr.Hang Ma, for introducing me to the areas of task assignment and pathfinding, and for providing technical supports and patience throughout my Master's study. This project 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. David Mitchell and my chair Dr. Angelica Lim for devoting time to read my thesis and providing useful comments.

I would like to thank all the members of AIRob Lab: Qinghong Xu, Dingyi Sun, Baiyu Li, Qiushi Lin, Danoosh Chamani, with whom I had lots of fun and inspiring conversations.

In the end, I am grateful to my parents, Hua Gao and Chongwang Zhong, for their unbounded support and encouragement. I would also like to thank my boyfriend, Hongyang Dong, for his patience and support during my study.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In recent years, the multi-agent pathfinding (MAPF) problem [29] has been well-studied in artificial intelligence and robotics; and has many applications such as warehouse automation [36], autonomous traffic management [4], autonomous aircraft towing [22], and video games [14]. In the MAPF problem, each agent must move from its current location to pre-assigned goal location while avoiding collisions with other agents in a known environment.

The MAPF problem has recently been extended to many real-world settings [24, 16] where goal locations are not pre-assigned to agents. For example, in a modern automated warehouse (Figure 1.1), each warehouse robot (orange square) needs to pick up an inventory pod from its storage location (green cell), deliver it to the inventory stations (pink and purple squares) that request one or several products stored in it, and send it back to its storage location. Such automated warehouse systems often employ a task planner to determine a set of tasks consisting of a sequence of goal locations. The problem is then to assign these tasks to the warehouse robots and plan paths for them.

We thus formalize and study the multi-goal task assignment and pathfinding (MG-TAPF) problem, where as many tasks as agents are given and each task consists of a sequence of goal locations. The MG-TAPF problem is to compute a one-to-one assignment of tasks to agents and plan collision-free paths for the agents from their current locations to the goal locations of their assigned tasks such that each agent visits the goal locations in the correct order specified by its assigned task and the flowtime (sum of the timesteps when each agent has reached the last goal location of its assigned task and stops moving) is minimized.

1.1 Contributions

In this thesis, we study the general version of TAPF problem where each task consists of a sequence of multiple ordered goal locations, denoted as MG-TAPF problem. Our contributions are as follows:

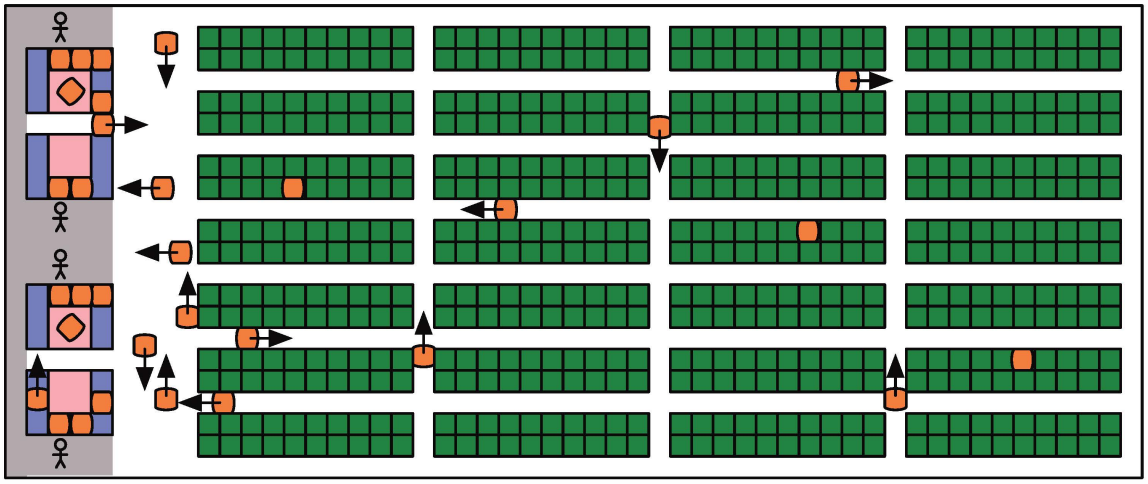

Figure 1.1: Layout of part of an Amazon automated warehouse, reproduced from [36].

1. We formalize the MG-TAPF problem as an extension of the MAPF problem that aims to minimize the flowtime. We prove that it is NP-hard to solve optimally, even for the case where each task only contains two goal locations. The proof is based on a reduction [20] from a specialized version of Boolean satisfiability problem [33] to the MG-TAPF problem.

2. We present an algorithm Conflict-Based Search with Task Assignment with Multi-Label A* (CBS-TA-MLA), that solves the MG-TAPF problem optimally for flowtime minimization. CBS-TA-MLA is a hierarchical algorithm. It uses CBS-TA [8], which is a best-first search algorithm, on the high level to search over all possible assignments of tasks to agents and resolve collisions among paths, and MLA* [7] on the low level to compute a time-optimal path for each agent that visits the goal locations of its assigned task. We prove that CBS-TA-MLA is correct, complete and optimal.

3. We develop three admissible heuristics for the high-level best-first search of CBS-TA-MLA based on the existing admissible heuristics [13] for CBS for the MAPF problem and generalize the Multi-Valued Decision Diagrams (MDDs) from the case of one goal location to the case of multiple goal locations. We also extend CBS-TA-MLA to a bounded-suboptimal version called ECBS-TA-MLA using ideas from the bounded-suboptimal version of CBS [1]. Finally, we experimentally compare the proposed algorithms in a variety of benchmark domains, showcasing their practicability for many real-world applications.

Chapter 2

Background

Many problems that are similar or related to our problem have been proposed and studied in recent years.

2.1 Pebble Motion on Graphs

The 15-puzzle problem [9] can be viewed as a special case of the MAPF problem where the graph is a 4×4 2D connected grid with 15 agents. The pebble motion problem [10] can be viewed as a generalization of the 15-puzzle problem where the number of agents (m) is at most $|V| - 1$. In the pebble motion on the graph problem, an agent can move from its current vertex to an adjacent vertex which is not occupied by another agent. The solution is the total number of edge traversals where each agent moves from its start vertex to its goal vertex [10, 25]. There exists a linear-time algorithm to decide whether an instance of the pebble motion on the graph problem is solvable, and an $O(|V|^3)$ algorithm for planning complete paths for the problem [6].

2.2 Multi-Agent Pathfinding

2.2.1 Problem Definition

The multi-agent pathfinding problem (MAPF) is a generalization of the single-agent pathfinding problem for the number of agents $m > 1$. It consists of

- an undirected graph $G = (V, E)$ where V corresponds to the set of locations and E corresponds to the unit-weight edges connecting locations that agents can move along;
- a set of m agents $\{a_1, \cdots, a_m\}$, and for each agent a_i, there is a start location $s_i \in V$ and a pre-assigned goal location $g_i \in V$. All start locations are pairwise different and all goal locations are pairwise different.

Let $\pi_i(t)$ denote the location of agent a_i at time t. A timestep t means the move from time t to time $t+1$. A path $\pi_i = \langle \pi_i(0), \cdots, \pi_i(T_i), \pi_i(T_i+1), \cdots \rangle$ for agent a_i is a sequence of locations that satisfies the following conditions:

1. The agent starts at its start location, $\pi_i(0) = s_i$.
2. In each timestep t, the agent either moves to a neighboring location $\pi_i(t+1) \in V$, so $(\pi_i(t), \pi_i(t+1)) \in E$; or stays in its current location so $\pi_i(t) = \pi_i(t+1)$.
3. The agent ends at its goal location at the finish time T_i, which is the minimum time T_i such that for all times $t = T_i, \cdots, \infty$, $\pi_i(t) = g_i$.

Agents need to avoid collisions while moving to their goal location. A collision between agent a_i and agent a_j is either

- a vertex collision $\langle a_i, a_j, u, t \rangle$ where two agents a_i and a_j are in the same location $u = \pi_i(t) = \pi_j(t)$ at timestep t;
- an edge collision $\langle a_i, a_j, u, v, t \rangle$ where two agents traverse the same edge (u, v), where $u = \pi_i(t) = \pi_j(t+1)$ and $v = \pi_i(t+1) = \pi_j(t)$, in the opposite direction at timestep t.

A MAPF *plan* consists of a path π_i assigned to each agent a_i. A MAPF *solution* is a plan whose paths are collision-free. The MAPF problem is to find a solution which minimizes

- makespan, which is the maximum of the finish times of all agents at their goal locations $\max_{i \leq i \leq m} T_i$; or,
- flowtime, which is the sum of the finish times of all agents at their goal locations $\sum_{i=1}^{m} T_i$.

2.2.2 Complexity

The makespan and flowtime of MAPF instances are bounded by $O(|V|^3)$ based on the results of a complete $O(|V|^3)$ algorithm that finds a solution of $O(|V|^3)$ edge traversals or distinguishes an unsolvable instance [40]. The MAPF problem is NP-hard to solve optimally for flowtime minimization [39] and even to approximate within any constant factor less than 4/3 for makespan minimization [32, 20].

2.2.3 Conflict-Based Search (CBS)

Existing MAPF algorithms include reductions to other well-studied problems [38, 31, 5] and specialized rule-based, search-based hybrid algorithms [17, 35, 3, 27, 34, 26, 12]. In particular, Conflict-Based Search (CBS) [26] is a popular two-level optimal MAPF algorithm that computes time-optimal paths for individual agents on low level and performs a best-first tree search to resolve collisions among paths on the high level.

High-Level Search of CBS

CBS performs a best-first search on the high-level to build a constraint tree (CT) and resolve constraints among agents. Each node N in the constraint tree contains:

- a set of *constraints*, where a vertex constraint $\langle a_i, u, t\rangle$ prohibits agent a_i from being at location u at timestep t, and an edge constraint $\langle a_i, u, v, t\rangle$ prohibits agent a_i from moving along the edge $(u, v) \in E$ from u to v at timestep t;
- a set of *paths* that obey the *constraints*; and
- a *cost*, which is the flowtime of the *paths*.

Algorithm 1 shows the high-level search of CBS. CBS starts with a root node R with an empty set of constraints and an empty set of paths. For each agent, it performs the low-level space-time A* [28] to find a time-optimal path independently. If the low-level space-time A* could not find a feasible path for any agent and returns no path, the high-level search terminates unsuccessfully. Otherwise, successfully planned paths are added to *paths* in R so the root node contains paths for all agents. The cost of the root node is the flowtime of paths. The root node is added to the OPEN list. CBS chooses a node N with the smallest cost and removes it from the OPEN list (breaking ties in favor of the node with the smallest number of collisions, then the node with the earliest generated time). If the paths of node N have no collision, then N is a goal node; CBS terminates successfully and returns $N.paths$. Otherwise, CBS chooses a vertex collision $\langle a_i, a_j, u, t\rangle$ or an edge collision $\langle a_i, a_j, u, v, t\rangle$ it needs to resolve, and generates two child nodes. Each node inherits $N.paths$ and $N.constraints$. CBS adds a vertex constraint $\langle a_i, u, t\rangle$ to the first child node and the other vertex constraint $\langle a_j, u, t\rangle$ to the second child node when resolving a vertex collision $\langle a_i, a_j, u, t\rangle$. CBS adds an edge constraint $\langle a_i, u, v, t\rangle$ to the first child node and the other edge constraint $\langle a_j, v, u, t\rangle$ to the second child node when resolving an edge collision $\langle a_i, a_j, u, v, t\rangle$. For each child node, CBS performs a low-level space-time A* search to find a time-optimal path for agent a_i (or a_j) that obeys all constraints of the child node relevant to a_i (or a_j). If the low-level space-time A* search successfully find such a path, CBS replaces the old path of a_i (or a_j) with the new one, updates the cost of the child node accordingly and adds the child node into the OPEN list. If the OPEN list becomes empty, the algorithm declares a failure of the search.

Low-Level Search of CBS: Space-Time A*

The low-level space-time A* search finds a time-optimal path for agent a_i that obeys the constraints of node N. A space-time A* search is an A* search whose states are pairs of a location and a timestep $\langle v, t\rangle$. It starts from the state $\langle s_i, 0\rangle$, indicating agent a_i being at its start location s_i at timestep 0. A directed edge exists from state $\langle u, t\rangle$ to state $\langle v, t+1\rangle$

Algorithm 1: High Level of CBS

```
1  OPEN ← ∅
   /* initialize root node R                                          */
2  R.constraints ← ∅
3  R.paths ← ∅
4  for each agent a_i do
5      if LowLevel(a_i, R.constraints) returns no path then
6          return No Solution
7      R.paths[a_i] ← LowLevel(a_i, R.constraints)
8  R.cost ← flowtime(R.paths)
9  OPEN ← OPEN ∪ {R}
10 while OPEN ≠ ∅ do
11     N ← min_{N'∈OPEN} N'.cost
12     OPEN ← OPEN \{N}
13     if N.paths has no collision then
14         return N.paths
15     collision ← ⟨a_i, a_j, u, v/NULL, t⟩ in N.plan
       /* generate child nodes                                        */
16     for agent a_k in {a_i, a_j} do
17         Q ← new node
18         Q.constraints ← N.constraints ∪{⟨a_k, u, v/NULL, t⟩}
19         Q.paths ← N.paths;
20         if LowLevel(a_k, Q.constraints) returns a path then
21             Q.paths[a_k] ← LowLevel(a_k, Q.constraints)
22             Q.cost ← flowtime(Q.paths)
23             OPEN ← OPEN ∪{ Q }
24 return No Solution
```

if and only if $u = v$ or $(u, v) \in E$. To obey the given constrains, the set of states $\{\langle u, t\rangle\}$ is removed from the state space of agent a_i if and only if there is a vertex constraint $\langle a_i, u, t\rangle$; similarly, the set of edges $\{(\langle u, t\rangle, \langle v, t+1\rangle)\}$ is removed if and only if there is an edge constraint $\langle a_i, u, v, t\rangle$. CBS uses an upper bound $\mathcal{U}$ on the makespan [40] to guarantee the optimality and completeness. The makespan of paths is bounded by $\mathcal{U}$ if and only if the finish times of all agents are bounded by $\mathcal{U}$. So the low-level space-time A* search is also bounded by $\mathcal{U}$. So the space-time A* search terminates unsuccessfully and returns no paths if it tries to expand a state whose timestep is larger than $\mathcal{U}$.

Properties of CBS

Theorem 1. *CBS is guaranteed to find an optimal flowtime solution if the given MAPF instance is solvable, or identify an unsolvable MAPF instance [26].*

Heuristics for High-Level Search of CBS

Recent research [2, 13] introduce three admissible heuristics for the high-level search of CBS to significantly reduce the number of expanded nodes and speed up the search. Collisions found in $N.paths$ can be classified into three types based on the cost of resulting child nodes:

- cardinal collision if both of the resulting child nodes have larger cost than N;
- semi-cardinal collision if only one child node has a larger cost than N;
- non-cardinal collision if both of the resulting child nodes have the same cost as N.

The classification of collisions is based on Multi-Valued Decision Diagrams (MDDs). An MDD for agent a_i at node N is a directed acyclic graph consisting of all cost-optimal paths of a_i from s_i to g_i that satisfy $N.constraints$. A collision between two agents a_i and a_j at timestep t is a cardinal collision if and only if the contested vertex or edge is the only vertex or edge at level t of the MDDs of both agents.

Collision Graph (CG) Heuristic: CG heuristic only considers cardinal collision between agents. The admissible heuristic value is the size of the minimum vertex cover of a collision graph of N, whose vertices represent the agents and edges represent cardinal collisions between agents.

Dependency Graph (DG) Heuristic: DG heuristic considers all types of collisions between agents. Two agents are dependent if (1) their paths have cardinal collisions; or (2) their paths have other types of collisions and the joint MDD of their MDDs are empty. The joint MDD of two agents consists of all combinations of their cost-optimal collision-free paths. An empty joint MDD indicates that two agents are dependent. The admissible heuristic value is the size of minimum vertex cover of the pairwise dependency graph, whose vertices represent the agents and edges represent the dependencies between two agents.

Weighted Dependency Graph (WDG) Heuristic: Based on the pairwise dependency graph from DG, the weights of edges can be calculated. The weight of an edge is the difference between the minimum flowtime of the collision-free paths of corresponding agents satisfying $N.constraints$ and the sum of the cost of their paths in $N.paths$. The admissible heuristic value is the size of the edge-weighted minimum vertex cover of the edge-weighted pairwise dependency graph.

Enhanced Conflict-Based Search

Enhanced CBS is a bounded-suboptimal MAPF algorithm. It uses Focal search on both high-level and low-level of CBS. Focal search maintains two lists of nodes: the OPEN list and the FOCAL list. The OPEN list is the regular OPEN list of A*. FOCAL list contains a subset of nodes from the OPEN list. Focal search uses two functions f_1 and f_2. f_1 is an admissible function that defines which nodes are in the FOCAL list. Let $f_{1_{min}}$ denote the

minimal f_1 value in the OPEN list. Given a suboptimality factor ω, FOCAL contains all nodes N in the OPEN list whose costs are less than or equal to $\omega \cdot f_{1_{min}}$. f_2 is used to prioritize nodes in the FOCAL list. It is guaranteed that the returned solution is at most $\omega \cdot f_{1_{min}}$.

Low level focal search: apply focal search to find a single agent path where $f_1(n)$ is the regular $f_1(n) = g(n) + h(n)$ of A* search and $f_2(n)$ is the number of conflicts between current planning agent and other agents. The low-level search returns $f_{1_{min}}(a_i)$ which is a lower bound on the optimal path, as well as the planned paths.

High level focal search: with the lower bound returned by low-level search, for a node N, let $LB(N) = \sum_{i=1}^{m} f_{1_{min}}(a_i)$, which is the sum of lower bounds of low-level searches. The cost of N is guaranteed to be less than or equal to the suboptimality factor times the the sum of lower bounds of low-level searches. The lower bound value $LB(N)$ is used to prioritize nodes in the OPEN list. Let $LB = \min(LB(N)), N \in OPEN$ denote a lower bound in the OPEN list and on the optimal solution of the entire MAPF problem, and the FOCAL list contain all nodes N where $N.cost \leq \omega \cdot LB$. Once a solution is found, it is guaranteed to have cost at most $\omega \cdot LB$.

2.3 Target-Assignment and Pathfinding

2.3.1 Problem Definition

An instance of the target-assignment and pathfinding (TAPF) problem consists of:

1. an undirected graph $G = (V, E)$ where V corresponds to the set of locations and E corresponds to the unit-weight edges connecting locations that agents can move along;

2. a set of m agents that are partitioned into K teams $team_1, \cdots, team_k$. Each team $team_k$ consists of m_k agents $a_1^k, \cdots, a_{m_k}^k$. Each team has m_k targets (single-goal tasks) $g_1^k, \cdots, g_{m_k}^k$ such that $m = \sum_{k=1}^{K} m_k$. Each agent a_i^k has a unique start locations s_i^k; all start locations are pairwise different and all targets are pairwise different.

An assignment of targets to agents in a team $team_k$ is a one-to-one mapping σ^k that maps each agent a_i^k to a target $g_j^k = \sigma^k(a_i^k)$ of the same team. An agent in a team can be assigned to any one of the targets of the same team. An agent cannot be assigned to a target of a different team.

Let $\pi_i^k(t)$ denote the location of agent a_i^k at time t. A path $\pi_i^k = \langle \pi_i^k(0), \cdots, \pi_i^k(T_i^k), \pi_i^k(T_i^k + 1), \cdots \rangle$ for agent a_i^k is a sequence of locations that satisfies the following conditions:

1. The agent starts at its start location, $\pi_i^k(0) = s_i^k$.

2. In each timestep t, the agent either moves to a neighboring location $\pi_i^k(t+1) \in V$, where $(\pi_i^k(t), \pi_i^k(t+1)) \in E$, or stays in its current location $\pi_i^k(t) = \pi_i^k(t+1)$.

3. The agent ends at its assigned target at the finish time T_i^k, which is the minimum time T_i^k such that for all timesteps $t = T_i^k, \cdots, \infty$, $\pi_i^k(t) = g_j^k = \sigma^k(a_i^k)$.

Agents need to avoid collisions while moving to its assigned target. A collision between teams k and $k' \neq k$ can be a vertex collision or an edge collision, where:

- a vertex collision between an agent a_i^k in team $team_k$ and a different agent $a_j^{k'}$ in team $team_{k'}$ is a tuple $\langle team_k, team_{k'}, u, t\rangle$, where agents a_i^k and $a_j^{k'}$ occupy the same location $u = \pi_i^k(t) = \pi_j^{k'}(t)$ at time t;
- an edge collision between an agent a_i^k in team $team_k$ and a different agent $a_j^{k'}$ in team $team_{k'}$ is a tuple $\langle team_k, team_{k'}, u, v, t\rangle$, where agents a_i^k and $a_j^{k'}$ traverse the same edge (u, v), where $u = \pi_i^k(t) = \pi_j^{k'}(t+1)$ and $v = \pi_i^k(t+1) = \pi_j^{k'}(t)$, in the opposite direction between times t and $t+1$.

If $k' = k$, the collision occurs between agents within the same team. A collision between agents within the same team can be

- a vertex collision $\langle a_i^k, a_j^k, u, t\rangle$ where two agents a_i^k and a_j^k are in the same location $u = \pi_i^k(t) = \pi_j^k(t)$ at time t;
- an edge collision $\langle a_i^k, a_j^k, u, v, t\rangle$ where two agents traverse the same edge (u, v), where $u = \pi_i^k(t) = \pi_j^k(t+1)$ and $v = \pi_i^k(t+1) = \pi_j^k(t)$, in the opposite direction between times t and $t+1$.

A TAPF *plan* consists of an assignment σ^k of targets to agents for each team $team_k$ and a path π_i^k for each agent a_i^k in each team $team_k$. A TAPF *solution* is a TAPF plan whose paths are collision-free. The TAPF problem is to find a solution which minimizes

- makespan, which is the maximum of the finish times of all agents at their assigned targets $\max_{i \in m_k, k \in K} T_i^k$; or
- flowtime, which is the sum of the finish times of all agents at their assigned targets $\sum_{i \in m_k, k \in K} T_i^k$.

The special case of TAPF when the number of agents $k = 1$ is also known as anonymous MAPF. Targets can be assigned to any agents.

2.3.2 Complexity

TAPF can be solved optimally in polynomial time for makespan minimization by Conflict-based min-cost flow algorithm [37, 18]. We consider the another special case of TAPF where the number of teams is the same as the number of agents, i.e. $m = K$, which is known as the MAPF problem. Since the makespan of any optimal MAPF solution is bounded by $O(|V|^3)$, the makespan of any optimal TAPF solution is also bounded by $O(|V|^3)$ [18]. For flowtime minimization, its complexity remains unclear.

Algorithm 2: firstAssignment

Input : cost matrix C
Output: best task assignment, ASG_OPEN

```
1 R ← new node
2 R.O ← ∅
3 R.I ← ∅;
4 R.solution ← constrainedAssignment(R, C)
5 ASG_OPEN ← { R }
6 return R.solution
```

2.3.3 Conflict-Based Search with Task Assignment (CBS-TA)

Conflict-Based Search with Task Assignment (CBS-TA) [8] solves the TAPF problem with $k = 1$ (anonymous MAPF problem) optimally for flowtime minimization. Based on CBS as described above, CBS-TA only changes the high-level search. Each high-level node has two additional fields:

- a Boolean value *root*, indicating whether a node is a root node;
- an *assignment*, which is the task assignment of the node.

Instead of building a search tree as CBS does, CBS-TA builds a search forest where each tree in the forest corresponds to a different task assignment. CBS-TA starts with a single root node with the best task assignment. A new root node is created only if the current expanding node is a root node. An $m \times m$ cost matrix C contains the lower bound for agent a_i to reach each target g_j. Each element in the cost matrix is the length of the shortest path for each pair of agent and task, ignoring all other agents. The Hungarian method compute the optimal task assignment for the cost matrix C. An assignment OPEN list (ASG_OPEN) is used to maintain all assignments which has been generated but not expanded. Algorithms 2, 3 and 4 show pseudo code for task assignments. The basic idea is to partition the solution space such that forbidding some assignments (denoted as O) and forcefully include others (donated as I). It is shown that such partitioning can cover the complete solution space [23].

Property of CBS-TA

Theorem 2. *CBS-TA computes a solution that minimizes the flowtime of all agents if a solution exists [8].*

Algorithm 3: nextAssignment

Input : cost matrix C, ASG_OPEN
Output: next best task assignment

```
1  if ASG_OPEN = ∅ then
2      return No next assignment
3  P ← best node from ASG_OPEN
4  for i ← 1 to m do
5      if i is not in P.I then
6          Q ← new node
7          Q.O ← P.O ∪{P.solution[i]}
8          Q.I ← P.I ∪{P.solution[j] : j < i}
9          Q.solution ← constrainedAssignment(Q, C)
10         if Q.solution is not None then
11             ASG_OPEN ← ASG_OPEN ∪{ Q }
12 return solution of best node from ASG_OPEN
```

Algorithm 4: constrainedAssignment

Input : cost matrix C, P
Output: task assignment

```
1  C' ← C
2  for each pair (i, j) in P.O do
3      C'_(i,j) ← ∞
4  for each pair (i, j) in P.I do
5      C'_(i,j) ← 0
6  assignment ← Hungarian(C')
7  if assignment is not None then
8      return assignment
```

2.4 Lifelong Task-Assignment and Pathfinding

Recent research has considered the online multi-agent pickup and delivery (MAPD) problem. In the MAPD problem, agents have to execute a stream of delivery tasks, where tasks appear at unknown times, and each task has a sequence of two goal locations [19].

2.4.1 Problem Definition of Multi-Agent Pickup and Delivery

An instance of a multi-agent pickup and delivery (MAPD) problem consists of:

1. an undirected graph $G = (V, E)$ where V corresponds to the set of locations and E corresponds to the unit-weight edges connecting locations that agents can move along;
2. a set of m agents $\{a_1, \cdots, a_m\}$, and for each agent a_i, there is a start location $s_i \in V$. All start locations are pairwise different;

3. a set of tasks $\mathcal{T}$ that contains all unexecuted tasks. Each task τ_j contains a pickup location $s_j \in V$ and a delivery location $g_j \in V$. An unexecuted task τ_j is added to the task set $\mathcal{T}$ at unknown timestep. A task is available for execution only when it has been added to the task set $\mathcal{T}$.

Let $\pi_i(t)$ denote the location of agent a_i at time t. A path $\pi_i = \langle \pi_i(0), \cdots, \pi_i(T_i), \pi_i(T_i+1), \cdots \rangle$ for agent a_i is a sequence of locations that satisfies the following conditions:

1. The agent starts at its start location, $\pi_i(0) = s_i$.
2. In each timestep t, the agent either moves to a neighboring location $\pi_i(t+1)$ inV, where $(\pi_i(t), \pi_i(t+1)) \in E$, or stays in its current location $\pi_i(t) = \pi_i(t+1)$.

Agents need to avoid collisions while moving to its goal location. A collision between agent a_i and agent a_j is either

- a vertex collision $\langle a_i, a_j, u, t \rangle$ where two agents a_i and a_j are in the same location $u = \pi_i(t) = \pi_j(t)$ at timestep t;
- an edge collision $\langle a_i, a_j, u, v, t \rangle$ where two agents traverse the same edge (u, v), where $u = \pi_i(t) = \pi_j(t+1)$ and $v = \pi_i(t+1) = \pi_j(t)$, in the opposite direction between timesteps t and $t+1$.

An agent is *free* if and only if it is not currently executing any task, otherwise it is *occupied*. A free agent can be assigned any task $\tau_j \in \mathcal{T}$. In order to execute a task τ_j, the agent needs to move from its current location to the pickup location s_j of the task and then move from the pickup location s_j to the delivery location g_j. The agent starts to execute a task τ_j when it reaches the pickup location s_j, and finishes executing the task when it reaches the delivery location g_j. An agent can be assigned a different task when it is moving to the pickup location of the currently assigned task. But, an agent has to finish executing the task after it has reached the pickup location of the task before it can be assigned another task.

The MAPD problem is to find collision-free paths to finish executing all tasks, while minimizing:

- service time, which is the average number of timesteps needed to finish executing each task after it was added to the task set; or
- makespan, which is the earliest time when all tasks are finished.

2.4.2 Mult-Label Space-Time A*

Grenouilleau et al.[7] introduced the Multi-Label Space-Time A* (MLA*) algorithm, which computes a time-optimal path for an agent that visits two goal locations and solves the task assignment and the pathfinding independently.

MLA* uses the space-time A* algorithm structure described above. The search space of MLA* is a tuple of a location, a time and a label $\{\langle v, t, l\rangle | l \in \{1, 2\}\}$ where $l = 1$ indicates the agent is seeking a path to the pickup location and $l = 2$ indicates the agent is seeking a path to the delivery location. Let π_1 and π_2 denote the pickup location and the delivery location, respectively. The heuristic value of a state is defined

$$h(\langle v, t, l\rangle) = \begin{cases} \text{dist}(v, \pi_1) + \text{dist}(\pi_1, \pi_2) & \text{if } l = 1 \\ \text{dist}(v, \pi_2) & \text{if } l = 2 \end{cases}$$

where $\text{dist}(\cdot, \cdot)$ is the distance between two points in the graph G.

Chapter 3

Problem Definition

An MG-TAPF instance consists of:

1. an undirected graph $G = (V, E)$, where V is the set of locations, and E is the set of unit-weight edges connecting locations;
2. m agents $\{a_1, a_2, \ldots, a_m\}$ and for each agent a_i, there is a start location $s_i \in V$;
3. m tasks $\{\boldsymbol{g}_1, \boldsymbol{g}_2, \ldots, \boldsymbol{g}_m\}$, where each task $\boldsymbol{g}_j$ is characterized by a sequence of K_j goal locations $\boldsymbol{g}_j = \langle \boldsymbol{g}_j[1], \ldots, \boldsymbol{g}_j[K_j] \rangle$.

An assignment of tasks to agents is a one-to-one mapping σ that maps each agent a_i to a task $\boldsymbol{g}_j$. Each agent a_i can be assigned any task $\boldsymbol{g}_j$.

Let $\pi_i(t)$ denote the location of agent a_i at time t. A path $\pi_i = \langle \pi_i(0), \ldots, \pi_i(T_i), \pi_i(T_i + 1), \ldots \rangle$ for agent a_i is a sequence of locations that satisfies the following conditions:

1. The agent starts at its start location, $\pi_i(0) = s_i$;
2. In each timestep t, the agent either moves to a neighboring location $\pi_i(t+1) \in V$ where $(\pi_i(t), \pi_i(t+1)) \in E$, or stays in its current location, $\pi_i(t) = \pi_i(t+1)$;
3. The agent visits all goal locations of its assigned tasks $\boldsymbol{g}_j$ in sequence and remains in the final goal location at the *finish time* T_i, which is the minimum time T_i such that $\pi_i(t) = \sigma(a_i) = \boldsymbol{g}_j[K_j]$, for all times $t = T_i, \ldots, \infty$.

Agents need to avoid collisions while moving to their goal locations. A collision between agents a_i and a_j is either

1. a vertex collision $\langle a_i, a_j, u, t \rangle$ where two agents a_i and a_j are in the same location $u = \pi_i(t) = \pi_j(t)$ at time t; or
2. an edge collision $\langle a_i, a_j, u, v, t \rangle$ where two agents a_i and a_j traverse the same edge (u, v) in the opposite directions $u = \pi_i(t) = \pi_j(t+1)$ and $v = \pi_i(t+1) = \pi_j(t+1)$ in the same timestep t.

A *plan* consists of an assignment σ of tasks to agents and a path for each agent. A *solution* is a plan whose paths are collision-free. The problem of MG-TAPF is to find a solution which minimizes:

- flowtime $\sum_{i=1}^{m} T_i$, which is the sum of the finish times of paths of all agents; or
- makespan $\max_{1 \leq i \leq m} T_i$, which is the maximum of the finish times of all agents.

In this thesis, we only consider the flowtime objective, even though many of our results could be easily generalized to other objectives such as makespan minimization, partially because the flowtime objective matches the throughput objective in the lifelong problems (for example, MAPD) well.

Chapter 4

Complexity

We show that the MG-TAPF problem is NP-hard to solve optimally for flowtime minimization, even when each task has only two goal locations. Similar to [20, 21], we use a reduction from ≤3,=3-SAT [33], an NP-complete version of the Boolean satisfiability problem. A ≤3,=3-SAT instance consists of N Boolean variables $\{X_1, \ldots, X_N\}$ and M disjunctive clauses $\{C_1, \ldots, C_M\}$, where each variable appears in exactly three clauses, uncomplemented at least once and complemented at least once. Each clause contains at most three literals. Its decision question asks whether there exists a satisfying assignment. We first show a constant-factor inapproximability result for makespan minimization.

Theorem 3. *For any $\epsilon > 0$, it is NP-hard to find a $(4/3 - \epsilon)$-approximate solution to the MG-TAPF problem for makespan minimization, even if each task has only two goal locations.*

Proof. We use a reduction similar to that is used in the proof of Theorem 3 in [20] to construct an MG-TAPF instance with $m = M + 2N$ agents and m tasks that has a solution with makespan three, if and only if a given ≤3,=3-SAT instance with N Boolean variables and M disjunctive clauses is satisfiable. Figure 4.1 shows an example.

For each variable X_i in the ≤3,=3-SAT instance, we construct two "literal" agents a_{iT} and a_{iF}, with start locations s_{iT} and s_{iF} respectively, and two tasks $\boldsymbol{g}_{iT}$ and $\boldsymbol{g}_{iF}$, each with two goal locations, namely $\boldsymbol{g}_{iT}[1] = s_{iT}$, $\boldsymbol{g}_{iT}[2] = t_{iT}$, $\boldsymbol{g}_{iF}[1] = s_{iF}$, $\boldsymbol{g}_{iF}[2] = t_{iF}$. Therefore, any optimal solution must assign every task to the agent whose start location is the first goal location of the task and let the agent execute the task. For each literal agent, we construct two paths to move it from its start location to the final goal location in three timesteps: "shared" paths, namely $\langle s_{iT}, u_{iT}, v_i, t_{iT} \rangle$ for a_{iT} and $\langle s_{iF}, u_{iF}, v_i, t_{iF} \rangle$ for a_{iF}; and "private" path, namely $\langle s_{iT}, w_{iT}, x_{iT}, t_{iT} \rangle$ for a_{iT} and $\langle s_{iF}, w_{iF}, x_{iF}, t_{iF} \rangle$ for a_{iF}. The shared paths p_{iT} and p_{iF} intersect at vertex v_i. Only one of the two paths can thus be used if a makespan of three is to be achieved. Moving literal agents a_{iT} (or a_{iF}) along the shared path corresponds to assigning *True* (or *False*) to X_i in the ≤3,=3-SAT instance.

For each clause C_j in the $\leq$3,=3-SAT instance, we construct a "clause" agent a_j with the start location c_j and a task $\boldsymbol{g}_j$ with two goal locations $\boldsymbol{g}_j[1] = c_j$ and $\boldsymbol{g}_j[2] = d_j$. It has multiple but at most three "clause" paths to move the agent from its start location to its goal locations in three timesteps, which have a one-to-one correspondence to the literals in C_j. Every literal X_i (or $\overline{X_i}$) can appear in at most two clauses. If C_j is the first clause that it appears in, the clause path is $\langle c_j, w_{iT}, b_j, d_j \rangle$ (or $\langle c_j, w_{iF}, b_j, d_j \rangle$). If C_j is the second clause that it appears in, a vertex e_j is introduced and the clause path is instead $\langle c_j, e_j, x_{iT}, d_j \rangle$ (or $\langle c_j, e_j, x_{iF}, d_j \rangle$). The clause path of each C_j with respect to any literal in that clause and the private path of literal intersect. Only one of the two paths can thus be used if a makespan of three is to be achieved.

If a satisfying assignment to the $\leq$3,=3-SAT instance exists, the a solution with makespan three to the MG-TAPF instance is obtained by moving literal agents along their private paths and clause agents along the clause paths corresponding to one of the true literals in those clauses.

Conversely, if a solution with makespan three to the MG-TAPF instance exists, then each clause agent traverses the clause path corresponding to one of the literals in that clause, and the corresponding literal agent traverses its shared path. Since the agents of a literal and its complement cannot both use their shared path if a makespan of three is to be achieved, we can assign *True* to every literal whose agent uses its shared path without assigning *True* to every literal to both the uncomplemented and complemented literals. On the other hand, if the agents of both literals use their private paths, we can assign *True* to any one of the literals and *False* to the other one. A solution to the MG-TAPF instance with makespan three thus yields a satisfying assignment to the $\leq$3,=3-SAT instance.

Thus, we can show that the constructed MG-TAPF instance has a solution with makespan three if and only if 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. For any $\epsilon > 0$, any MG-TAPF algorithm with approximation ratio $4/3 - \epsilon$ thus computes a solution with makespan three whenever the $\leq$3,=3-SAT instance is satisfiable and thus solves $\leq$3,=3-SAT. □

In the proof of Theorem 3, the MG-TAPF instance reduced from the given $\leq$3,=3-SAT instance has the property that the length of every path from the start location of every agent to the last (second) goal location of the task assigned to the agent is at least three. Therefore, if the makespan is three, every agent arrives at the last location of the task assigned in exactly three timesteps, and the flowtime is $3m$. Moreover, if the makespan exceeds three, the flowtime exceeds $3m$, yielding the following theorem.

Theorem 4. *It is NP-hard to find the optimal solution to the MG-TAPF problem for flowtime minimization, even if each task has only two goal locations.*

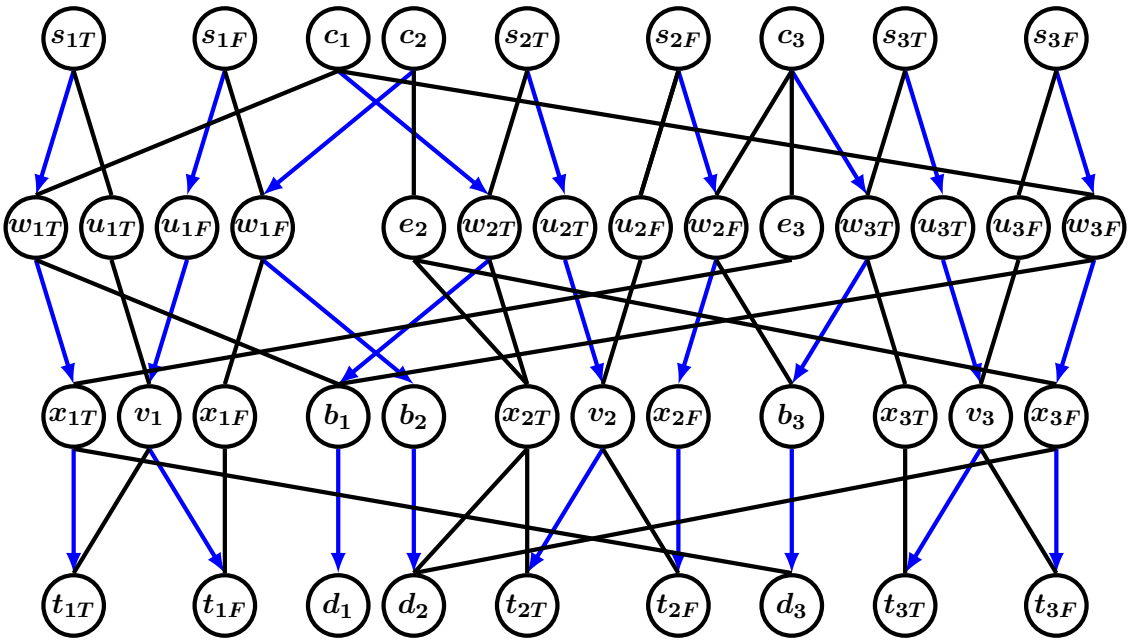

Figure 4.1: Example of the reduction from a $\leq$3,=3-SAT problem instance $(X_1 \vee X_2 \vee \overline{X_3}) \wedge (\overline{X_1} \vee X_2 \vee \overline{X_3}) \wedge (X_1 \vee \overline{X_2} \vee X_3)$. Clause C_1 is the first clause that literal X_1 appears in. The corresponding clause path is $\langle c_1, w_{1T}, b_1, d_1 \rangle$. Since clause C_3 is the second clause that X_1 appears in, vertex e_3 is introduced. The corresponding clause path is $\langle c_3, e_3, x_{1T}, d_3 \rangle$. The blue (directed) edges represent one optimal solution to the MG-TAPF instance of the makespan three, which corresponds to the satisfying assignment $(X_1, X_2, X_3) = (False, True, False)$.

Chapter 5

Conflict-Based Search with Task Assignment with Multi-Label A*

The Conflict-Based Search with Task Assignment with Multi-Label A* (CBS-TA-MLA) algorithm is a two-level search algorithm, where the low-level MLA* plans an optimal path for each agent based on the task assignment and the constraints provided by the high-level algorithm CBS-TA.

5.1 High-level: Conflict-Based Search with Task Assignment

Conflict-Based Search with Task Assignment (CBS-TA) is a best-first search algorithm, which was initially designed to solve the TAPF problem with $k = 1$ [8]. We extend it to solve MG-TAPF by replacing the low-level search algorithm with Multi-Label Space-Time A* [7]. Algorithm 5 shows the pseudo-code. CBS searches a binary tree called the constraint tree (CT), while CBS-TA searches a forest that contains multiple constraint trees. Each tree in the forest corresponds to a different task assignment. Each node in the tree (CT node) contains:

1. a Boolean value *root*, indicating whether the node is a root of a constraint tree;
2. an *assignment*, which is the task assignment of the node (nodes in the same tree have the same task assignment);
3. a set of *constraints*, where a vertex constraint $\langle a_i, u, t\rangle$ prohibits agent a_i from being at location u at time t, and an edge constraint $\langle a_i, u, v, t\rangle$ prohibits agent a_i from moving along the edge $(u, v) \in E$ from u to v at timestep t;
4. a set of *paths* that obey the *assignment* and the *constraints*; and
5. a *cost*, which is the flowtime of the *paths*.

An $m \times m$ cost matrix C contains the distances from the start locations of each agent to the final goal locations of each task (all intermediate goal locations of the task are visited

in order) while ignoring the other agents. CBS-TA starts with a single root node with the best task assignment (the task assignment with the lowest flowtime that ignores collisions between agents). The best task assignment is calculated by applying the Hungarian method [11] to the cost matrix C (Algorithms 2 and 4 show the pesudo code). A new root node with the next best task assignment will be created if the current expending node is a root node. The idea of computing the next best task assignment is as follows [8] (Algorithms 3 and 4 show the pseudocode). The basic idea is to partition the solution space such that forbidding some assignments (denoted as O) and forcefully include others (donated as I). It is shown that such partitioning can cover the complete solution space [23]. Then we compute a new cost matrix C' by copying C except that the costs of entries in I are changed to 0 and the costs of entries in O are changed to ∞. The Hungarian method is applied to the new cost matrix C' to calculate the new task assignment. So, the entries in I are always included and the entries in O are always excluded in the new assignment.

Once the task assignment is computed, the corresponding paths of the agents are planned by the low-level search algorithm MLA*. CBS-TA then finds collisions among planned paths, stores the number of collisions in the node and adds the node to the OPEN list.

CBS-TA chooses a node N with the lowest cost $N.cost$ from the OPEN list to expand (breaking ties in favor of the node with the smallest number of collisions, then the node with the earliest generated time). First, it checks whether the number of collisions is 0. If so, N is declared as the goal node, and $N.assignment$ and $N.paths$ will be returned as the solution to the instance. Otherwise, CBS-TA chooses the resolve the earliest vertex collision $Col = \langle a_i, a_j, u, t\rangle$ (that is, the paths of agents a_i and a_j have a collision at vertex u at time t) or the earliest edge collision $Col = \langle a_i, a_j, u, v, t\rangle$ (that is, the paths of agents a_i and a_j have a collision at edge (u, v) at timestep t), and generates two child nodes. Child nodes inherit the constraint set and paths from their parent node N. It CBS-TA resolves a vertex collision $Col = \langle a_i, a_j, u, t\rangle$, CBS-TA adds an vertex constraint $\langle a_i, u, t\rangle$ to the constraint set of one child node and the other vertex constraint $\langle a_j, u, t\rangle$ to the constraint set of the other child node. If CBS-TA resolves an edge collision $Col = \langle a_i, a_j, u, v, t\rangle$, CBS-TA adds an edge constraint $\langle a_i, u, v, t\rangle$ to the constraint set of one child node and the other edge constraint $\langle a_j, v, u, t\rangle$ to the constraint set of the other child node. It then calls the low-level search algorithm MLA* to replan the path of a_i (or a_j) for each child node while satisfying its constraint sets. If such a path does not exist, the child node is pruned. Otherwise, CBS-TA detects the collisions between the newly planned path and former paths of the other agents, updates the number of collisions and adds the child node to the OPEN list. If the OPEN list becomes empty, the algorithm declares a failure of search.

Algorithm 5: High Level of CBS-TA-MLA

```
1  OPEN ← ∅
   /* initialize first root node R                                          */
2  R ← new node
3  R.root ← True
4  R.assignment ← firstAssignment()
5  R.constraints ← ∅
6  for each agent a_i do
7      if MLA*(a_i, R.assignment[a_i], R.constraints) returns no path then
           /* no solution for this task assignment, try next assignment     */
8          continue to line 4 with R.assignment ← nextAssignment()
9      R.paths[a_i] ← MLA*(a_i, R.assignment[a_i], R.constraints)
10 R.cost ← flowtime(R.paths)
11 R.collisions ← findCollisions(R)
12 OPEN ← OPEN ∪ {R}
13 while OPEN ≠ ∅ do
14     N ← lowest cost node from OPEN
15     if N.collisions = ∅ then
16         return N.assignment, N.paths
17     if N.root is True then
           /* initialize new root node R with next best assignment          */
18         R ← new node
19         R.root ← True
20         R.assignment ← nextAssignment()
21         R.constraints ← ∅
22         for each agent a_i do
23             if MLA*(a_i, R.assignment[a_i], R.constraints) returns no path then
                   /* no solution for this task assignment                  */
24                 continue to the next iteration in line 13
25             R.paths[a_i] ← MLA*(a_i, R.assignment[a_i], R.constraints)
26         R.cost ← flowtime(R.paths)
27         R.collisions ← findCollisions(R)
28         OPEN ← OPEN ∪ {R}
29     OPEN ← OPEN \{N}
30     Collision ← ⟨a_i, a_j, u, v/NULL, t⟩ by chooseCollision(N)
31     for agent a_k in {a_i, a_j} do
32         Q ← new node
33         Q.root ← False
34         Q.assignment ← N.assignment
35         Q.constraints ← N.constraints ∪{⟨a_k, u, v/NULL, t⟩}
36         Q.paths ← N.paths
37         if MLA*(a_k, Q.assignment[a_k], Q.constraints) returns a path then
38             Q.paths[a_k] ← MLA*(a_k, Q.assignment[a_k], Q.constraints)
39             Q.cost ← flowtime(Q.paths)
40             Q.collisions ← findCollisions(Q)
41             OPEN ← OPEN ∪{ Q }
42 return No Solution
```

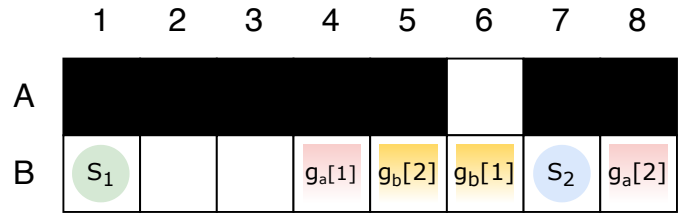

Figure 5.1: Example for comparing MLA* with calling A* many times

5.2 Low-level: Multi-Label Space-Time A*

Multi-Label Space-Time A* (MLA*) finds a time-optimal path for an agent a_i (a path with the smallest finish time T_i) that visits all goal locations $\boldsymbol{g}_j[K_j]$ of assigned task $\boldsymbol{g}_j$ in sequence and obeys a set of constraints. MLA* was first introduced for two goal locations [7], and then was extended to more than two goal locations [15]. MLA* extends Space-Time A* [28] by adding a label dimension indicating the different segments between goal locations, where label k represents that the next goal location to visit is $\boldsymbol{g}_j[k]$.

We formally describe MLA* in the context of CBS-TA-MLA. MLA* is an A* search algorithm whose states are tuples of a location, a timestep and a label. It starts from state $\langle s_i, 0, 1\rangle$, indicating agent a_i being at its start location s_i at timestep 0 with label 1. A directed edge exists from state $\langle u, t, k\rangle$ to state $\langle v, t+1, k'\rangle$ if and only if (1) $u = v$ or $(u, v) \in E$, and (2) $k' = k + 1$ if $v = \boldsymbol{g}_j[k]$ and $k' = k$ otherwise. To obey the given constraints, the set of states $\{\langle u, t, k\rangle | k = 0, \ldots, K_j\}$ is removed from the state space of agent a_i if and only if there is a vertex constraint $\langle a_i, u, t\rangle$; and similarly, the set of edges $\{(\langle u, t, k\rangle, \langle v, t+1, k'\rangle) | k = 0, \ldots, K_j - 1\}$ is removed if and only if there is an edge constraint $\langle a_i, u, v, t\rangle$. If MLA* expands a goal state $\langle \boldsymbol{g}_j[K_j], t, K_j + 1\rangle$ and the agent can stay at the goal location forever (without violating any vertex constraints), it terminates and returns the path (a sequence of locations) from the start state to the goal state.

During the search, the h-value of each state $\langle v, t, k\rangle$ is calculated by:

$$h(\langle v, t, k\rangle) = \text{dist}(v, \boldsymbol{g}_j[k]) + \sum_{k'=k}^{K_j - 1} \text{dist}(\boldsymbol{g}_j[k'], \boldsymbol{g}_j[k'+1])$$

that is, the shortest distance from location v to visit all unvisited goal locations in task $\boldsymbol{g}_j$ in sequence. The distances $\text{dist}(v, \boldsymbol{g}_j[k])$ from each location $v \in V$ to all goal locations $\boldsymbol{g}_j[k]$ with $j = 1, \ldots, m$ and $k = 1, \ldots, K_j$ are pre-computed by searching backward once from each goal location $\boldsymbol{g}_j[k]$ on the graph G.

Comparing MLA* with calling A* many times

Consider an example in Figure 5.1 with two agents a_1 and a_2 located at their start locations $s_1 = B1$ and $s_2 = B7$. Two tasks $\boldsymbol{g}_a$ and $\boldsymbol{g}_b$ will be assigned and executed by two agents,

where $\boldsymbol{g}_a[1] = B4$, $\boldsymbol{g}_a[2] = B8$, $\boldsymbol{g}_b[1] = B6$ and $\boldsymbol{g}_b[2] = B5$. If we choose to use A* and only consider the first goal locations of tasks at the beginning, the task assignment is $\boldsymbol{g}_a$ to a_1 and $\boldsymbol{g}_b$ to a_2. At time 2, a_2 reaches the final goal location $\boldsymbol{g}_b[2]$ and stays there for the remaining times. At time 3, a_1 reaches its first goal location $\boldsymbol{g}_a[1]$ and starts planning a path to its next goal location $\boldsymbol{g}_a[2]$. It finds that a_2 blocks its way to $\boldsymbol{g}_a[2]$, resulting in more collisions. However, if we consider every goals in the tasks at the beginning, a_2 will first move to the alcove way to let a_1 pass by. With MLA*, we could reduce unnecessary collisions with other agents, which also benefits the high-level search.

5.3 Properties

We now show that CBS-TA-MLA is complete and optimal. Yu and Rus [40] suggested that there exists and upper bound for any solvable MAPF instance of $O(|V|^3)$ single agent steps required for all agents move from their start locations to their goal locations. There even exists a linear algorithm to detect an unsolvable MAPF instance [40]. However, it is difficult to detect unsolvable MG-TAPF instances using existing algorithms. We still provide an upper bound for any solvable MG-TAPF instances.

Theorem 5. *CBS-TA-MLA is guaranteed to find an optimal solution if the given MG-TAPF instance is solvable and correctly identifies an unsolvable MG-TAPF instance with an upper bound of $\mathcal{O}(|V|^3 \cdot \sum_{j=0}^{m} K_j)$ on the finish time T_i of any agent at the final goal location of its assigned tasks.*

Proof. The proof of the optimally of CBS-TA-MLA is trivial as CBS-TA and MLA* have been proved to be optimal in[8] and [7], respectively. As for the completeness, consider an arbitrary optimal solution to the given MG-TAPF instance with paths $\pi_i(t)$. The solution can be divided chronologically into at most $\mathcal{K} = \sum_{j=0}^{m} K_j$ segments at breakpoints $t^{(0)} = 0, t^{(1)}, \ldots, t^{(\mathcal{K})} = \max_i T_i$ where the label of at least one agent changes (at least one agent reaches a new goal location of its assigned task) at each $t^{(\kappa)}, \forall \kappa \geq 1$. Since there exists a solution with at most $U = \mathcal{O}(|V|^3)$ agent movements (edge traversals) to any solvable MAPF instance [40], there exist collision-free paths for all agents with makespan at most U that move each a_i from $\pi(t^{(\kappa-1)})$ to $\pi(t^{(\kappa)}), \forall \kappa \geq 1$, and thus $t^{(\kappa)} - t^{(\kappa-1)} \leq U, \forall t \geq 1$.

Therefore, $t^{(\mathcal{K})} \leq U \cdot \sum_{j=0}^{m} K_j$. CBS-TA-MLA can thus safely prune any state whose timestep is larger than $U \cdot \sum_{j=0}^{m} K_j$ on the low level search and terminate for any given unsolvable MG-TAPF instance when the OPEN list eventually becomes empty in finite time. □

5.4 Example

Consider the example in Figure 5.2 with two agents a_1 and a_2 located at their start locations $s_1 = B1$ and $s_2 = A2$ respectively. Two tasks $\boldsymbol{g}_a$ and $\boldsymbol{g}_b$ will be assigned to two agents, where

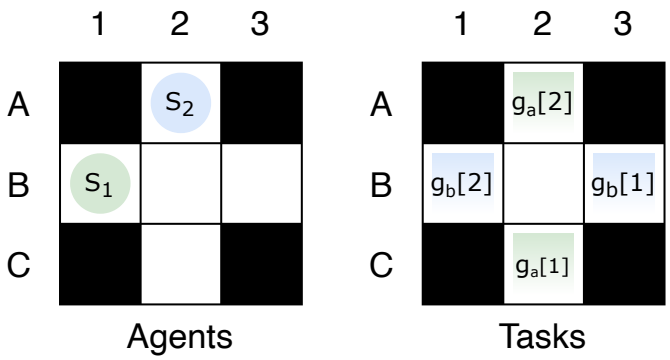

Figure 5.2: An example instance with agents and tasks

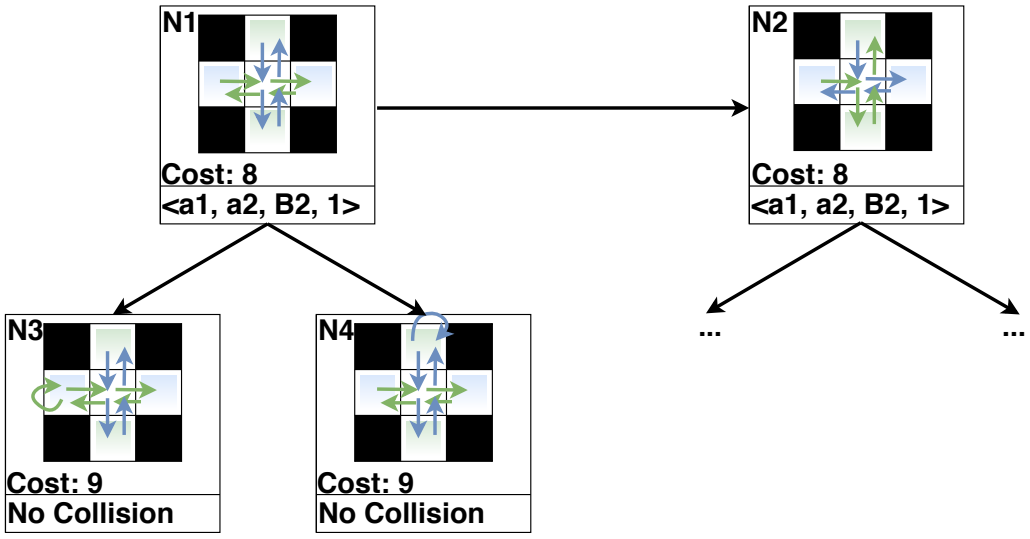

Figure 5.3: A search forest of the high-level search of CBS-TA-MLA of the example shown in Figure 5.2. The earliest collision found among paths is shown at the bottom of each node.

$\boldsymbol{g}_a[1] = C2$, $\boldsymbol{g}_a[2] = A2$, $\boldsymbol{g}_b[1] = B3$ and $\boldsymbol{g}_b[2] = B1$. The cost matrix C which contains the distance between all start locations and all final goal locations is: $\left(\begin{smallmatrix} 4 & 4 \\ 4 & 4 \end{smallmatrix}\right)$.

The corresponding high-level search forest of CBS-TA-MLA is shown in Figure 5.3. CBS-TA-MLA starts with generating its first root node N_1 with the best task assignment $\boldsymbol{g}_a$ to a_1 and $\boldsymbol{g}_b$ to a_2 by applying Hungarian method to the cost matrix C, plans path for each agent with respect to its assigned task, and adds it to the OPEN list. N_1 is the only node in the OPEN list, the algorithm chooses to expand it. Since N_1 is a root node, the algorithm generates a new root node with the next best task assignment $\boldsymbol{g}_a$ to a_2 and $\boldsymbol{g}_b$ to a_1, plans paths and adds it to the OPEN list. The algorithm detects two collisions in the planned paths in N_1, $\langle a_1, a_2, B2, 1\rangle$ and $\langle a_1, a_2, B2, 3\rangle$, and it chooses the earliest collision $\langle a_1, a_2, B2, 1\rangle$ to resolve. The collision is resolved by creating two child nodes N_3 and N_4. The assignment, constraints (empty set so far) and paths of N_3 and N_4 are copied from N_1.

A new constraint $\langle a_1, B2, 1\rangle$, where a_1 is prohibited from being at location $B2$ at timestep 1, is added to $N_3.constraint$; and similarly, a new constraint $\langle a_2, B1, 1\rangle$, where a_2 is prohibited from being at location $B2$ at timestep 1, is added to $N_4.constraint$. As the low-level MLA* search can find valid paths for the replanning agents with respect to the constraints in each node, both child nodes can be added to the OPEN list. In the next iteration, N_2 is picked for expansion because it has the smallest cost $N_2.cost = 8$ in the OPEN list. No more root node will be created since there are only two possible task assignments. The paths in N_2 have collisions, and thus the algorithm generates two child nodes and adds them to the OPEN list (omitted in the figure). Then N_3 is chosen to expanded and the algorithm finds there is no collision among planned paths in N_3. So, N_3 is declared as the goal node and $N_3.assignment$ and $N_3.paths$ are returned as the result.

Chapter 6

Extensions

This section introduces three extensions of CBS-TA-MLA, namely an improved optimal version (with heuristics on the high-level), a bounded-suboptimal version and a greedy version.

6.1 CBS-TA-MLA with Heuristics (CBSH-TA-MLA)

CBS with heuristics [13] introduces three admissible heuristics for the high level search of CBS, which can significantly reduce the number of expanded CT nodes. Collisions in a CT node can be classified into three types:

1. cardinal collision, if both of the resulting child nodes have a larger cost than the node;
2. semi-cardinal collision, if only one child nodes increases the cost; and
3. non-cardinal collision, if both of the child nodes have the same cost as the node itself.

6.1.1 Multi-Valued Decision Diagram Construction

The technique used to classify collisions is based on Multi-Valued Decision Diagrams (MDDs) [13]. An MDD for agent a_i at CT node N is a directed acyclic graph consisting of all possible cost-optimal paths of agent a_i with respect to the assignment $N.assignment$ and the constraint set $N.constraints$. In the original MDD [13], each node $\langle v, t\rangle$ consists of a location v and a level/timestep t. A collision between agent a_i and agent a_j at timestep t is cardinal if the contested vertex or edge is the only vertex or edge at level t of their MDDs. To make the MDD applicable in our case in which the cost-optimal paths contain all goal locations of the assigned task in the correct sequence, we add a label l to each MDD node, and each node $\langle v, t, l\rangle$ is a tuple of location v, level t and label l (see Figure 6.1).

6.1.2 Collision Graph (CG) Heuristic

If $N.paths$ contains one cardinal collision, a cost of one is an admissible h-value for N since the cost of its child nodes should be at least one larger than $N.cost$. If $N.paths$ contains

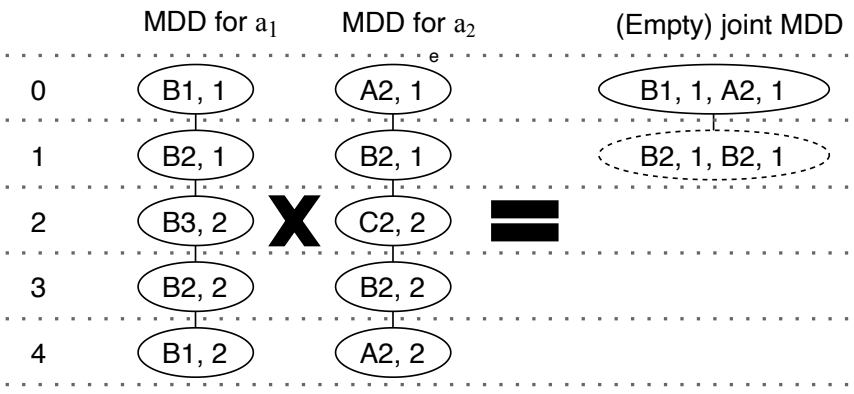

Figure 6.1: The MDDs and joint MDD for the example instance in Figure 5.2. Each node $\langle v, t(omitted), l\rangle$ contains a tuple of location v, level/timestep t and label l. Levels t of MDD nodes are shown on the left and omitted in the node.

multiple cardinal collisions, a collision graph of N is built, whose vertices represent agents and edges represent cardinal collisions between agents. The size of the minimum vertex cover (MVC) of this collision graph is an admissible h-value for N, which we call the CG heuristic.

6.1.3 Dependency Graph (DG) Heuristic

While CG only considers cardinal collisions, DG considers all collisions to decide whether two agents associated with the collisions are dependent or not. Two agents are dependent if (1) their paths have cardinal collisions, or (2) their paths have semi-cardinal or non-cardinal collisions and the joint MDD of their MDDs are empty. The joint MDD of two agents' MDDs consists all combinations of the cost-optimal collision-free paths of the agents that obey *N.constraints*. An empty joint MDD indicates that no such paths exist. Similar to CG, a pairwise dependency graph, a generalization of the collision graph, is built, where edges represent the dependencies between two agents. The size of the MVC of the pairwise dependency graph is an admissible h-value for N, which we call the DG heuristic.

6.1.4 Weighted Dependency Graph (WDG) Heuristic

WDG improves DG by building a weighted pairwise dependency graph. The weight of the edges between two vertices(agents) is the difference between the minimum flowtime of the collision-free paths of corresponding agents satisfying *N.constraints* and the flowtime of their paths in *N.paths*. The minimum flowtime of two agents' collision-free paths is calculated by running the two-agent CBS-TA-MLA algorithm (with or without CG heuristic). The size of the edge-weighted MVC of the weighted pairwise dependency graph is an admissible h-value for N, which we call the WDG heuristic.

6.1.5 Techniques application

We adopt the techniques of CBS to form the CBS-TA-MLA with Heuristics algorithm (CBSH-TA-MLA). We maintain a new variable min_f_val as the minimum f-value of all the nodes in the OPEN list. Each node N has two additional fields: $N.h\_val$ to represent the admissible h-value and $N.f\_val = N.cost + N.h\_val$ for prioritization in the OPEN list. Instead of calculating $N.h\_val$ immediately after generating a new node, we compute a cheaper and admissible h-value by a lazy computation method (refer to [13] for more details). Initially, the $R.f\_val$ for the root nodes are their actual cost $R.cost$ and the $R.h\_val = 0$. When a node N is chosen to expand, if $N.h\_val$ has not been computed yet, CBSH-TA-MLA computes the $N.h\_val$ according to the heuristic method and updates the $N.f\_val$. If the $N.f\_val$ is larger than min_f_val, N is put back to the OPEN list, and a new node is picked to expand. The chooseCollision(N) function chooses cardinal collisions first, semi-cardinal collisions then and non-cardinal collisions last (breaking ties in favor of the earliest collision).

6.2 Enhanced CBS-TA-MLA (ECBS-TA-MLA)

Similar to Enhanced CBS (ECBS) [1], ECBS-TA-MLA is a bounded sub-optimal algorithm for MG-TAPF. If uses a focal search instead of a best-first search with the same sub-optimality factor ω on both its high and low level searches. A focal search maintains two lists: the OPEN list and the FOCAL list. The OPEN list is a regular OPEN list as the best-first search, sorting nodes according to the f-values. The best node N_{best} is the node with the minimum f-value in the OPEN list, and we use f_{best} to denote the f-value of the node. The FOCAL list contains a subset of nodes of the OPEN list whose f-values are less than or equal to $\omega \cdot f_{best}$. The FOCAL list is sorted according to some other heuristic. It guarantees to find solutions at most ω times worse than the optimal solution by always expanding the best node (with respect to heuristic) in the FOCAL list.

6.2.1 Low-level focal search

The low-level focal search of ECBS-TA-MLA prioritizes nodes in the OPEN list with their regular f-value. The heuristic used for prioritizing nodes in the FOCAL list is the number of collisions between the current planning agent and the paths of the other agents. When it finds a solution, it returns not only the path, but also the f-value of the best node n in the OPEN list, which is guaranteed to be the lower bound on the cost of the time-optimal path. We denote this lower bound by $f_{best}(a_i)$, where a_i is the agent doing the low-level search.

6.2.2 High-level focal search

The high-level search of ECBS-TA-MLA sorts CT nodes in the OPEN list according to the sum of lower bounds of all agents $LB(N) = \sum_{i=1}^{m} f_{best}(a_i)$. Let N_{best} denote the node in the OPEN list with the minimum $LB(N)$. The FOCAL list contains a subset of CT nodes N such that $N.cost < \omega \cdot LB(N_{best})$. The nodes in the FOCAL list are sorted according the number of collisions among $N.paths$. Since $LB(N_{best})$ is provably a lower bound on the optimal flowtime C^*, the cost of any CT node in the FOCAL list is no higher than $\omega \cdot C^*$. As a result, once a solution is found, its flowtime is no larger than $\omega \cdot C^*$, so it is bounded sub-optimal with the pre-defined suboptimality constant ω.

6.3 Greedy CBS-TA-MLA (TA+CBS-MLA)

The greedy version of CBS-TA-MLA, called TA+CBS-MLA, is the best task assignment (TA) followed by the CBS-MLA algorithm. TA+CBS-MLA implements the same CBS-TA-MLA structure except for the single-root expansion. Thus, it starts with the root node with the best task assignment and does not generate any other root nodes. TA+CBS-MLA has no optimality or completeness guaranteed.

Chapter 7

Experiments

This section describes our experimental results on a 2.3GHz Intel Core i5 laptop with 16GB RAM. The algorithms are implemented in Python and tested on three maps: (1) a dense map which is a 20×20 warehouse map with 30% obstacles from the MAPF benchmarks [30] (Figure 7.1); (2) a sparse map which is a 32×32 random map with 10% obstacles (Figure 7.2); and (3) an empty map of size 32×32.

7.1 CBSH-TA-MLA

We evaluate CBSH-TA-MLA in three different maps in two test sets. In the first test set, we use the dense map (Figure 7.1) with 10 randomly generated agents/tasks and report the success rate, the average number of expanded CT nodes, and the average run time over 100 instances with a time limit of 120 seconds. In the second test set, we use the sparse map (Figure 7.2) and the empty map with the same size 32×32 and report the above three values over 50 instances with a time limit of 300 seconds. Table 7.1 shows that WDG always has the smallest number of expanded nodes, while DG always have the shortest average runtime of all instances. This is because WDG needs to compute the weights of the edges of pairwise dependency graph, which exquires executing the two-agent CBS-TA-MLA algorithm. Although we set a small time limit of 5 seconds and use the CG heuristic for the two-agent CBS-TA-MLA algorithm, it still requires a significant amount of time.

7.2 ECBS-TA-MLA

We evaluate ECBS-TA-MLA with two maps with different suboptimality bound ω, different number of agents/tasks and different number of goal locations in the task in two test sets. We use the dense map in our first test set and report the success rate, the average number of expanded CT nodes, the average run time and the average cost over 100 instances in Table 7.2. The last three values are averaged over instances that are successfully solved by ECBS-TA-MLA with all four different ω. As expected, ECBS-TA-MLA achieves a high success rate on the instances with smaller number of agents/tasks. However, it only achieves

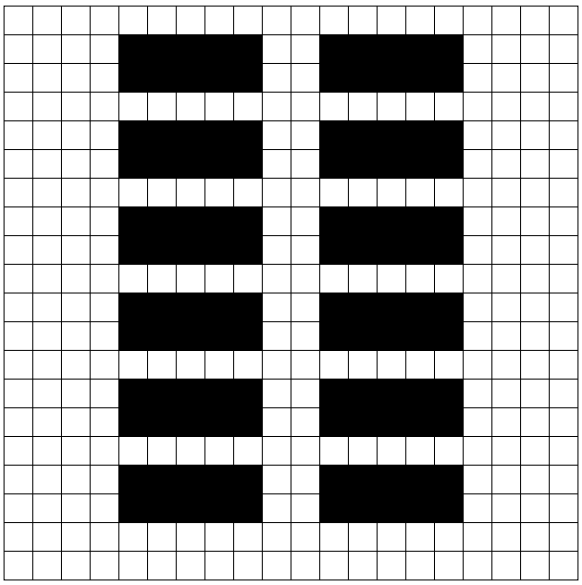

Figure 7.1: Dense map (20×20 warehouse map with 30% obstacles)

a high success rate with $\omega = 1.3$ when the number of agents/tasks is 30 on the dense map. In addition, when we increase ω, the average number of expanded CT nodes and the average run time reduce, while the average cost increases. The second test set uses the sparse map. We vary the number of agents/tasks and the number of goal locations in one task (2 goal locations per task and 10 goal locations per task). When the number of goal locations per task is 10, the success rate decreases as ω increases from 1.05 to 1.3.

We also test ECBS-TA-MLA on instances with different numbers of goal locations per task. We use 10 agents/tasks and report the same values as in Table 7.1 with a time limit of 60 seconds in Table 7.3. The experiment shows that with appropriate ω, the success rate is still over 80% within a minute, even with a maximum of 20 goal locations per task.

7.3 TA+CBS-MLA

We compare TA+CBS-MLA against CBS-TA-MLA on the dense map and report the success rate, average flowtime and the average runtime with 10 agents/tasks and 2 goal locations per task, 15 agents/tasks and 2 goal locations per task and 10 agents/tasks and 5 goal locations per task with a time limit of 60 seconds in Table 7.4. CBS-TA-MLA outperforms TA+CBS-MLA in the average cost, but TA+CBS-MLA outperforms CBS-TA-MLA in both success rate and average runtime in all three tests. The reason is that the number of possible task assignment with the same cost is large, so CBS-TA-MLA spends a significant amount of time computing task assignments and finding paths with the same costs.

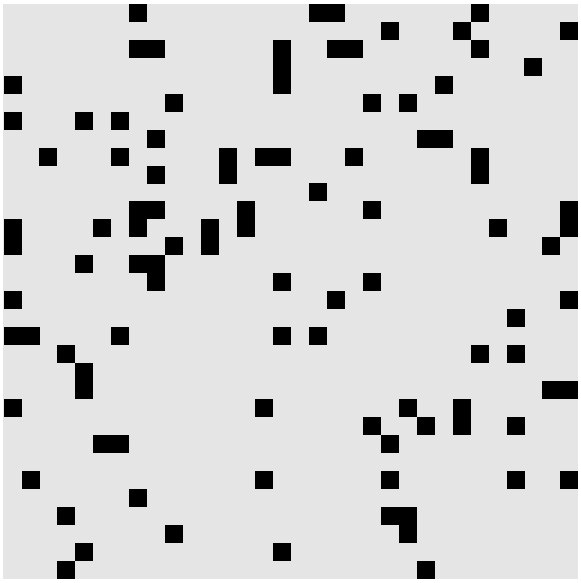

Figure 7.2: Sparse map (32×32 random map with 10% obstacles)

Maps	Agents	Heuristics	Success Rate	Nodes Expanded	Run time (s)
Dense map	10	No	98/100	34.18	2.54
		CG	98/100	28.86	2.23
		DG	98/100	25.54	**2.09**
		WDG	97/100	**8.98**	3.53
Sparse map	20	No	44/50	42.59	15.68
		CG	44/50	34.65	13.21
		DG	46/50	30.58	**11.8**
		WDG	46/50	**4.63**	13.17
Empty map	20	No	46/50	23.02	6.96
		CG	46/50	12.08	4.03
		DG	50/50	3.48	**1.52**
		WDG	48/50	**2.11**	2.82
	30	No	40/50	20.225	9.55
		CG	40/50	14.75	7.15
		DG	46/50	7.375	**4.24**
		WDG	46/50	**4.375**	4.52

Table 7.1: Results for CBSH-TA-MLA using different heuristics on three maps with different number of agents/tasks, where each task has two goal locations.

Maps	Agents	Goal Locations	ω	Success Rate	Nodes Expanded	Runtime (s)	Cost
Dense map	10	2	1.0	100/100	22.68	2.2	**144.69**
			1.05	100/100	6.0	0.7	145.57
			1.1	100/100	3.27	0.32	146.77
			1.3	100/100	**0.85**	**0.08**	148.19
	20	2	1.0	29/100	247.46	28.47	**249.46**
			1.05	78/100	20.42	2.22	255.75
			1.1	97/100	6.68	0.83	260.21
			1.3	**100/100**	**3.46**	**0.58**	262.32
	30	2	1.0	0/100	/	/	/
			1.05	14/100	/	/	/
			1.1	48/100	/	/	/
			1.3	**96/100**	/	/	/
Sparse map	10	2	1.0	100/100	3.14	0.58	**304.79**
			1.05	100/100	0.47	0.12	305.64
			1.1	100/100	0.36	0.11	306.32
			1.3	100/100	**0.35**	**0.10**	306.79
	20	2	1.0	76/100	28.34	9.22	**537.49**
			1.05	100/100	1.96	0.79	541.38
			1.1	100/100	1.48	**0.69**	543.28
			1.3	100/100	**1.35**	0.73	544.36
	10	10	1.0	73/100	9.33	13.35	**1846.46**
			1.05	**88/100**	0.68	4.52	1856.49
			1.1	86/100	**0.65**	3.56	1858.03
			1.3	85/100	**0.65**	**0.75**	1858.03

Table 7.2: Results for ECBS-TA-MLA with different ω on different maps with different numbers of agents/tasks and different numbers of goal locations per task.

Goal Locations	ω	Success Rate	Nodes Expanded	Runtime (s)
2-5	1.1	95/100	1.26	1.96
	1.3	95/100	1.26	2.13
5-10	1.1	72/100	2.04	3.51
	1.3	72/100	2	3.61
10-15	1.1	80/100	7.03	18.36
	1.3	83/100	7.2	19.46
15-20	1.1	78/100	14.61	48.11
	1.3	83/100	15.15	51.37

Table 7.3: Results for ECBS-TA-MLA with different numbers of goal locations per task with different ω on the sparse map. The number of goal locations differs in different tasks.

Agents	Goal Locations	Algorithm	Success Rate	Average Runtime	Average Cost	Cost Saving	Runtime Saving
10	2	TA+CBS-MLA	99/100	0.3957	152.4947	1.1578	-0.7597
		CBS-TA-MLA	95/100	1.1555	151.3368		
15	2	TA+CBS-MLA	94/100	2.9499	208.6086	2.3333	-7.8878
		CBS-TA-MLA	71/100	10.8377	206.2753		
10	5	TA+CBS-MLA	71/100	4.6151	395.8297	0.7234	-5.6129
		CBS-TA-MLA	48/100	10.228	395.1063		

Table 7.4: Results for comparison between TA+CBS-MLA and CBS-TA-MLA on the dense map with different number of agents/tasks and different number of goal locaitons.

Chapter 8

Conclusion and Future Work

In this thesis, we presented the CBS-TA-MLA algorithm to solve the MG-TAPF problems optimally. We then presented three enhanced variants of CBS-TA-MLA, namely (1) optimal variant CBSH-TA-MLA, which speeds up CBS-TA-MLA by adding heuristics, (2) bounded suboptimal variant ECBS-TA-MLA, which speeds up CBS-TA-MLA by using focal search, and (3) greedy variant greedy CBS-TA-MLA, which commits to the most promising task assignment and does not explore the other assignments. We conducted an extensive set of experiments to evaluate these algorithms in different settings with different maps, different numbers of agents/tasks, and different numbers of goal locations per task.

The future direction is to combine other enhancement techniques (such as disjoint splitting on the high-level search and incremental A* on the low-level search) to further improve the efficiency of CBS-TA-MLA.

Bibliography

[1] M. Barer, G. Sharon, R. Stern, and A. Felner. Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem. In *SoCS*, pages 19–27, 2014.

[2] 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.

[3] B. de Wilde, A. W. ter Mors, and C. Witteveen. Push and rotate: Cooperative multi-agent path planning. In *AAMAS*, pages 87–94, 2013.

[4] Kurt Dresner and Peter Stone. A multiagent approach to autonomous intersection management. *Journal of Artificial Intelligence Research*, 31:591–656, 2008.

[5] E. Erdem, D. G. Kisa, U. Oztok, and P. Schueller. A general formal framework for pathfinding problems with multiple agents. In *AAAI*, pages 290–296, 2013.

[6] Gilad Goraly and Refael Hassin. Multi-color pebble motion on graphs. *Algorithmica*, 58(3):610–636, nov 2010.

[7] 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.

[8] Wolfgang Hönig, Scott Kiesel, Andrew Tinka, Joseph Durham, and Nora Ayanian. Conflict-based search with optimal task assignment. In *AAMAS*, 2018.

[9] Wm. Woolsey Johnson and William E. Story. Notes on the "15" puzzle. *American Journal of Mathematics*, 2(4):397–404, 1879.

[10] D. Kornhauser, G. Miller, and P. Spirakis. Coordinating pebble motion on graphs, the diameter of permutation groups, and applications. In *Annual Symposium on Foundations of Computer Science*, pages 241–250, 1984.

[11] Harold W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.

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

[13] Jiaoyang Li, Ariel Felner, Eli Boyarski, Hang Ma, and Sven Koenig. Improved heuristics for multi-agent path finding with conflict-based search. In *IJCAI*, pages 442–449, 2019.

[14] Jiaoyang Li, Kexuan Sun, Hang Ma, Ariel Felner, T. K. Satish Kumar, and Sven Koenig. Moving agents in formation in congested environments. In *AAMAS*, pages 726–734, 2020.

[15] 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 *AAMAS*, pages 1898–1900, 2020.

[16] 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.

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

[18] H. Ma and S. Koenig. Optimal target assignment and path finding for teams of agents. In *AAMAS*, pages 1144–1152, 2016.

[19] 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.

[20] 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.

[21] Hang Ma, Glenn Wagner, Ariel Felner, Jiaoyang Li, T. K. Satish Kumar, and Sven Koenig. Multi-agent path finding with deadlines: Preliminary results. In *AAMAS*, pages 2004–2006, 2018.

[22] R. Morris, C. Pasareanu, K. Luckow, W. Malik, H. Ma, S. Kumar, and S. Koenig. Planning, scheduling and monitoring for airport surface operations. In *AAAI Workshop on Planning for Hybrid Systems*, 2016.

[23] Katta G. Murty. Letter to the editor - an algorithm for ranking all the assignments in order of increasing cost. *Oper. Res.*, 16:682–687, 1968.

[24] 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.

[25] G. Röger and M. Helmert. Non-optimal multi-agent pathfinding is solved (since 1984). In *SoCS*, 2012.

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

[27] G. Sharon, R. Stern, M. Goldenberg, and A. Felner. The increasing cost tree search for optimal multi-agent pathfinding. *Artificial Intelligence*, 195:470–495, 2013.

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

[29] 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.

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

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

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

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

[34] Glenn Wagner. *Subdimensional Expansion: A Framework for Computationally Tractable Multirobot Path Planning*. PhD thesis, Carnegie Mellon University, 2015.

[35] K. Wang and A. Botea. MAPP: A scalable multi-agent path planning algorithm with tractability and completeness guarantees. *Journal of Artificial Intelligence Research*, 42:55–90, 2011.

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

[37] J. Yu and S. M. LaValle. Multi-agent path planning and network flow. In E. Frazzoli, T. Lozano-Perez, N. Roy, and D. Rus, editors, *Algorithmic Foundations of Robotics X, Springer Tracts in Advanced Robotics*, volume 86, pages 157–173. 2013.

[38] J. Yu and S. M. LaValle. Planning optimal paths for multiple robots on graphs. In *ICRA*, pages 3612–3617, 2013.

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

[40] J. Yu and D. Rus. Pebble motion on graphs with rotations: Efficient feasibility tests and planning algorithms. In *Algorithmic Foundations of Robotics XI, Springer Tracts in Advanced Robotics*, volume 107, pages 729–746. 2015.